IRON.IO WHITE PAPER: FEB 2015

MICROSERVICES

PATTERNS FOR BUILDING MODERN APPLICATIONS

INTRODUCTION

Being able to build, evolve, and scale large applications is critical for organizations, but the challenges involved make it a difficult task. Because of this, microservices has emerged as a dominant pattern for building modern cloud applications by breaking apart individual components as independent services that are centered around specific business capabilities.

While using microservices comes with a myriad of benefits in making large applications more manageable, building a reliable distributed system at scale is incredibly challenging in any scenario, as there are numerous considerations for dealing with failure, consistency, and performance, among others.

This white paper details the path to the microservices architecture and examines the benefits and drawbacks of the pattern. It also discusses best practices that will help developers and application architects achieve their application goals.

CONTENTS

THE EVOLUTION OF APP DEVELOPMENT 3

ENTER MICROSERVICES 5

BENEFITS OF A MICROSERVICES ARCHITECTURE 8

DRAWBACKS TO A MICROSERVICES ARCHITECTURE 9

BRINGING IT ALL TOGETHER 10

THE IRON.IO ADVANTAGE 14

THE MONOLITHIC ERA

In the beginning, there was the monolith: applications developed and deployed as a single entity. These monolithic applications are easy to deploy, since they only have one codebase and deployment configuration. They are well-suited for proof of concepts and MVP applications, but pose a number of challenges and constraints for production environment applications that require scalability.

Growing monoliths can easily become bloated in size and complexity, making it difficult to move quickly in development, testing, and deployment. A new developer joining the team needs to learn the inner workings of the entire application, regardless of role, and any minor change must run through a complete testing and deployment cycle before being updated. Most importantly, as a single entity, a monolith can only scale by replicating the entire application. This is costly to a business and a waste of resources as traffic and load grows.

EVOLUTION TO MULTI-TIER

The drawbacks of monolithic applications quickly became clear to developers. They began breaking apart their applications into logical distributed tiers that allowed for more efficient scalability. This multi-tier approach generally consists of a data layer, a business logic layer, and a presentation layer. Scaling an individual process due to increasing load would mean only needing to scale the business logic layer. Databases could be replicated independently, while the client layer could remain thin and cross-platform.

As applications built with this pattern grow, however, so does the strain on the business logic layer, leading to many of the drawbacks of the monolith. Again, as a single entity, scalability is challenging and expensive. This pattern did start the trend of decoupling components; however, it does not provide enough benefits in itself to serve modern applications.

© 2015 IRON.IO, INC.

SERVICE ORIENTED ARCHITECTURE (SOA)

Taking things one step further, developers began to envision their applications as a collection of business capabilities, thereby isolating components more around their purpose than their place in the stack. For example, developers would create a user service that handles authentication, an order service that handles billing, or a notification service that handles sending emails. Doing so provided more effective scalability, as each service is smaller and more focused.

While this pattern provided a framework for building effective application architectures, its practice has generally been ineffective due to unnecessarily complex abstractions and legacy protocols. Developers would attempt to use SOA to connect a wide range of applications that all spoke a different language, requiring an extra layer for an Enterprise Service Bus. This leads to archaic and costly configurations that cannot keep up as the technology and business landscape evolved.

ENTER MICROSERVICES

> "In computing, microservices is a software architecture design pattern, in which complex applications are composed of small, independent processes communicating with each other using language-agnostic APIs. These services are small, highly decoupled, and focus on doing a small task."
> -- Wikipedia

THE NEED FOR A NEW PATTERN

The evolution of the modern cloud drastically changed the way developers build and deploy applications. In its Top 10 Strategic Technology Trends for 2015, Gartner stated, "To deal with the rapidly changing demands of digital business and scale systems up — or down — rapidly, computing has to move away from static to dynamic models. Rules, models and code that can dynamically assemble and configure all of the elements needed from the network through the application are needed."

This shift in thinking around application architectures has also introduced a shift in practice. Further predictions from Gartner state that, "The first step toward the Web-scale IT future for many organizations should be DevOps — bringing development and operations together in a coordinated way to drive rapid, continuous incremental development of applications and services."

Using web-scale IT makes it easier for organizations to build applications and infrastructure similar to those offered by Amazon, Google, and Facebook. It puts them in a position to further embrace the cloud in an enterprise IT setting, delivering capabilities of large service providers to internal users.

HOW MICROSERVICES DIFFER FROM SOA

While decoupling application components is nothing new, the microservices pattern is clearer than SOA in its defining characteristics, providing a real-world framework

that satisfies modern application architecture requirements. Microservices is often referred to as "SOA done right;" however, that is not the only distinction.

Instead of connecting various applications together, the microservices pattern aims to create a single, cohesive application comprised of independently developed and deployed services that each follow the single responsibility principle. The term micro can be deceiving as to the characteristics of a microservice, however, since size is not the defining trait of microservices. While generally small, what is important is that each service is its own encapsulated process that can be developed and deployed independently. By limiting the scope of what a service can do, developers can ensure they do not unintentionally end up with a large number of decoupled monoliths.

In line with the modern cloud, communication between services is done over HTTP via RESTful APIs, passing JSON data, often through a message queue, to ensure reliability. The individual Microservices are generally processed asynchronously, triggered by an event such as an API call, push queue, schedule, or a webhook. A lightweight and efficient framework around communication and processing further distinguishes microservices from SOA.

THE NEW UNIT OF SCALE

From servers to virtual machines to containers, the goal has always been to minimize what needs to be replicated in a scalable environment. Along with the evolution of application development has been an equally important evolution in infrastructure resource provisioning. Both of these parallel evolutions have converged into the Microservices architecture, one of the primary reasons it has been welcomed with such regard.

Aside from the promise of a standard runtime across all environments, Docker has spread through the industry like wildfire by shrinking the unit of scale to a minimum. These days, every developer has a Docker story. By abstracting the host OS, containerized applications only need the code and dependent libraries to function, allowing for more granular scalability than with servers and virtual machines. As individual Microservices are meant to be limited to code and

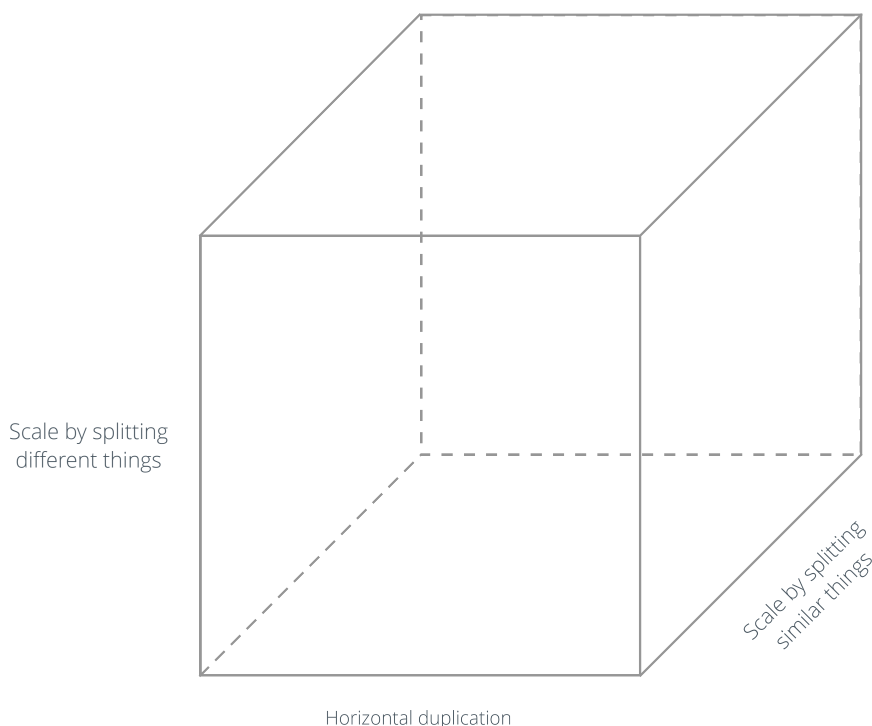
© 2015 IRON.IO, INC.

dependencies, it is only natural that the container would be the infrastructure unit to power the services within this pattern.

This new unit of scale will continue to drive infrastructure provisioning in the cloud, with IaaS vendors adopting containers and further competing over price. IDC predicts that "60% of SaaS applications will leverage new function-driven, micro-priced IaaS capabilities by 2018, adding innovation to a 'commodity' service." Docker may be a new technology, with the requisite concerns around orchestration and security, but an ecosystem has formed to solve many of the challenges in making Docker a production-ready technology.

THE SCALE CUBE

With a more granular unit of compute, scalability becomes more efficient and effective. As a pattern, microservices promotes Y-Axis scalability by decomposing functional elements as individual services, as opposed to traditional replication.



BENEFITS OF A MICROSERVICES ARCHITECTURE

This separation of components creates a more effective environment for building and maintaining highly scalable applications. Smaller services that are developed and deployed independently are easier to maintain, fix, and update, leading to more agile capabilities for responding to today's changing environments.

ELIMINATE SINGLE POINTS OF FAILURE

Separating components of an application makes it less likely that an individual bug or hardware failure will take down an entire system. Failed processes can be isolated, and down endpoints can be retried until reached.

STREAMLINED ORCHESTRATION

DevOps can be narrowed to automate each individual service with less complexity. Environments can remain consistent from development to staging to production, with less configuration management.

QUICKER ITERATIONS

Developers can be more focused on specific tasks and work in familiar languages. Pushing updates means they only need to run through the deployment process for that specific service, leading to more agile capabilities.

EFFECTIVE SCALABILITY

Scaling at the individual service level becomes more cost-effective and is elastic on-demand. Individual tasks can be processed concurrently without affecting the rest of the application.

VERSIONING

APIs can be versioned more effectively since individual services can follow their own scheme. Major releases can be done at the application level, while services can be updated on-demand.

LANGUAGE FLEXIBILITY

While not necessarily polyglot programming, each individual service can be in a different language based on a developer's preference, task suitability, or to match a certain library.

DRAWBACKS TO A MICROSERVICES ARCHITECTURE

Any application architecture that attempts to solve issues of scale does have a number of concerns, given the complex nature of distributed systems. Decoupling an application into independent services means that there are now more moving parts to maintain. This is intended as part of the style, but new factors must be taken into consideration.

COMPLEX ORCHESTRATION

While a key benefit of microservices is its streamlined orchestration capabilities, more services means maintaining more deployment flows. DevOps takes an even more important role with this pattern, as each service must be configured properly across its entire lifecycle.

INTER-SERVICE COMMUNICATION

Decoupled services need a reliable, effective way to communicate while not slowing down the whole application. Delivering data over the network introduces latency and potential failure, which can interfere with the user experience. A common approach is to introduce a message queue as a reliable transport layer.

DATA CONSISTENCY

As with any distributed architecture, ensuring consistency is a challenge, both for data at rest and data in motion. Multiple replicated databases and constant data delivery can easily lead to inconsistencies without the proper mechanisms in place.

MAINTAINING HIGH AVAILABILITY

Ensuring high availability is a requirement in any production system. Microservices provides more effective isolation and scalability; however, the uptime of each service contributes to the overall availability of applications. Each service must then have its own distributed measures implemented to ensure application wide availability.

TESTING

While keeping code and dependencies tight means a simpler development environment for specific services, it does introduce challenges with testing as it relates to the entire application. Services will often need to communicate with each other or rely on a data source or API. Testing one service independently would then require a complete test environment to be effective.

© 2015 IRON.IO, INC.

BRINGING IT ALL TOGETHER

One thing to always keep in mind when building out a microservices architecture is that the end result is a single application, both in how it functions and how it is perceived by end users. This means that, in the end, there must be a strong sense of unity in how it is maintained and delivered to preserve the intended user experience.

CONFIGURATION

Clear and effective configuration management in place for testing, continuous integration, deployment, service discovery, and more is critical due to the potential to grow to a very large number of services. The more automation, the better, provided the processes have been tested thoroughly and are monitored properly.

A large ecosystem of deployment and configuration tools exist in the marketplace, including Chef, Puppet, Ansible, CircleCI, Jenkins, and Consul, among others. These provide ways to build streamlined workflows. A well-configured microservices architecture allows developers to focus on their code and their code alone, maximizing efficiency and personal satisfaction.

API GATEWAY

Applications that follow the microservices pattern will not consist entirely of independent services, as there is still a need for a “home base” component that acts as both a request handler for responding to user events and a router for initiating individual service processes.

This API gateway forms the foundation of the entire architecture but would not be considered a service by itself. In fact, this component could be viewed as a monolith, although it is good practice to remain as light as possible to avoid the drawbacks.

API gateways need to be highly available, perform quickly and have the ability to scale on-demand based on traffic load. Modern development languages, like Go

and Node.js, provide an excellent framework for building highly effective API gateways that can quarterback the rest of the microservices architecture.

SERVICE DECOUPLING

Whether you are planning a new architecture from the ground up or refactoring an existing system, a key exercise is determining which components to break apart into independent services. The goal is always to make development, deployment, and scalability easier – not to add unnecessary complexity – thereby making the selection process very important.

Aside from aligning with the characteristics of a microservice, decoupled services generally should be event-driven tasks that can be processed asynchronously outside of the immediate user response loop. These could be quick background tasks, such as sending an email or placing an order; long running or memory-intensive tasks such as media encoding or big data processing; batch jobs like data cleansing or sending push notifications; or scheduled tasks such as daily digests or usage reports.

Once a service has been decoupled, it is essential that it both follows the characteristics of an independent microservice and remains properly integrated with the whole application.

STATE

To maintain an encapsulated environment, individual services themselves need to remain stateless. Services that must interact with a data source will do so via a secure connection and communication mechanism. Data sources may be isolated to a specific service or shared between services, depending on how the system is structured. For example, a user database could be shared across a registration service and an order service.

As database technologies continue to evolve, a common approach with microservices is to select a database type based on the nature of the service. Relational databases such as MySQL provide consistent data sets; NoSQL databases

such as MongoDB provide greater flexibility; and Key/Value databases such as Redis provide greater performance. This polyglot persistence pattern allows for more effective state management when building out large scale microservices applications.

PRESENTATION

Modern API-driven applications remove all state and business logic from clients to support multiple device targets and to avoid data inconsistency. Similar to the API gateway, the presentation layer is not a service in itself and should remain as light as possible to preserve a good user experience.

The rise of client-side frameworks such as AngularJS, ReactJS, Ember, and others supports this trend towards greater separation of concerns. Even established frameworks such as Ruby on Rails and Django are able to address presentation while supporting a service-oriented approach.

DEPLOYMENT

Software must be deployed before it can be used, but not every service will follow the same pattern. As each service is configured for its own life cycle, there are various ways to handle deployment. Components of the architecture that need to remain alive while handling requests, such as the API Gateway, databases, and the presentation layer, must be deployed as fault tolerant applications that persist indefinitely.

However, event-driven and asynchronous microservices do not need to be deployed as applications in the traditional sense. Modern cloud technologies such as Docker provide ephemeral compute resources that are triggered by an application event. This leads to highly efficient and cost-effective architectures, as no infrastructure resources are required.

The next wave in cloud technologies is becoming more workload-aware. Infrastructure (IaaS) and Platform as a Service (PaaS) providers such as AWS, Azure, Heroku, OpenShift, and Pivotal Web Services offer more automated elastic

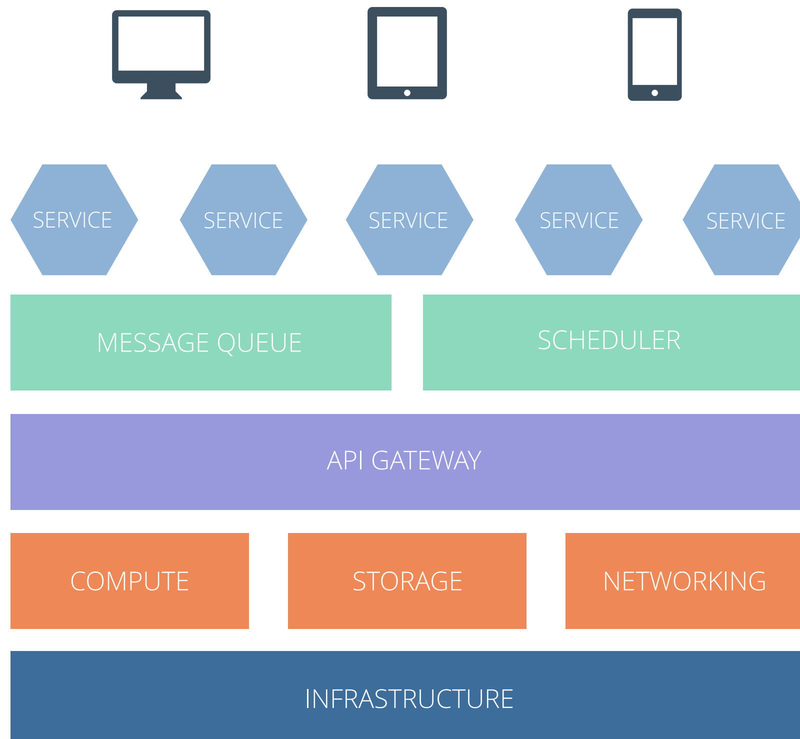
scalability in response to traffic and events, providing a viable framework for deploying large scale applications.

MONITORING & ANALYTICS

As each microservice is a single process, it is important to not only understand how it performs independently but also how it fits within the whole application. While isolation helps avoid single points of failure, an errant service can still slow down an application. Without unified monitoring, it may be difficult to hone in on the cause.

Services such as NewRelic, AppDynamics, Librato, PaperTrail, DataDog, Airbrake, and PagerDuty provide valuable insight into all the workings of the application and systems, but it does take some configuration to ensure each service is tracked properly as part of the whole application. A good practice is to configure tools to treat the individual services as if they were processes, essentially treating a decoupled microservices architecture as if it were a monolith when it comes to monitoring and analytics.

THE MICROSERVICES STACK



© 2015 IRON.IO, INC.

THE IRON.IO ADVANTAGE

As building modern applications forces modern thinking, business adopting the microservices pattern will require a new generation of services that fit their needs. According to 451 Research, "IaaS [Infrastructure as a Service] is not enough. To crack the next wave of cloud buyer opportunity, suppliers will need to raise their IQs beyond technology and provide services packaged with IaaS that can support specific enterprise workload needs (together with appropriate certification, regulatory compliance, etc.), whether as services or hosted business processes."

Iron.io is a cloud-native infrastructure services provider that offers a complete platform for building and maintaining highly scalable microservices architectures, the IaaS+ that rises to the challenges faced by today's developers and application architects.

Iron.io has been operating at scale since 2011, with over 20,000 developer accounts serving billions of messages, and millions of process hours every month.

Reliable message queue solution that lets you connect systems and decouple components

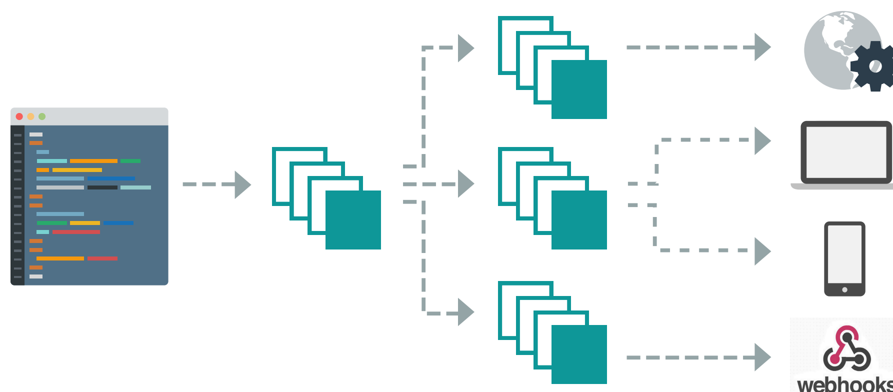
Multi-language worker platform that runs parallel tasks in the background at massive scale.

© 2015 IRON.IO, INC.

INTER-SERVICE COMMUNICATION WITH IRONMQ

In line with the microservices pattern of communication, IronMQ is a cloud-native message queue that delivers JSON data over HTTP via a RESTful API. This means no additional abstraction is required to incorporate a message queue for inter-service and third-party API communication.

One of the defining characteristics of IronMQ is its ability to persist data in transit without a major performance loss. Messages are stored in a high performance key/value data store and only removed when acknowledged by the consumer.



KEY FEATURES & BENEFITS

Highly Available
Runs on top of cloud infrastructures and uses multiple high-availability data centers. Scales without the need to add and maintain resources yourself.

Cloud-Native Technologies
Uses HTTP/REST-based APIs for simple and efficient cloud use. Built with modern MQ standards for maximum flexibility and configuration.

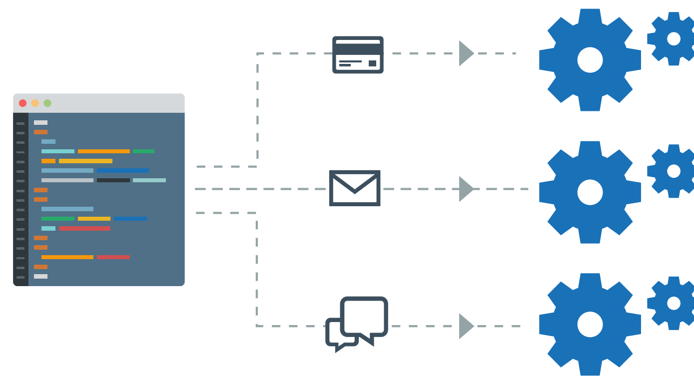
Reliable Data Persistence
Uses reliable data stores for message durability and persistence. Messages are delivered once and only once, in the order that they are received.

Advanced Feature Set
Supports a rich set of features, including Push and Pull Queues, Long Polling, Error Queues, Alerts and Triggers, an advanced Dashboard, and more.

MICROSERVICE PROCESSING WITH IRONWORKER

Microservices' independent, stateless processes encapsulated with only the required dependencies mean that their hosting and processing capabilities fit neatly with the pattern of IronWorker, a highly scalable Dockerized runtime environment.

Services can be developed in any language, packaged, uploaded, and processed in a managed cloud environment on-demand. Microservices are event-driven in nature, and IronWorker tasks can be initiated via an API call, a push queue, a webhook, or a schedule. The task is processed inside of a Docker container with only the required language and library dependencies for efficient resource allocation.



KEY FEATURES & BENEFITS

Containerized Environment
Write workers as if to run in the app but then simply upload and run them in IronWorker. Works with Ruby, Python, PHP, Java, .NET, Node, Go, and more.

High-Scale Processing
Meant for on-demand elastic parallel processing without the need to provision and manage infrastructure resources on your own.

Flexible Scheduling
Schedule jobs to run in the future using flexible options. Run once, a set number of times, on a recurring schedule, and many options in between.

Reliable & Secure
Uses SSL connections and runs each task in an isolated Docker sandbox. Uses OAuth2 to provide simplicity, flexibility, scalability, and security.

CONCLUSION

Microservices is more than a fad. It is a lasting pattern that takes full advantage of modern cloud technologies while addressing a number of challenges that occur when scaling an application. While a relatively new practice, large companies such as Netflix are proving that the pattern is ready for a production environment. A growing ecosystem of IaaS, PaaS, and SaaS companies provide the necessary components for building and maintaining large scale applications that follow this pattern. As businesses look to invest further in the cloud, the microservices pattern will be a common architecture that continues to attract more attention.

ABOUT THE AUTHOR

Ivan Dwyer is the Director of Channels and Integrations at Iron.io, working with various partners across the entire cloud technology and developer services ecosystem to form strategic alliances around real world business solutions.

READY TO START BUILDING MICROSERVICES APPS?

Visit www.iron.io to sign up for a free 30 day trial

© 2015 IRON.IO, INC.