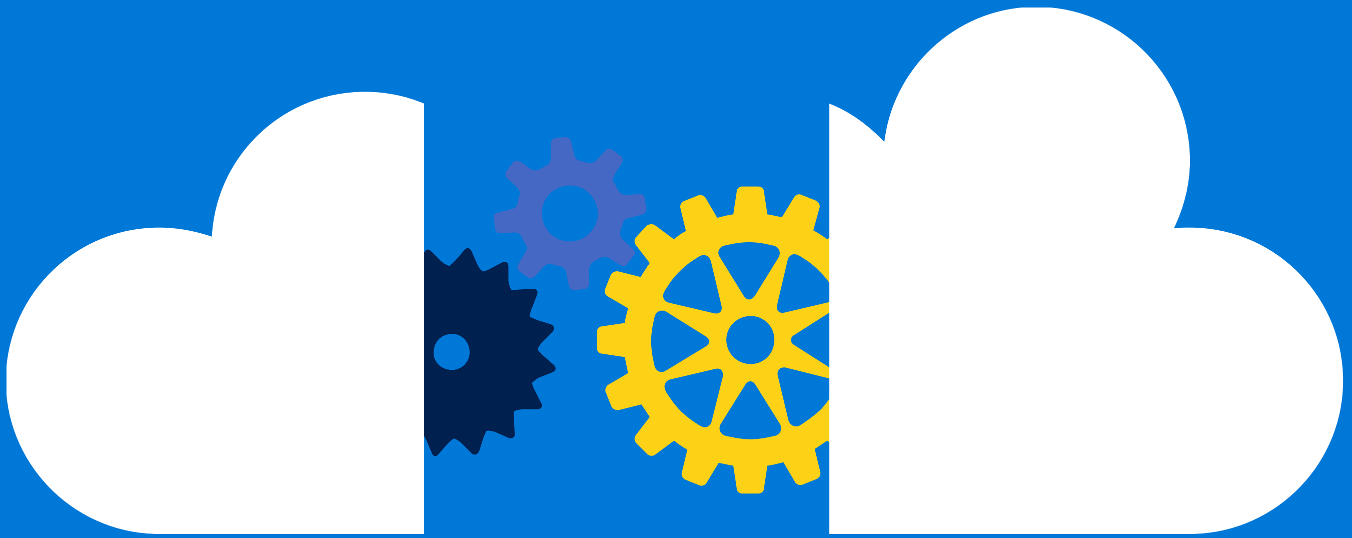
Microsoft



Cloud SOLID Part I

Cloud architecture and the single responsibility principle

By Casey Watson
Senior Program Manager
Azure Customer Advisory Team (AzureCAT)

June 2017

Contents

What is SOLID? 3

Contoso Outdoor Living 3

Contoso chooses Azure 5

The single responsibility principle 6

Platform as a service (PaaS) 7

Containers and microservices 8

Serverless 9

Conclusion 10

List of figures

Figure 1. Initial Contoso website architecture—lift-and-shift approach. 5

Figure 2. Revised Contoso website architecture based on PaaS. 7

Figure 3. Updated Contoso website architecture based on containers and microservices. 8

Figure 4. Contoso website architecture based on a serverless design. 9

Authored by Casey Watson. Edited by RoAnn Corbisier. Reviewed by Rick Rainey and Kirk Evans

© 2017 Microsoft Corporation. This document is for informational purposes only. MICROSOFT MAKES NO WARRANTIES, EXPRESS OR IMPLIED, IN THIS SUMMARY. The names of actual companies and products mentioned herein may be the trademarks of their respective owners.

What is SOLID?

First introduced in 2003 by Robert "Uncle Bob" Martin, SOLID principles are a set of prescriptive guidelines that, when followed, can help developers write code that is easy to comprehend, maintain, and test. SOLID is a mnemonic device used to help remember these five principles:

- **S**ingle responsibility principle
- **O**pen/closed principle
- **L**iskov substitution principle
- **I**nterface segregation principle
- **D**ependency inversion principle

While these widely-adopted principles have guided a generation of developers in creating higher quality code, they were introduced at a time when the information technology landscape was virtually unrecognizable from what it is today. In 2000, there was no cloud. There were no containers or microservices. Applications were composed of monolithic, tightly-coupled codebases that were deployed to bare metal servers in private, on-premises datacenters. Scalability meant adding additional CPU or RAM to existing servers. Development and operations lived in two disconnected worlds, resulting in painfully long release cycles.

Today's enterprises are adopting the cloud at an unprecedented rate. Along with this shift comes new, modern architectural patterns that favor smaller, more granular services over large, monolithic applications and blur the lines between development and operations. Now developers can decompose existing services into single-responsibility microservices and gain many benefits that we'll cover through this article. SOLID principles are still an essential tool for many developers. This article shows how can we adapt the first one—single responsibility—to a cloud-enabled world and gain scalable, maintainable cloud architectures.

This is the first in a series of five articles that focus on extending SOLID beyond code into the service layer and further into the cloud.

Contoso Outdoor Living

To illustrate how these principles can be applied to building highly modular and scalable cloud-based applications, we'll share the story of Contoso Outdoor Living's journey from on-premises to the cloud. Contoso Outdoor Living, a Seattle-based retailer, opened its first brick and mortar location in the mid-90s and sells "everything outdoors" from patio furniture to grills to camping equipment. Contoso has an extremely loyal customer base and has been experiencing rapid growth, opening three new locations in the last year alone, but experiences seasonal fluctuations in business.

Recently, Contoso expanded its presence in the outdoor fitness market with the acquisition of Adventure Works Cycling, a popular local cycling retailer. While Contoso has a basic online presence with limited e-commerce functionality, they have recently been losing more and more business to big box online retailers.

Mike is Contoso's CIO. Back in 1998, Mike built and helped launch Contoso's original website using tools that, at the time, were considered bleeding edge like Visual Basic 6 and ASP. Like most CIOs, Mike has limited staff and even more limited budget but is frequently encouraged by Contoso's board to "do more with less." While Mike has had a remarkably successful track record as Contoso's CIO, his technical skills have become quite dated and, per his own development team, knows "just enough about modern application development to be dangerous." Recently, Mike's focus has been on the daunting task of merging the Contoso and Adventure Works IT organizations.

Sharon is Lead Application Architect at Contoso. Sharon has been with Contoso for only three months. With deep background in .NET and web application development, she has recently been learning about the Azure platform through blog posts and tutorials. Sharon manages a small team of three junior .NET developers who spend most of their time maintaining the current website and various line-of-business applications.

During a recent board meeting, it was decided that Contoso is going to completely refresh its online presence to better compete with the larger online retailers. The overhaul will include modern features that customers have come to expect including: full-text catalog search, customer ratings and reviews, detailed order history, and shipment tracking.

Mike has committed to launching the new website in six months.

Mike realizes that this is an aggressive timeline and encourages Sharon to leverage the newly acquired Adventure Works development staff wherever she can. However, Mike has inadvertently introduced a new challenge—while the Contoso development team utilizes the .NET stack almost exclusively, the Adventure Works developers come from a MEAN (MongoDB, Express, AngularJS, and NodeJS) background.

During the initial design session, several important questions arise. How can we limit costs? How can we ensure scalability to meet fluctuating demand? How can we guarantee availability? Most importantly, how can we overcome the language barrier that now exists within the development team? Ultimately, the success of the project hinges on Sharon finding a common ground that will allow the development team to work together effectively across languages and platforms.

After researching different cloud platforms, Mike and Sharon decide that Azure meets all their application requirements and provides services that can augment their solution later, such as data and analysis services.

Contoso chooses Azure

Sharon soon creates an initial architecture and shares it with her team.

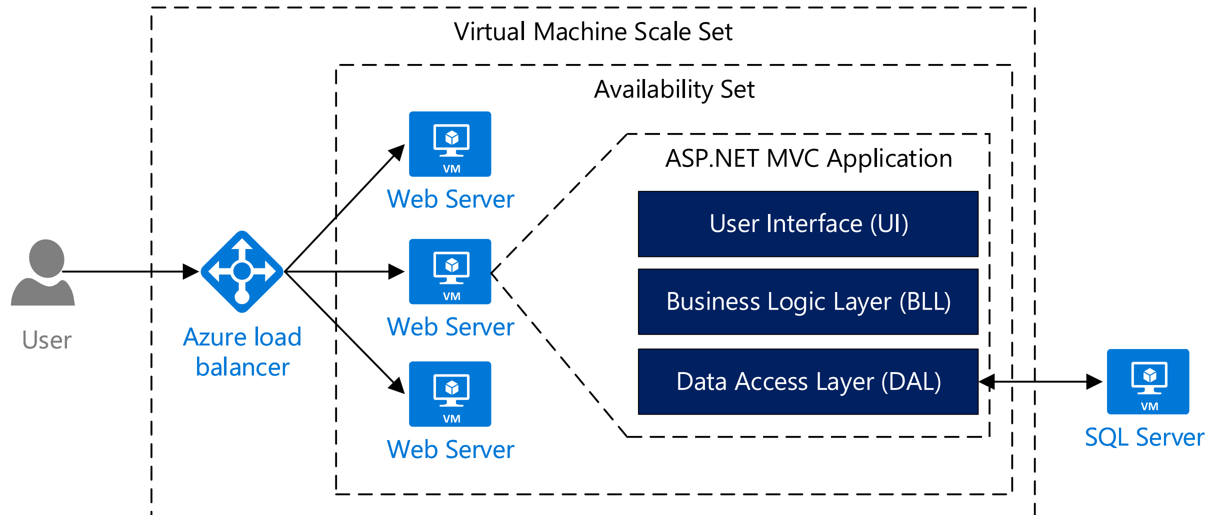


Figure 1. Initial Contoso website architecture—lift-and-shift approach.

In this design, user requests are load balanced across a pool of Windows virtual machines managed by a virtual machine scale set. Scale sets make it simple to create highly available and scalable pools of up to 1,000 identical virtual machines while automatically managing the complexities of the underlying network infrastructure. Served on each virtual machine by Internet Information Services (IIS), the web application itself implements a traditional three-tier architecture consisting of a user interface layer, business logic layer, and data access layer all rolled in to a single, deployable ASP.NET MVC package. On the back end, the application uses SQL Server hosted on a Windows virtual machine for persistence.

When organizations are first adopting the cloud, it is very common to attempt to lift-and-shift knowledge from on premises. In this case, Sharon has essentially replicated what she would normally deploy in Contoso's on-premises data center by leveraging Infrastructure as a Service (IaaS) and Windows virtual machines in Azure. While this is a common and perfectly functional approach, it makes it difficult to fully meet Contoso's requirements.

In this design, scaling involves adding or removing virtual machines from the scale set. We call this coarsely-grained scalability. This approach has a few critical limitations. First, scaling out can be slow. Windows virtual machines typically take several minutes to boot and start accepting requests, making it difficult to adequately meet rapidly fluctuating user demand. Second, depending on the size of the virtual machines, this approach can quickly become costly.

Virtual machines that belong to the same scale set also share one or more common availability sets. Sharon selected availability sets to ensure that at least some virtual machines are always accessible by evenly distributing them across multiple upgrade and fault domains, protecting against planned and unplanned maintenance events respectively. Together with availability sets, she uses managed disks to further increase availability by ensuring that virtual machine disks are sufficiently isolated from each other by placing them on different storage clusters. When she deploys two or more virtual machines to an availability set, Azure guarantees that at least one of

those machines will be accessible 99.95 percent of the time. But Sharon faces a problem with. This service level agreement (SLA), however, does not extend to the software running on the virtual machines nor does it guarantee that the website or database will be available.

Furthermore, because the architecture ultimately depends on virtual machines, Contoso's IT department is still responsible for configuring, maintaining, and patching the operating system, IIS, and SQL Server. This approach adds additional management burden to an already thinly-stretched IT department, which ultimately limits their ability to reduce costs.

Finally, and most importantly, the application is built on the ASP.NET MVC framework, making it virtually impossible for the MEAN stack developers to contribute.

How can Sharon apply the single responsibility principle—the "S" in SOLID—to overcome these challenges?

The single responsibility principle

Classes are one of the most fundamental building blocks of modern application development and the foundation of [object-oriented design (OOD)](). Classes consist of both state, exposed through fields and properties, and logic, exposed through methods. Applications that adhere to the single responsibility principle consist of many small classes, each of which have only one responsibility or reason to change, that are used collectively to build higher-level features. Having more, smaller, focused classes makes applications easier to maintain and test.

> *"A class should have only one reason to change."*
>
> *—Robert Martin*[1]

Revisiting Contoso's ASP.NET MVC application, how can the development team apply this principle when building out the ordering functionality? The developers could group all the ordering logic into one class. This approach violates the single responsibility principle. Instead, they should break the ordering functionality down into several smaller classes that handle everything from tax calculation to processing credit card payments to persisting order details to the database. When adhering to this principle, a developer that later changes how taxes are calculated is less likely to introduce a bug that impacts credit card processing.

Although the code itself may adhere to the single responsibility principle, the overall application architecture does not. The entire site is packaged into a single, deployable ASP.NET MVC application that has many responsibilities. The application has many reasons to change, ranging from updating the user interface to changing how orders are processed. If a change needs to be made to any part of the application, an entire tier or possibly even the entire application must be redeployed. More critically, the recently added Adventure Works developers are sitting idle and unable to contribute to the application, as they are not familiar with .NET.

Let's focus on one word for a moment—responsibility. Although Contoso is hosting the application in Azure, they are still responsible for maintaining the virtual machines on which the

[1] Martin, Robert C. *Agile Software Development: Principles, Patterns, and Practices*. Pearson. October 25, 2002.

application runs. How can Sharon modify the architecture to reduce the management burden on Contoso's IT department?

Platform as a service (PaaS)

PaaS greatly simplifies cloud application development. Developers can access common application services such as database, messaging, and storage in seconds without having to worry about the complexities of the underlying infrastructure.

Sharon returns to the whiteboard and updates the application architecture to leverage PaaS.

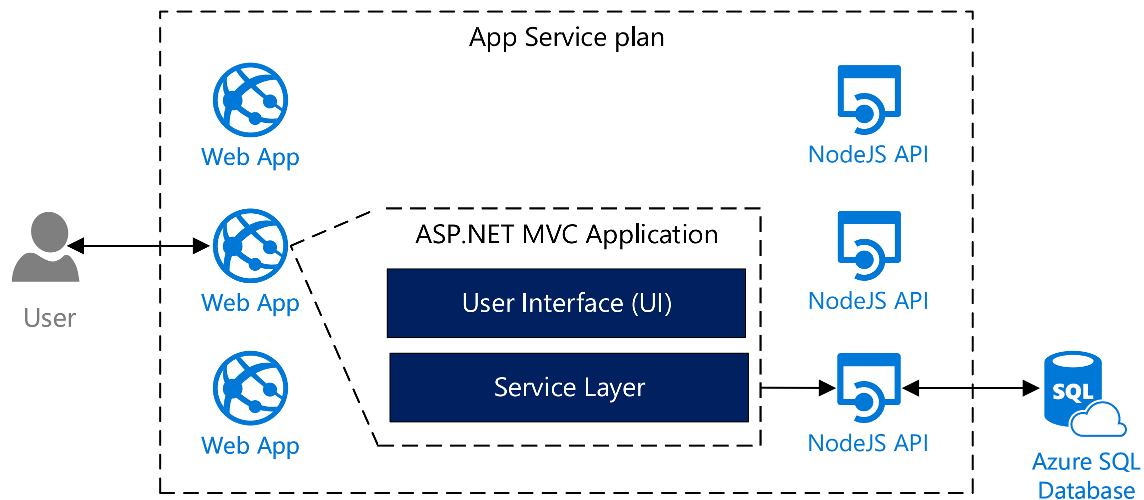


Figure 2. Revised Contoso website architecture based on PaaS.

All the virtual machines have disappeared. The application has been broken down into a front-end ASP.NET MVC application and a back-end NodeJS REST API. Both components are hosted within an App Service plan, a fully managed pool of compute resources capable of hosting a variety of different types of applications. App Service supports common development languages, including .NET, Java, NodeJS, PHP, and Python. (An alternative method would be to expose a web API on the back end and write a single-page application for the interface.)

Notice how Sharon has tried to apply the single responsibility principle in the updated design. The web application is responsible only for presenting the user interface, relying on the back-end API for business logic and data access. This change will finally allow the MEAN stack developers to start contributing, using the languages they already know such as NodeJS. However, the API still has multiple responsibilities—its scope hasn't changed, so it doesn't fully reflect the single responsibility principal. In the following sections, we'll examine ways that Sharon can break the API down further into smaller, more granular components.

In this architecture, Contoso does not have direct access to the physical virtual machines. That's OK, though, because Azure is now managing the virtual machine's operating system and security updates, leaving more time for the development team to focus on the application itself. This is the tradeoff that you make when you choose PaaS—more automation for less control. Autoscaling makes it incredibly simple for Contoso to automatically meet fluctuating user demand and, starting with the App Service plan basic tier, Azure provides a SLA guaranteeing that the web application and the API will be reachable at least 99.95 percent of the time.

Sharon has also made another important change by replacing SQL Server hosted on a virtual

machine with Azure SQL Database. Azure SQL Database allows Contoso to create databases up to 4 TB in size while automatically managing the complexities of high availability, disaster recovery, security, and scalability behind the scenes. Again, Contoso cannot directly access the physical infrastructure hosting Azure SQL Database, but Azure does provide a SLA guaranteeing that the database will be accessible at least 99.99 percent of the time.

In a more abstract way, this architecture also implements the single responsibility principle at a platform level. Contoso is now only responsible for creating the application logic that is deployed to the underlying platform that Azure is responsible for managing.

Containers and microservices

The changes that Sharon has made to the architecture have both unlocked the MEAN stack developers and removed significant management burden from Contoso's IT department. There are, however, still significant improvements that can be made. While the front-end web application is now responsible only for the user interface, the back-end API layer still has several loosely-related responsibilities that don't necessarily belong in the same service.

Containers and microservices are driving a revolution—enabled largely by the proliferation of cloud platforms like Azure—in the way that we think about modern application architecture. Microservices promote breaking application logic down into small, independent, granular services that focus on specific business areas. Containers offer an elegant platform-agnostic solution for developing and deploying these microservices.

Sharon again updates the application architecture leveraging these two key concepts to further decompose the API layer and more evenly distribute responsibility.

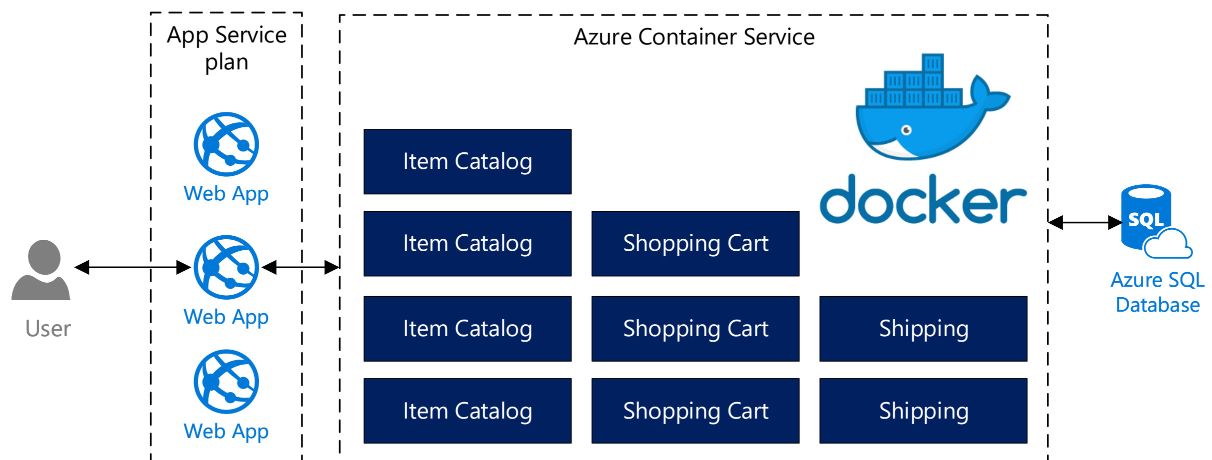


Figure 3. Updated Contoso website architecture based on containers and microservices.

The NodeJS-based API layer, previously hosted as an API App in a shared App Service plan, has been broken down further into a collection of independent microservices hosted within Docker containers. They may consider decomposing the presentation layer at a future date, but in the meantime, this update allows Contoso to scale different functional areas of the application independently while making it easier for the development team to update the API layer with minimal downtime. Are customers spending more time browsing the item catalog and less time tracking shipments? More Item Catalog containers can be added to the cluster while the number of Shipping containers can be reduced. If the development team needs to change the way that

items are removed from shopping carts, the Shopping Cart microservice can be updated without impacting other parts of the application. By continuing to apply the single responsibility principle to the application architecture, Sharon has increased agility, potentially driving down costs, and improved the overall availability and user experience of the application.

While it's easy to get started with Docker and spin up your first Hello World container, managing a network of interconnected containers like Contoso's API across a pool of virtual hosts can quickly become complex. This is where orchestration comes in to play. Orchestrators provide services like autoscaling, intelligent load balancing, and health monitoring that greatly simplify the management of distributed container-based applications and help ensure high availability. The Azure Container Service (ACS) rapidly provisions production-ready Docker clusters, preconfigured with your choice of popular open source orchestrators: Docker Swarm, Mesos DC/OS, or Kubernetes on the Azure platform.

Serverless

While the combination of containers and microservices provides an elegant, cloud-native solution to building loosely-coupled, highly-scalable applications, it is still rooted in a world driven by processes with their own independent lifecycles hosted on virtual machines that need to be managed by higher-level systems. What if Sharon could break these services down to the most independent atoms of business functionality and rely completely on the platform to transparently manage availability and scalability? What if the NodeJS consultants could forget about the glue that binds application logic to the underlying platform and focus exclusively on delivering new features? Serverless architecture makes this dream a reality and is the point at which the service-level and code-level definitions of the single responsibility principle ultimately converge.

Serverless architecture has been experiencing explosive growth spawning several open source frameworks and even a major international conference. This architecture focuses on the smallest independent unit of business logic—the function. When we compare this architecture to the microservice approach described in the previous section, we can also think of these functions as nanoservices.

Sharon again revises the application architecture, implementing a serverless design.

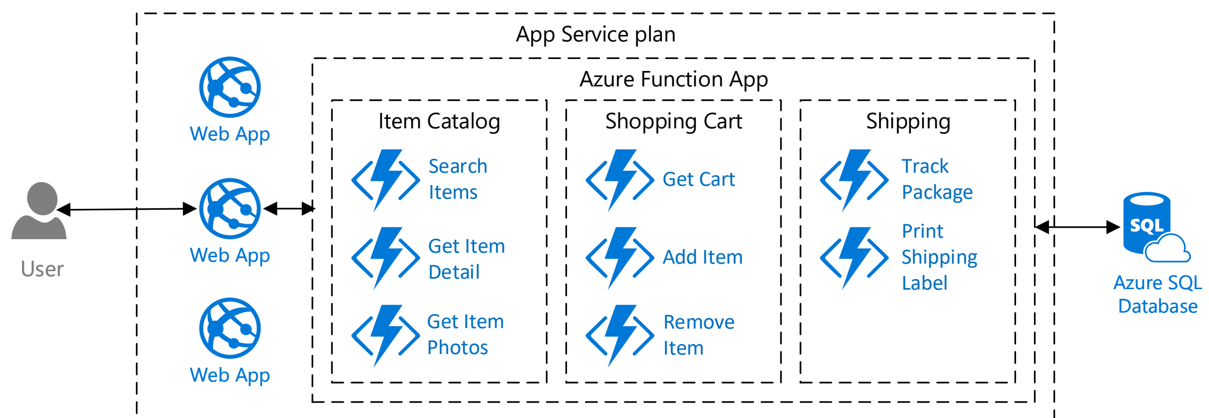


Figure 4. Contoso website architecture based on a serverless design.

In the updated architecture, Sharon has broken down the microservices introduced in the previous section into logical services implemented using Azure Functions. Functions are

fundamentally different from what we have seen before in that they are ephemeral. A function's lifetime is bound only to its execution. Due to their transient nature, functions are implicitly scalable and can yield significant cost-saving benefits over traditional service models.

Functions can be triggered by a variety of sources including Azure Service Bus, Azure Blob Storage, and third-party services like GitHub. They can also be invoked on a configurable schedule. Functions can be created in a variety of languages including C#, F# and, ideally for Contoso, Javascript (NodeJS).

In this case, Contoso's functions are triggered by HTTP requests made by the front-end web interface. In this configuration, each function essentially becomes its own API—a nano-API, if you will. Each function also exposes a unique URL that, as more functions are inevitably added to the back end, creates additional complexity for API consumers. The Azure Functions Proxies feature, still in preview at the time of this writing, helps solve this problem by presenting a single API surface, or façade, in front of a collection of back-end Azure Functions. API consumers interact with this façade instead of directly with the functions. This approach allows you to restructure your API endpoints in a more organized way making it easier for developers to work with your API.

Azure Functions offers two kinds of pricing plans—Consumption or App Service. When using the Consumption plan, Azure transparently provides all the computational resources, and customers are charged only for the time that their code is running. Customers can also choose to host their functions as part of an App Service plan at no additional cost. Since Contoso is already using an App Service plan to host its front-end web application, Sharon naturally chooses that plan.

Conclusion

Earlier in this article, we defined the single responsibility principle as "a class should have only one reason to change." As we followed Contoso on their journey from on premises to the cloud, we saw how Sharon expanded this idea beyond the code to create highly scalable and maintainable cloud architecture. While it's important that Contoso's developers continue to adhere to this practice when writing code, from an overall architecture perspective, we can augment this principle and say also that *a service should have only one reason to change*.

Applying the single responsibility principle at a service level highlights the importance of DevOps. Deploying and configuring a single, monolithic application is relatively simple but managing a constellation of granular back-end services can quickly become a nightmare. Automation is key when managing a microservice application.

This architecture works well for Contoso but that doesn't mean that it is the right answer for every project. Depending on your application, you may choose Azure Container Service and Kubernetes. You may choose Azure App Service. You may have legacy application dependencies that require IaaS and virtual machines. Each one of these approaches has strengths and weaknesses that must be carefully weighed within the context of your application's requirements. Often, the answer is not this service *or* that service; it's this service *and* that service. For example, in Contoso's design, they used both App Service and Azure Functions to create a highly flexible architecture.

It's also important to remember that SOLID principles are guidelines, not concrete rules. The uncontrolled proliferation of services comes with its own headaches. When designing your cloud architecture, it's important to balance the single responsibility principle against potential management burden and, in general, common sense.