BROUGHT TO YOU IN PARTNERSHIP WITH redhat.

GETTING STARTED WITH Microservices

UPDATED BY VINEET REYNOLDS

INTRODUCTION

The term "microservices" describes a new software development style that has grown from recent trends in software development/management practices—practices meant to increase the speed and efficiency of developing and managing software solutions at scale.

Agile methods, DevOps culture, cloud, Linux containers, and the widespread adoption (both culturally and technically) of CI/CD methods across the industry are making it possible to build truly modular large-scale service-optimized systems for both internal and commercial use.

This Refcard aims to introduce the reader to microservices, define their key characteristics, and provide solutions to common problems involved in transitioning to microservices, as succinctly as possible.

WHAT ARE MICROSERVICES?

Microservices involve an architectural approach that emphasizes the decomposition of applications into single-purpose, highly cohesive and loosely coupled services managed by cross-functional teams, for delivering and maintaining complex software systems with the velocity and quality required by today's digital businesses.

Microservices are language-, platform-, and operating system-agnostic. They can be used to break down a big monolithic application (typically packaged as a single archive that is vertically scaled up or horizontally scaled out) into smaller and simpler services. Each microservice does one thing, and does it well, so the "micro" in microservices refers to the scope of the services' functionalities, not the number of Lines of Code (LOC).

Each microservice is generally built by a full-stack team, which reduces potential communication difficulties between disparate teams that could otherwise be problematic.

Microservices may not be suitable for simpler applications and are better suited for complex applications that have grown over a period of time, with large codebases and teams. Microservices are ideal in realizing multiple business functions, with each microservice focusing on realizing a single business function or sub-function. Teams are free to innovate at their own pace.

The frequency with which an application needs to be updated, and the quick turnaround that is possible using a microservices design, are a few key factors driving this style of architecture.

The concept behind microservices is similar to Service-Oriented Architecture (SOA), which is why this style of architecture has been referred to as "SOA with DevOps," "SOA for hipsters," and "SOA 2.0".

KEY CHARACTERISTICS OF MICROSERVICES

1. **Domain-Driven Design.** Functional decomposition can be easily achieved using Eric Evans's DDD approach.
2. **Single Responsibility Principle.** Each service is responsible for a single part of the functionality, and does it well.
3. **Explicitly Published Interface.** A producer service publishes an interface that is used by a consumer service.
4. **Independent DURS (Deploy, Update, Replace, Scale).** Each service can be independently deployed, updated, replaced, and scaled.
5. **Smart Endpoints and Dumb Pipes.** Each microservice owns its domain logic and communicates with others through simple protocols such as REST over HTTP.

BENEFITS OF MICROSERVICES

INDEPENDENT SCALING

Each microservice can scale independently via X-axis scaling (cloning with more CPU or memory) and Z-axis scaling (sharding), based on what is needed. This is very different from monolithic applications, which may have very diverse requirements but must still be deployed together as a single unit.

INDEPENDENT UPGRADES

Each service can be deployed independently of other services. Any change local to a service can be easily

Make
MICROSERVICES
WORK for **YOU.**

Want to increase speed and efficiency of development? Microservices can help and combined with the strength of a DevOps culture you can move even faster. Get access to content, solutions and more with Red Hat Developers.

Learn more. Code more. Share more.
developers.redhat.com

RED HAT DEVELOPERS

redhat

made by a developer without requiring coordination with other teams. For example, to meet ever changing business requirements, a service can be improved by replacing the underlying implementation. As a result this improves the agility of the microservice. This is also a great enabler of CI/CD.

EASY MAINTENANCE

Code in a microservice is restricted to one capability and is thus easier to understand. IDEs can load the smaller amounts of code more easily, and smaller codebases increase velocity by enabling awareness of the side effects of the code developers are currently writing.

POTENTIAL HETEROGENEITY AND POLYGLOTISM

Developers are free to pick the language and stack that are best suited for their service. They are free to innovate within the confines of their service. This enables one to rewrite the service using newer technologies as opposed to being penalized because of past decisions, and gives freedom of choice when picking a technology, tool, or framework.

FAULT AND RESOURCE ISOLATION

A misbehaving service, such as a memory leak or an unclosed database connection, will only affect that service, as opposed to an entire monolithic application. This improves fault isolation and limits how much of an application a failure can affect. With well-designed microservices, faults can be isolated to a single service and not propagate to the rest of the system, allowing a certain degree of anti-fragility to manifest.

IMPROVED COMMUNICATION ACROSS TEAMS

A microservice is typically built by a full-stack team. All members related to a domain work together in a single team, which significantly improves communication between team members, as they share the same end goals, deliver cadence and perhaps most importantly their service is their product and they are wholly responsible for it even in the production environment.

OPERATIONAL REQUIREMENTS FOR MICROSERVICES

Microservices are not the silver bullet that will solve all architectural problems in your applications. Implementing microservices may help, but that is often just the byproduct of refactoring your application and typically rewriting code using guidelines required by this architecture style. True success requires significant investment.

SERVICE REPLICATION

Each service needs to replicate, typically using X-axis cloning or Y-axis partitioning. There should be a standard mechanism by which services can easily scale based upon metadata. A container-optimized application platform such as OpenShift by Red Hat, can simplify this functionality.

SERVICE DISCOVERY

In a microservice world, multiple services are typically distributed in a container application platform. Immutable infrastructure is provided by containers or immutable VM images. Services may scale up and down based upon certain pre-defined metrics. The exact address of a service may not be known until the service is deployed and ready to be used.

The dynamic nature of a service's endpoint address is handled by service registration and discovery. Each service registers with a broker and provides more details about itself (including the endpoint address). Other consumer services then query the broker to find out the location of a service and invoke it. There are several ways to register and query services such as ZooKeeper, etcd, consul, Kubernetes, Netflix Eureka, and others. Kubernetes, in particular, makes service discoverability very easy as it assigns a virtual IP address to groups of like resources and manages the mapping of DNS entries to those grouped resources.

SERVICE MONITORING

One of the most important aspects of a distributed system is service monitoring and logging. This enables one to take proactive action if, for example, a service is consuming unexpected resources. Elasticsearch, Fluentd, and Kibana can aggregate logs from different microservices, provide a consistent visualization over them, and make that data available to business users.

RESILIENCY

Software failure will occur, no matter how much and how hard you test. This is all the more important when multiple microservices are distributed all over the Internet. The key concern is not "how to avoid failure" but "how to deal with failure." It's important for services to automatically take corrective action to ensure user experience is not impacted. The Circuit Breaker pattern allows one to build resiliency in software—Netflix's Hystrix is a good library that implements this pattern.

DEVOPS

Continuous Integration and Continuous Deployment (CI/CD) are very important in order for microservices-based applications to succeed. These practices are required so that errors are identified early via well automated testing and deployment pipelines.

GOOD DESIGN PRINCIPLES FOR EXISTING MONOLITHS

Refactoring a monolith into a microservices-based application will not help solve all architectural issues. Before you start breaking up a monolith, it's important to make sure the monolith is designed following good software architecture principles. Some common rules are:

1. Separate your concerns. Consider using Model-View-Controller (MVC) for front-end abstraction. Use well-defined APIs for high cohesion and low coupling. Separate interfaces/APIs and implementations.

2. Use Convention over Configuration (CoC). Keep configuration to a minimum by using or establishing conventions.

3. Follow the Law of Demeter. Each unit should have only limited knowledge about other units: only units

"closely" related to the current unit. Each unit should only talk to its friends; don't talk to strangers. Only talk to your immediate friends.

4. Use Domain-Driven Design. Keep objects related to a domain/component together.
5. Focus on Automation. Automate testing against production-like deployments in development environments and crafting a fully automated deployment pipeline will provide value even if you stick with your existing monolith.
6. Design to Interfaces / APIs. Classes shouldn't call other classes directly just because they happen to be in the same archive. Highly cohesive classes may be extracted later into their own microservices, and clean interfaces allow for their clients to discover and use their functionality even when they move to microservices.
7. Group code by functionality, not by layer. Monoliths usually have a layered architecture, and it's common to group classes by layer. Instead you should break up your code into packages separated by functionalities to allow easier splitting into future microservices. It also makes easier to see the dependencies between your future microservices.
8. Make your code stateless and externalize your application's state. One cannot rely on static variables or global variables as the source of your application's state. Use external data sources such as key-values stores or databases to maintain state.

REFACTORING A MONOLITH TO MICROSERVICES

Consider a Java EE monolithic application that is typically defined as a WAR or an EAR archive. The entire functionality for the application is packaged in a single unit. For example, an online shopping cart may consist of User, Catalog, and Order functionalities. All web pages are in the root of the application, all corresponding Java classes are in the WEB-INF/classes directory, and all resources are in the WEB-INF/classes/META-INF directory.

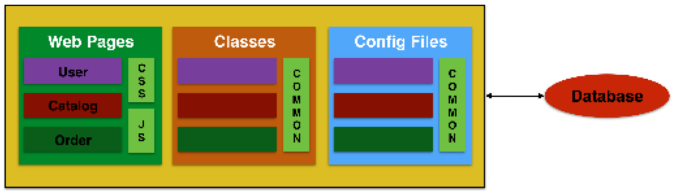

Figure 1: Monolith Architecture

Such an application can be refactored into microservices, which would create an architecture that would look like the following:

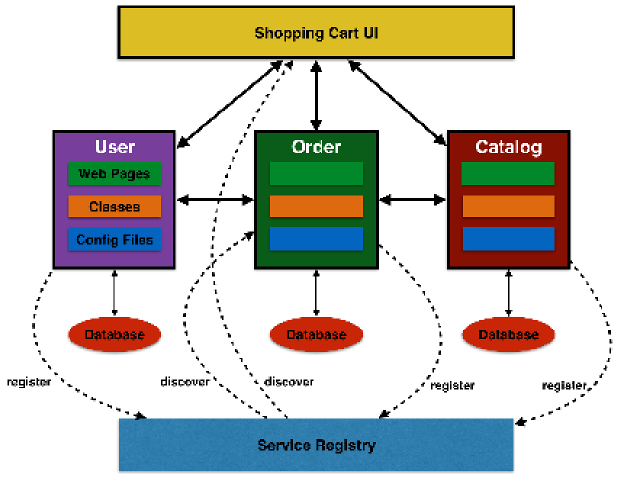

Figure 2: Refactoring to Microservices

The above application is functionally decomposed where User, Order, and Catalog components are packaged as separate WAR files. Each WAR file has the relevant web pages, classes, and configuration files required for that component.

Java EE is used to implement each component, but there is no long term commitment to the stack, as different components talk to each other using a well-defined API.

Different classes in this component belong to the same domain, so the code is easier to write and maintain. The underlying stack can also change, possibly keeping technical debt to a minimum.

Each archive has its own database (i.e. data stores are not shared). This allows each microservice to evolve and choose whatever type of data store—relational, NoSQL, flat file, in-memory, or something else—is most appropriate.

Each component registers with a Service Registry. This is required because multiple stateless instances of each service might be running at a given time, and their exact endpoint locations will be known only at the runtime. Kubernetes, Netflix Eureka, etcd, and Zookeeper are some options to implement a service registry/discovery.

If components need to talk to each other, which is quite common, then they would do so using a predefined API. REST for synchronous or Pub/Sub for asynchronous communication are the most common means to achieve this. In this case, the Order component discovers User and Catalog service and talks to them using a REST API.

Client interaction for the application is defined in another application (in this case, the Shopping Cart UI). This application discovers the services from the Service Registry and composes them together. It should mostly be a dumb proxy (discussed in a later section), where the UI pages of the different components are invoked to display the interface. A common look and feel can be achieved by providing standard CSS/JavaScript resources

PATTERNS IN MICROSERVICES

1. USE BOUNDED CONTEXTS TO IDENTIFY CANDIDATES FOR MICROSERVICES

Bounded contexts (a pattern from domain-driven design) are a great way to identify what parts of the domain model of a monolithic application can be decomposed into microservices. In an ideal decomposition, every bounded context can be extracted out as a separate microservice, which communicates with other similarly-extracted microservices through well defined APIs. It is not necessary to share the model classes of a microservice with consumers; the model objectvs should be hidden as you wish to minimize binary dependencies between your microservices.

The use of an anti-corruption layer is strongly recommended during the process of decomposing a monolith. The anti-corruption layer allows the model in one microservice to change without impacting the consumers of the microservice.

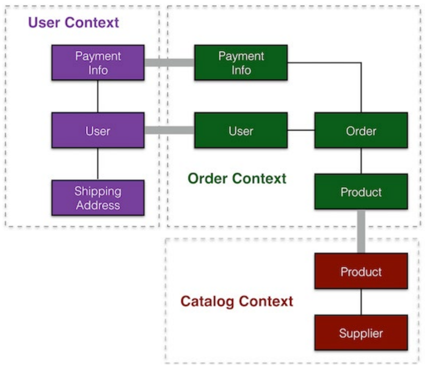

Figure 3: Bounded contexts for a shopping cart application

2. DESIGNING FRONTENDS FOR MICROSERVICES

In a typical monolith, one team is responsible for developing and maintaining the frontend. With a microservices-architecture, multiple teams would be involved, and this could easily grow into an unmanageable problem.

This could be resolved by componentizing various part of the frontend so that each microservice team can develop in a relatively isolated manner. Each microservice team will develop and maintain it's own set of components. Changes to various parts of the frontend are encapsulated to components and thus do not leak into other parts of the frontend.

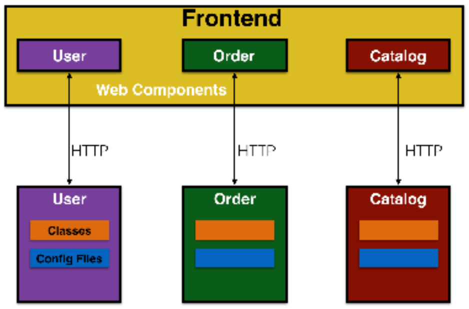

Figure 4: Web Components for various front-end modules

3. USE AN API GATEWAY TO CENTRALIZE ACCESS TO MICROSERVICES

The frontend, acting as a client of microservices, may fetch data from several different microservices. Some of these microservices may not be capable of responding to protocols native to the web (HTTP+JSON/XML), thus requiring translation of messages from one protocol into another. Concerns like authentication and authorization, request-response translation among others can be handled in a centralized manner in a facade.

Consider using an API Gateway, which acts as a facade that centralizes the aforementioned concerns at the network perimeter. Obviously, the API Gateway would respond to client requests over a protocol native to the web, while communicating with underlying microservices using the approach preferred by the microservices. Clients of microservices may identify themselves to the API Gateway through a token-authentication scheme like OAuth. The token may be revalidated again in downstream microservices to enforce defense in depth.

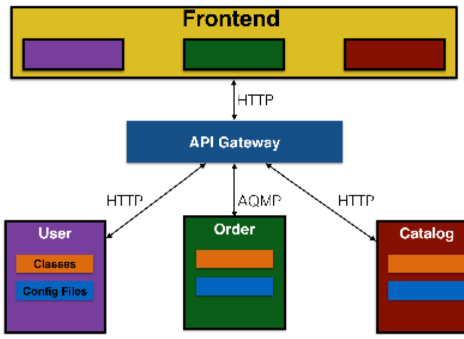

Figure 5: Use an API gateway as an intermediary

4. DATABASE DESIGN AND REFACTORINGS

Unlike a monolithic application, which can be designed to use a single database, microservices should be designed to use a separate logical database for each microservice. Sharing databases is discouraged and is an anti-pattern, as it forces one or more teams to wait for others, in the event of database schema updates; the database essentially becomes an API between the teams. One can use database schemas or other logical schemes to create a separate namespace of database objects, to avoid the need to create multiple physical databases.

Additionally, one should consider the use of a database migration tool like Liquibase or Flyway when migrating from a monolithic architecture to a microservices architecture. The single monolithic database would have to be cloned to every microservice, and migrations have to be applied on every database to transform and extract the data appropriate to the microservice owning the database. Not every object in the original database of the monolith would be of interest to the individual microservices.

5. PACKAGING AND DEPLOYING SERVICES

Each microservice can in theory utilize its own technology stack, to enable the team to develop and maintain it in isolation with few external dependencies. It is quite possible for a single organization to utilize multiple runtime platforms (Java/JVM, Node, .NET, etc.) in a microservices architecture. This potentially opens up problems with provisioning these various runtime platforms on the datacenter hosts/nodes.

Consider the use of virtualization technologies that allow the tech stacks to be packaged and deployed in separate virtual environments. To take this further, consider the use of containers as a lightweight-virtualization technology that allows the microservice and its runtime environment to be packaged up in a single image. With containers, one forgoes the need for packaging and running a guest OS on the host OS. Following which, consider the use of a container orchestration technology like Kubernetes, which allows you to define the various deployments in a manifest file, or a container platform, such as OpenShift. The manifest defines the intended state of the datacenter with various containers and associated policies, enabling a DevOps culture to spread and grow across development and operations teams.

6. EVENT-DRIVEN ARCHITECTURE

Microservices architectures are renowned for being eventually consistent, given that there are multiple datastores that store state within the architecture. Individual microservices can themselves be strongly consistent, but the system as a whole may exhibit eventual consistency in parts. To account for the eventual consistency property of a microservices architecture, one should consider the use of an event-driven architecture where data-changes in one microservice are relayed to interested microservices through events.

A pub-sub messaging architecture may be employed to realize the event-driven architecture. One microservice may publish events as they occur in its context, and the events would be communicated to all interested microservices, that can proceed to update their own state. Events are a means to transfer state from one service to another.

7. CONSUMER-DRIVEN CONTRACT TESTING

One may start off using integration testing to verify the correctness of changes performed during development. This is, however, an expensive process, since it requires booting up a microservice, the microservice's dependencies, and all the microservice's consumers, to verify that changes in the microservice have not broken the expectations of the consumers.

This is where consumer-driven contracts aid in bringing in agility. With a consumer-driven contract, a consumer will declare a contract on what it expects from a provider.

CONCLUSION

The microservices architectural style has well known advantages and can certainly help your business evolve faster. But monoliths have served us well and will continue to work for years to come.

Consider the operational requirements of microservices in addition to the benefits before refactoring your monolith to a microservices architecture. Many times, better software engineering, organizational culture, and architecture will be enough to make your monoliths more agile. But, if you decide to follow the microservice route, then the advice in this Refcard should help to get you started.

ABOUT THE AUTHOR

VINEET REYNOLDS is a Senior Software Engineer at Red Hat, focusing on developer experience. He works primarily on writing example applications to demonstrate Red Hat's software offerings. He currently works on an application that uses a microservices-oriented architecture to validate the entire development experience starting with conception of a business idea, development, and refactoring, all the way through deployment and operations.

DZONE, INC.
150 PRESTON EXECUTIVE DR.
CARY, NC 27513
888.678.0399
919.678.0300

REFCARDZ FEEDBACK WELCOME
refcardz@dzone.com

SPONSORSHIP OPPORTUNITIES
sales@dzone.com

ISBN-13: 978-1-936502-77-6
ISBN-10: 1-936502-77-1
50795
9 781936 502776

BROUGHT TO YOU IN PARTNERSHIP WITH
redhat