Google Cloud Platform

Taking the Cloud-Native Approach with Microservices

by Andy Wu, Solutions Architect, Magenic

Table of Contents

Executive summary

Although n-tier, monolithic architectures are the norm for most applications running in production today, they are often not the best fit for complex cloud-based systems. The primary driver for cloud adoption across businesses both big and small is the need for agility and flexibility in the face of accelerating innovation and disruptions from competitors. Yet, more and more companies are finding out that simply moving and shifting their legacy system to the cloud does not sufficiently meet their needs. Their systems, built with a monolithic architecture, are holding them back from realizing their goals.

In this whitepaper, we discuss the problems with monoliths and why we need a different architecture to maximize our cloud investments. We also discuss how this new architecture is better suited for the cloud and, finally, we will walk through a real-world scenario where we illustrate the process of migrating a working monolithic system and transform it into this new architecture.

Why monoliths are a suboptimal architecture for the cloud

Over the years, most clients we at Magenic have engaged with have given us three primary reasons why they're interested in moving to the cloud as follows, and in no particular order:

- Cost savings
- Better agility in managing their computing infrastructure
- Increased velocity with respect to system releases

Over time, however, we have seen a trend develop: Of these three, the need to increase velocity is bubbling up as the top reason companies are interested in the cloud. The motivation is obvious: it is a matter of survival. As software increases in strategic importance to the business, companies are looking for ways to differentiate themselves through their digital presence—digital transformation, if you will. To achieve this transformation, they need to make rapid changes and adjustments to their systems as they find out more about their customers through sensors and other data collections. Failure to iterate and quickly adapt to their customers' needs makes them vulnerable to competitors that will gladly take their customers away. Given the need for businesses to call for systems to be modified, scaled, and updated at an increasing rapid pace, let's examine why monoliths are standing in the way.

Defining a monolith

Before we describe the issues with monoliths, let's first make sure we are on the same page about what they are. A monolithic application is built as a single deployable unit, e.g. a single WAR file in Java or a single Web Application/Web Site in .NET. They are usually built in three

parts: a database (consisting of many tables in a relational database management system), a client-side user interface layer (consisting of HTML pages and/or JavaScript running in a browser), and a server-side application. The server-side application handles HTTP requests, executes some domain specific logic, retrieves and updates data from the database as needed, and generates the HTML pages to be sent as response to the client browser. Monoliths are developed with object-oriented principles; they are usually long-lived, and are of critical importance to the health of the business. Because of the complexities, they have wide and deep class hierarchies, and many interdependencies between them.

The Canonical Monolithic Application Design
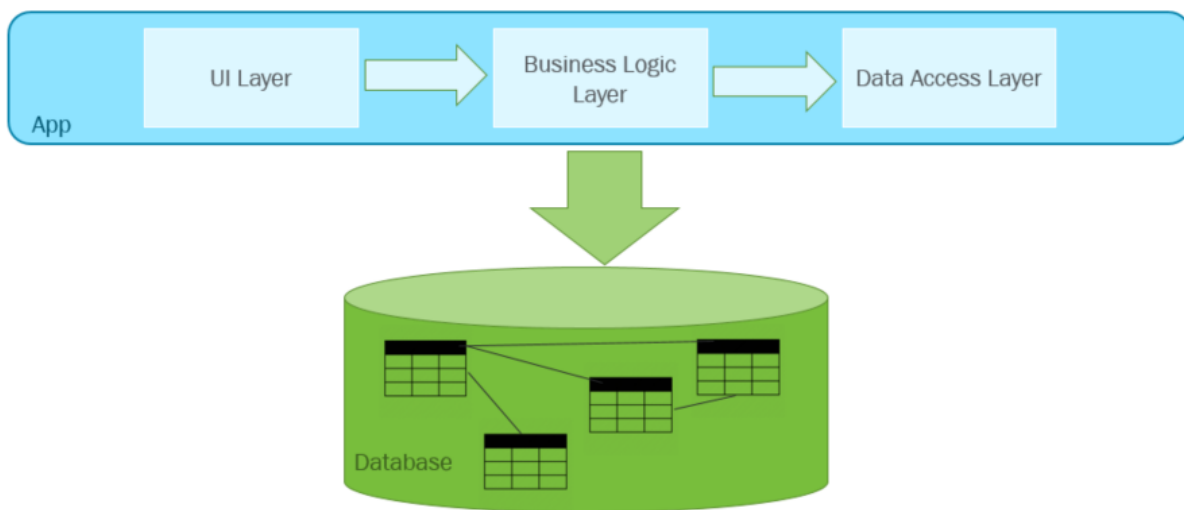
Fault isolation cannot be contained

As mentioned above, monoliths have many inter-dependencies and a large scope. As a result, features usually touch many parts of the system and inevitably cause unintended side effects. By definition, monoliths are built and deployed as a single unit; there is no physical separation between different areas of the systems. Thus, there is no way to guarantee any new release will only affect the area(s) they are targeted for. Even with a full regression test, there is no guarantee of change isolation—unintentional side-effects are always a possibility.

Hard to scale properly

Scaling "logical" parts of a monolith is difficult and requires significantly more resources than just that functionality demands. In most cases, the only available path for scaling a monolithic system is to deploy multiple instances of the entire monolith application, resulting in more overall memory and computing resource usage. For instance, say you need to increase the throughput of an ordering subsystem on your website. You'll need to either deploy an additional instance(s) of the entire application and setup load balancers or get a VM with more computing power. Even then, it may not necessarily solve the issue. The scalability problem might cause

database-locking issues and adding more instances of the monolith might actually make things worse. Bottom line: scaling parts of a monolith application requires a lot of resources—the antithesis of agility.

Deployments are cumbersome and time consuming

Although the monolithic approach is the prevailing type of architecture today, it becomes a bottleneck in complex, large-scale systems. It requires longer and longer development and QA cycles as the system grows in features and complexity. In addition, the monolithic model is, at best, inconvenient if you need to make frequent changes to your deployment. To introduce minor changes to a single component or add a new feature, you have to re-build, do full application regression testing, and deploy the entire monolith again. As a result, updating monolithic applications is time-consuming and requires proper coordination of all the people involved in the project, another contributing factor that prevents faster release cycles.

Last but not least

The final downside we cite with monolithic architecture systems is their propensity to require a long-term commitment to a particular technology stack. Layers in monoliths are tightly coupled in-process calls, and they are usually developed with the same technology for the sake of interoperability. With each release of the system, you're essentially increasing your commitment to the existing technology stack. As more code is developed in that stack, the harder it is for developers and architects to switch or experiment with another technology stack when they become available, not to mention not being able to easily take advantage of new technologies. If you believe that "software is eating the world," being able to adopt new technologies when they become available is a key factor in being able to compete on an equal footing.

An architecture optimized for the cloud

Applications in the cloud break all the traditional application rules; they can move around failures in order to provide resiliency and scale when workloads change (sometimes even without warning). Components within the application can be mixed and matched to speed development and improve deployment efficiency. However, none of these benefits happen automatically simply by moving to the cloud. With a few exceptions such as cost saving and faster provisioning, we've learned that the same old application architectures running in the cloud end up running the same old way as when they ran in your data center, and fundamentally fail to achieve the promises businesses were looking for. To truly maximize benefits of the cloud, we need to change our application architecture and models.

Pioneered by Netflix and others in Silicon Valley, microservices architecture has emerged as a prime candidate for maximizing the return for companies making the move to the cloud. The canonical definition of microservices as "loosely-coupled, service-oriented architecture with bounded contexts" is often attributed to Adrian Cockcroft, who led the microservices effort at Netflix. On the surface, microservices sound very similar to the Services Oriented Architecture

(SOA) that was quite popular during the last decade, but to be sure, there are distinct differences between the two which we will go into detail later on in the paper. The core premise of microservices is that by modularizing a system into small services that provide well-defined, narrowly scoped APIs, coupling is dramatically reduced while making it easier to implement functionality using well-accepted design principles such as the separation of concerns (SoC) and the single responsibility principle (SRP). At its core, microservices is a decomposition technique for overcoming system complexity. It accomplishes this goal by splitting complex systems into multiple independent, narrowly focused services, each with its own isolated business logic and data store. In microservices-based applications, any service can be scaled and deployed separately. Most of all, several teams can simultaneously work on different modules to enhance the overall system's time-to-market without the risk of stepping on each other. That is why large number of software companies large and small are embracing this architecture. They know it is the only way they can achieve the release velocity needed to support their business model.

The modularity introduced by this architecture offers ease and speed-of-development to match the accelerated pace of the business. Updating a subset of functionality with guaranteed isolation is now made possible since it is localized in one service boundary. Also, the modular nature of microservices inherently enhances security and fault isolation. Indeed, if a given set of code is corrupted or compromised, it is adequately isolated from the remaining services, hence preventing the entire application from becoming unavailable thus increasing the overall uptime of the system.

Another difference between monoliths and microservices is isolation of the data layer. Each microservice is supposed to be self-contained; they do not share a data layer, giving them complete autonomy. Each one may have its own datastore and load balancer. Isolation is a critical component of microservices architectures; different microservices may require different scaling characteristics and storage technology. For instance, some microservices might use relational databases whereas others might employ NoSQL databases or even mounted file systems. Building applications this way increases the scalability of teams building applications. With monolithic code, it is common to have one big team of people working on one large section of code and stepping on each other's feet all the time. The speed of development slows exponentially with the growth of the system. With microservices architecture, apps are built by small, decentralized development teams that work and change microservices independently. This makes it easier to test and upgrade services and add functionality over time. Eventually, if one microservice grows in size and functionality, it can be broken down and separated into multiple microservices, keeping microservices small, manageable, and autonomous. That is how the Netflix came to be made up of incredibly large number of microservices.

The following diagram gives a pictorial view of the differences between the two architectures:

Monolithic Architecture

Microservices Architecture
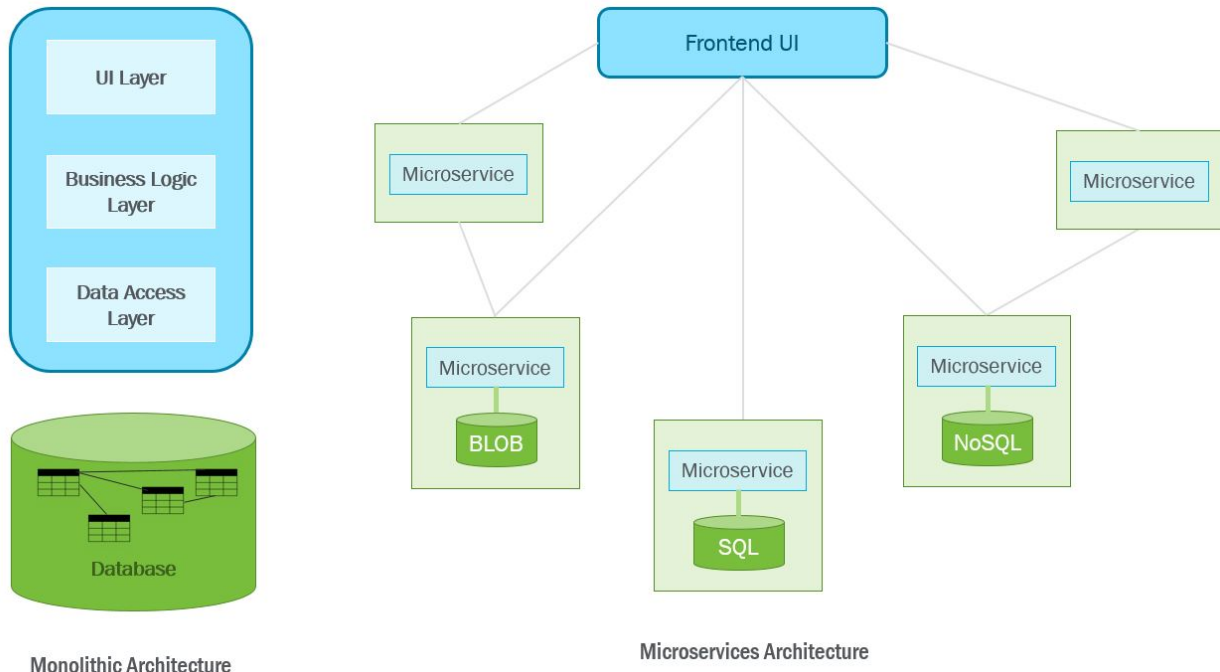
Lastly, adopting a microservices architecture allows for individual services to be written in any language and the technology stack that's most appropriate for its development. There is no hard restriction that all microservices must be developed with same technology, just as long they all communicate over the same lightweight protocol, such as HTTP and messages, and data structures are serialized over the same format, JSON being the most popular choice. This frees up teams to use any stack they feel comfortable with, alleviating the talent recruitment issues one can have with monolithic architectures, and enabling teams to select the best technology for their needs.

Nothing is free in system architecture

Despite all the benefits, the microservices approach is not a panacea. While alleviating many of the issues inherent to monolith applications, microservices also create other challenges. As with anything in technology, there are always tradeoffs with different architectures and microservices are no different. What it offers in agility and speed of development comes at a cost of increased operational complexity as there are naturally more moving parts (or services)—perhaps many more than with a monolith. (Fortunately, just like any other technology hurdle, other technologies come to the rescue, as we will touch on in other whitepapers in this series.)

Using a microservices architecture will likely increase operating overhead. With this approach, your deployment may require significantly more resources simply because there are a larger number of deployments. As a result, you may need more time and effort to create the infrastructure. All services potentially need clustering for failover and resilience. Your system may have dozens of separate components, and as you add new features, it will become increasingly complicated. Instead of a single monolith system, you may get a solution that

consists of 20, 30, or more services, each running multiple processes. As a best practice, you should address this added overhead with automation, putting a premium on staff that are skilled at DevOps and other infrastructure automation methods.

Haven't we seen this before?

Before we wrap up the introduction on microservices, I want to briefly address the often-asked question of "Aren't microservice just SOA (service oriented architecture)?" SOA services are typically implemented in deployment monoliths; however, microservices must be independently deployable. Microservices philosophy is fundamentally about removing the burden that application and database monoliths place on systems. It is about creating highly-distributed, autonomous, and horizontally scalable applications. The hallmarks of microservices are:

- Lightweight components
- Independent deploy-ability
- Lightweight, coarse-grained APIs
- Lightweight service bus
- Lightweight data storage

The approach for the journey from monolith to microservices

Now that we have discussed the issues with monolithic systems and why a microservices architecture is a better fit for those who are seeking to maximize their investments going to the cloud, let's walk through an actual example of how one would take a monolithic system and decompose it into a cloud-optimized microservices-based system.

Systems of strategic value to companies never die; they are just refactored (or if one prefers, "modernized"). A microservices transformation should be treated like any other app refactoring and modernization effort: very carefully with a risk-mitigated approach. One strategy we will not recommended is the "Big Bang" rewrite. That is when you focus all of your development efforts on rewriting the system with a new microservices based architecture from scratch. As appealing as it may sound, since it offers the opportunity to start with a clean slate, it's not a practical approach. In the real world, rarely, if ever, would one be able to get the approval of a budget to embark on such an effort, and it is also extremely risky from a project management perspective. It requires a full understanding of the entire application up front, which can take an extraordinary long period time to achieve.

Instead of a Big Bang rewrite, we recommend a more gradual approach, where you incrementally refactor a monolithic system, gradually turning it into a "new" application consisting of microservices. By doing this over time, the amount of functionality implemented by the monolithic application shrinks until either it disappears entirely or it becomes just another

microservice. Lastly, don't feel the need to start decomposing everything immediately; take the time and work on what makes the most sense for your team.

A word of caution

Before we start the actual migration process, we want to sound a word of caution: to be certain, a system with the microservice architecture offers a great deal of benefits, for example independent deployment, strong subsystem boundaries, and technology diversity. However, it does come with a price tag of increased complexity related to distributed application development (e.g., independent data models, resilient communication between microservices, eventual consistency, and operational complexity). These aspects will contribute to a higher level of complexity than a traditional monolithic application. Even with proper tooling and design, developing and running a multitude of distributed services at scale is not an easy task, and certainly one that you should only embark on after careful consideration. If you have a system that has a stable feature set with well known system load, e.g. an internal expense reporting system, there is nothing wrong with simply lifting-and-shifting this system onto the cloud; you will still reap the many benefits cloud computing offers. For details on the how-to's of lifting-and-shifting legacy systems onto the cloud, see our [whitepaper](#) on this topic.

Enough talk, let's make it real!

We will leverage a popular Reference Implementation, Microsoft .NET PetShop 4, to illustrate our migration process. PetShop is a system well known in the Windows/.NET community (prior to that, in the Java community as well). It is a well-designed and architected system based on the concept of proper layering of different concerns within the system. As you can see from the illustration below, all of the various components and functions needed by the systems are properly segregated and compartmentalized. It is a great representation of many of the systems running in production today, and makes a great system to illustrate our migration journey.
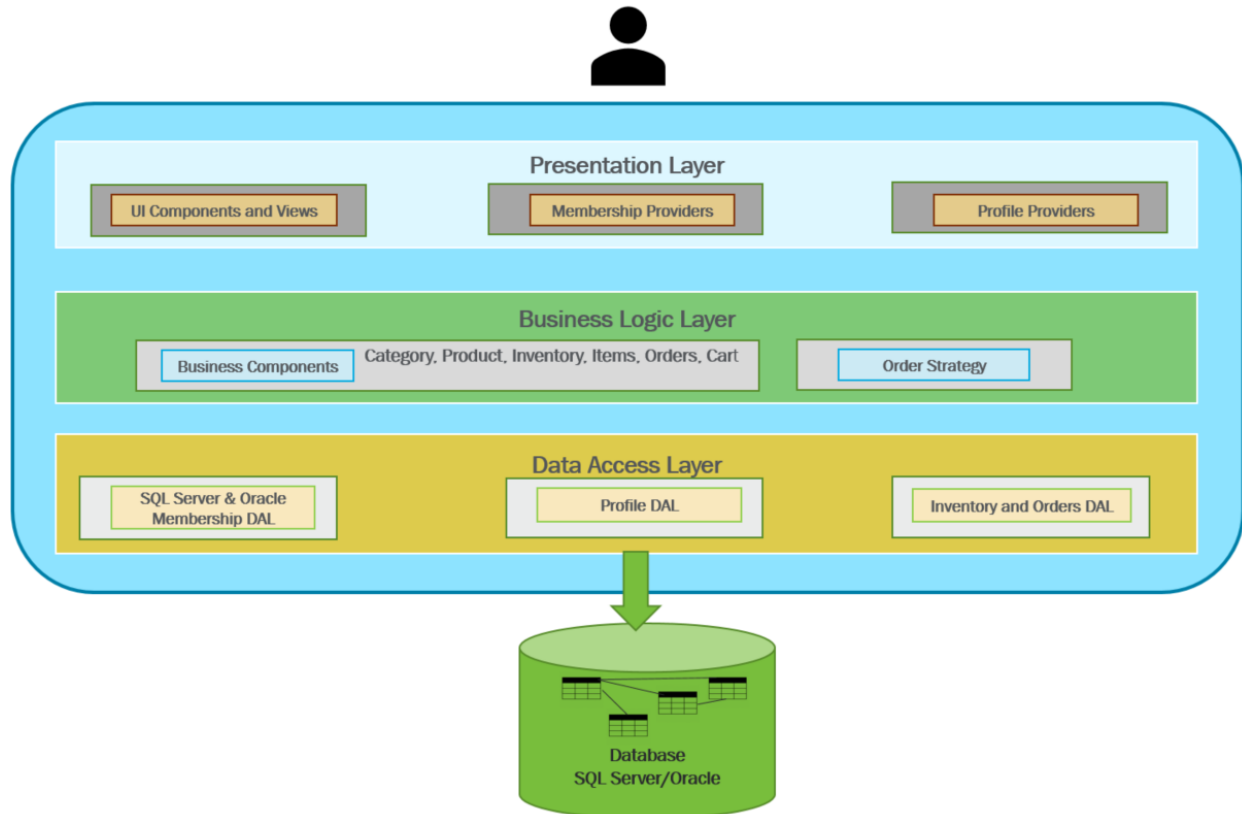
High level PetShop System Architecture

The starting point

In most web applications, there are typically three different layers that may or may not be deployed into different physical tiers:

- Presentation layer – Components that handle browsers (or HTTP) requests and implement an HTML–based web UI.
- Business logic layer (BLL) – Components that are the core of the application and implement business rules.
- Data Access layer (DAL) – Components that access infrastructure components such as databases and message brokers.

If designed properly, there is usually a clean separation between the presentation and business logic layer, and fortunately, PetShop falls into this category. It has a well thought out Business Logic Layer with explicit interfaces (as in actual .NET Interfaces) that can be leveraged for pluggable implementation. With its current implementation, PetShop Business Logic Layer makes in-process calls to its Oracle-based Data Access Layer for interactions with the database. Given its pluggable nature, we can seamlessly replace the existing implementation

with remote calls to a service based implementation, as long as we preserve the method call signatures.

Splitting a monolith in this manner enables you to develop, deploy, and scale the web site independently of the services. In particular, it allows the presentation–layer developers and QA testers to iterate rapidly on the UI and easily perform A/B testing, verifying the correctness of the new services.

How does one define the boundaries for microservices?

Now that we have defined a seamless and easy way to "inject" our service implementations into the existing code base, we need to turn our focus to defining the scope, or service boundary, for each of microservices we will be developing.

Right from the beginning, I want to stress size should not be the determining factor for service boundaries. There seems to be a lot of chatter in the community about how small a given microservice needs to be in order to be called "micro", and I suppose that stems from the name of the architecture. Rather than focusing on the size, we instead need to focus on the key tenets of the architecture: to achieve system agility by decomposing complex systems into multiple independent, narrowly focused services, each with its own isolated business capabilities and data store. Thus, the emphasis needs to be placed on the "narrow focus" of business capabilities and the autonomous nature of each service, not the size.

Rather than using size to drive the design of microservices, let's talk about Domain-Driven Design (DDD). DDD is a methodology that advocates modeling based on the practical use cases of the actual business. In its simplest form, DDD consists of decomposing a business domain into smaller functional chunks, at either the business function or business process level, so that the complexity of both a business and problem domain can be better understood and resolved through technology. This aligns very well with the goals of microservices and is seeing a resurgence in popularity as a result. DDD also encourages users to describe system problems as domains and subdomains in actual business terminologies, using a ubiquitous language so that the problem domain in design can be understood by both business and IT stakeholders. This allows code and system design artifacts to properly align and verify.

DDD describes independent steps/areas of the problem domain as bounded contexts; a Bounded Context encapsulates the details of a single domain, such as domain model, data model, application services, etc., and it also defines the integration points with other bounded contexts. When done right, bounded context enables you to work without having to consider, or swap, between context. This concept synergistically matches the definition of a microservice— autonomous, well defined interfaces, implementing a business capability. This is what makes DDD an excellent tool in our architect's toolbox for identifying and designing microservices.

A thorough and detailed discussion of DDD and its various components are clearly outside the scope of this whitepaper, but I strongly encourage one to explore and understand this methodology at length. A great reference is the seminal book that started it all: Domain-Driven Design: Tackling Complexity in the Heart of Software 1st Edition.

What about PetShop?

Adopting a microservice architecture is facilitated when the domain is well understood, and PetShop is a canonical eCommerce website that easily qualifies it as such. Its essential business capabilities are as follows:
- Ordering
- Product Catalog and Inventory
- User Profile Management

As expected, these capabilities coincide nicely with how PetShop was originally designed. With this in mind, using the technique described above, we simply have to create an equivalent set of microservice implementations that are identical to the original methods, signature wise, and we will achieve our goal of breaking down the monolith!
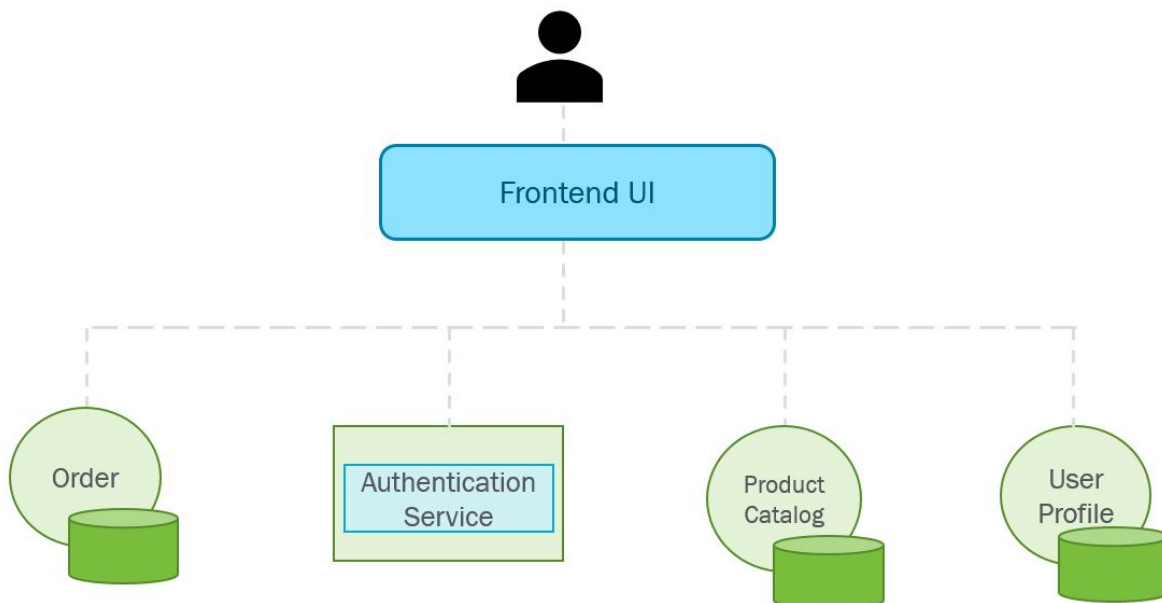
PetShop with Microservices Architecture

Conclusion

The microservices architecture grew out of developers hitting a wall. At some point, traditional monolithic application architectures simply are not able to scale anymore. Inevitably, this happens to every successful software project that is based on a monolithic architecture, no matter how well the application was originally architected or how much care and effort went into the maintaining a high-level of code quality. Either the database grew too large, or there are too many lines of code, or more likely these days, developers simply couldn't add features quickly enough. In some ways, microservices architecture embraces the failures of monoliths, addressing them using the true-and-proven technique of decomposition, with the focus on agility and replaceability rather than reusability. Moreover, unlike the monolithic architecture, this is a sustainable architecture as with it, technical debt is contained, and rapidly changing business requirements are met by adding new microservices, not modifying (and breaking) old ones.

What's next?

At this point, we are well on our way to transforming PetShop into a cloud-optimized system, but we still have a lot to do before we reach the final destination. Some of the areas for improvement are:

- Authentication
- Database
- Caching
- .NET Framework support

We'll cover these in depth in the second white paper of this series. Then, in the final installment, we will cover the topic of maximizing cloud's advantage through containerization and container orchestration using the technology that currently has the entire industry buzzing: Kubernetes!