클라우드 컴퓨팅에서 Java 사용

Red Hat을 통해 Java 워크로드를 클라우드 환경으로 이전

25년이 넘는 시간 동안 Java™는 매우 인기 있는 프로그래밍 언어였고, 오늘날도 계속해서 엔터프라이즈 소프트웨어 개발 분야에서 가장 많이 사용되는 3대 언어 중 하나입니다. 전 세계에서 수많은 비즈니스 크리티컬 시스템이 이 상징적인 언어를 사용해 구축되었습니다.

1999년 출시된 Java 2 Platform, Enterprise Edition(J2EE™) 1.2는 엔터프라이즈 Java의 탄생을 세상에 알렸고 소프트웨어 개발의 미래를 형성하는 데 일조했습니다. 메이저 버전에는 Java 1.3, Java 1.5, Java 8, Java 11이 포함됩니다. 수년 동안 많은 사람들이 "Java는 죽었다"고 선언했지만 최근 Java 18 출시와 함께 여전히 강세를 유지하고 있습니다.

Java는 객체 지향적이고, 플랫폼에 독립적이고, 안정적이며, 보안이 유지되고, 역호환성을 지니고 있으므로 항상 엔터프라이즈 환경에 언제나 적합합니다. 게다가 Java는 처음부터 잘 정의된 일련의 API와 광범위한 개발자 툴링을 제공해 왔습니다. 또한 Java는 배우기 쉬워 많은 사랑을 받고 있으며, 어디서나 실행할 수 있다는 장점 덕분에 수십 억대가 넘는 수의 기기(예: 서버, 컴퓨터, 스마트폰, 게임 콘솔)에 Java가 도입되었습니다. 대체로 JVM은 속도가 빠르고, 안정적이며, 전 세계 기업의 요구 사항을 충족하는 성숙한 언어입니다.

하지만 지난 10년 동안 개발자들이 종래의 모놀리식 아키텍처 패턴에서 벗어나 클라우드 컴퓨팅과 쿠버네티스를 사용해 구축된 더 가벼운 모듈식 서비스와 기능으로 옮겨가면서 애플리케이션 개발은 중대한 변화를 겪었습니다. 이 모든 변화는 더 효율적인 분산형 클라우드 환경을 활용하기 위한 움직임이었습니다. 이로 인해 원래 빠른 부팅 시간, 적은 메모리 공간 또는 컨테이너를 염두에 두고 설계된 것이 아닌 Java의 입지는 어떤 변화를 겪게 될까요? Java는 진화해야 했고 개발자가 클라우드 환경에서 Java의 장점을 누리고자 한다면 Java 외에 다른 것을 찾아서는 안 됩니다. Java가 클라우드 컴퓨팅, 그리고 무엇보다도 엔터프라이즈 애플리케이션 풋프린트를 위한 효과적인 선택지로 자리 잡는 데 기여한 몇 가지 설계 가정과 변경 사항을 살펴보겠습니다.

설계 가정

Java는 클라우드 환경에서 바로 사용할 수 있을까요? 원래 Java는 클라우드 컴퓨팅 및 쿠버네티스와 호환되지 않도록 하는 특정한 설계 가정을 기반으로 구축되었습니다. 지난 10년 동안 애플리케이션 개발이 진화해온 양상을 감안하면 이러한 설계 가정은 더 이상 유효하지 않습니다.

이러한 설계 가정 중 몇 가지는 다음과 같습니다.

장기 실행 애플리케이션

Java는 애플리케이션이 어디서든 작동하도록 설계되었으며 오랜 시간 동안 풋프린트를 희생하며 처리량을 극대화했습니다. 이로 인해 Java는 한 대의 서버에서 다수의 애플리케이션을 호스팅하는, 최대 메모리와 CPU를 보유한 대규모 데이터센터에 이상적인 언어로 자리 잡았습니다. 수직 스택의 특성으로 인해 Java는 비용과 성능 측면을 모두 충족하는 최적의 솔루션으로 각광받았습니다. 당시 이러한 특정 개발 스타일은 수직 스택에 효과적이었지만 Java 애플리케이션을 더 분산되고 본질적으로 탄력적인

스케일링 기능을 제공하는 클라우드 환경으로 이전하는 경우 당연히 문제가 발생하게 됩니다. 따라서 대규모 모놀리스로 구축된 애플리케이션은 기본 스택에 더 많은 리소스를 필요로 하게 되고, 릴리스 주기가 길어지면서 기술 부채가 축적됨에 따라 기반 스택을 더 '고수'하게 되었습니다.

Java는 장기 실행 애플리케이션에 매우 효과적인 언어였습니다. 장기 실행 애플리케이션이 규모에 맞게 성능을 발휘할 수 있도록 지원하는 두 가지 핵심적 특징을 살펴보겠습니다.

가비지 컬렉션(Garbage Collection)

가비지 컬렉션(GC)은 JVM의 자동화된 메모리 관리 기능으로서, 과거에는 장시간 실행되고 종료되지 않는 애플리케이션에 필요한 기능이었습니다. GC는 더 이상 사용되지 않는 메모리 내 객체를 주기적으로 제거함으로써 필요한 다른 곳에 사용할 수 있도록 메모리를 회수합니다.

이 기능은 중요한 목적을 달성하는 데 기여하지만 막대한 CPU 부하가 요구되기 때문에 GC 프로세스 중에 애플리케이션이 느려지거나 일시 정지하거나 심지어 중단될 수 있다는 단점도 있습니다. 미션 크리티컬 애플리케이션에서는 이러한 대기 시간 및 가용성 관련 문제로 인해 비즈니스 트랜잭션이 중단되거나 사용자 경험이 저하될 수 있습니다.

GC는 당시 필수적인 기능이었지만 이제 더 이상 유효하지 않습니다. 클라우드의 애플리케이션은 대부분 규모가 작고, 신속하게 비활성화 및 활성화되고 확장 및 축소되도록 설계되어 있기 때문입니다. 클라우드의 애플리케이션은 일반적으로 밀리초 동안 실행된 후 종료되므로 가비지 컬렉션이 필요할 만큼 오래 실행되지 않습니다.

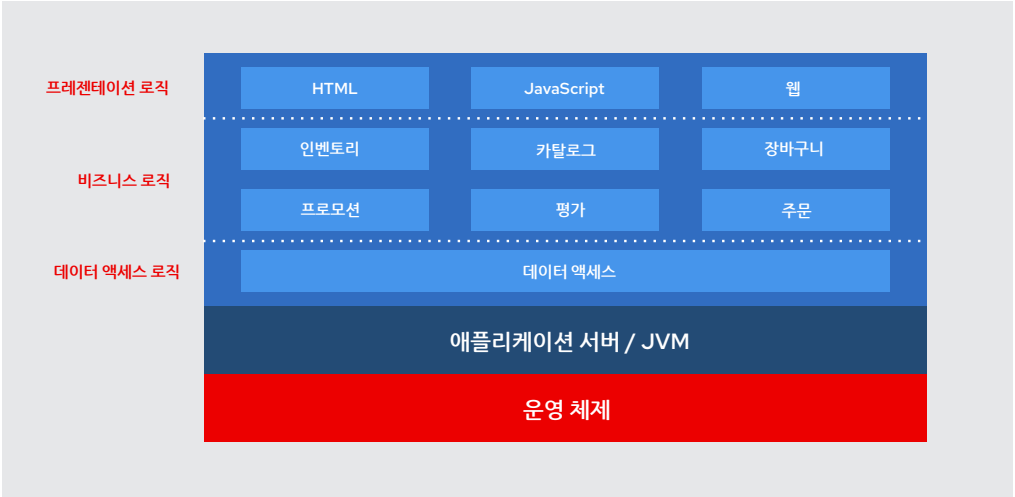
JIT(Just-In-Time) 컴파일러

과거에 장기 실행 애플리케이션에 중요했던 Java 기능의 또 다른 예가 바로 JIT 컴파일러입니다. 시간이 지나면서 JVM은 애플리케이션의 액세스 패턴을 학습하고 요청 처리 방식을 최적화하여 불필요한 코드를 제거하고 자주 사용되는 코드의 성능을 극대화할 수 있습니다.

GC와 유사하게 JIT 컴파일러는 애플리케이션과 동일한 프로세스에서 실행되고 리소스를 차지하기 위해 애플리케이션과 경쟁한다는 단점이 있습니다. 또한 GC와 마찬가지로 JIT 컴파일러는 애플리케이션이 밀리초 동안만 실행되기 때문에 클라우드 환경에서는 필요하지 않습니다.

수직 스택

하나의 머신에서 여러 개의 애플리케이션 실행



대규모 분산형 아키텍처에 적합하지 않은 또 하나의 설계 가정은 바로 애플리케이션 서버입니다. 애플리케이션 서버는 사용자가 기본 하드웨어의 가치를 극대화할 수 있도록 설계되었습니다. Java는 한 대의 서버에서 여러 애플리케이션에 고성능을 제공할 수 있으므로 조직은 매우 비싼 하나의 대규모 서버에 투자하여 모든 애플리케이션을 호스팅할 수 있습니다.

이론상 애플리케이션 서버는 타당했지만, 단점은 동일한 애플리케이션 서버를 공유하는 애플리케이션을 지속적으로 제공하거나 다른 애플리케이션에 영향을 주거나 전체 시스템을 중단시킬 수 있는 기술적 문제(예: 메모리 유출)에 부딪힐 만큼 충분히 격리할 수 없다는 것입니다. 이 문제의 해결 방법(제2의 해결책으로 분류하는 것이 더 적절할 수 있음)으로 사용자는 애플리케이션 서버의 원래 목적을 완전히 무시하고 서버당 한 개의 애플리케이션을 배포할 수 있습니다.

클라우드 환경에서 애플리케이션 서버라는 개념은 클라우드 컴퓨팅의 내재적 속성인 '분산'으로 인해 그 입지가 흔들리게 됩니다. 애플리케이션은 여러 머신 전반에 분산된 클라우드 플랫폼에서 호스팅됩니다. 이것이 가능한 이유는 애플리케이션이 클라우드 환경에 이상적인 경량화, 모듈식, 이식성의 특성을 유지하도록 설계되었기 때문입니다. 이 시나리오에서는 대규모의 온프레미스 하드웨어가 필요 없습니다. 애플리케이션 서버와 같은 값비싼 온프레미스 하드웨어가 필요 없다는 것은 클라우드 환경의 가장 매력적인 장점 중 하나입니다.

처리량을 고려한 설계

기존 Java 개발에서는 애플리케이션이 수직 스택에서 설계되었습니다. 따라서 관련 구성 요소가 모두 스택에 포함되어 있으며 모두 단일 모놀리식 애플리케이션의 일부입니다. Java는 최대 처리량과 짧은 대기 시간을 실현하도록 설계되었으며, 이를 위해서는 수직 스택에 상당히 많은 리소스가 필요합니다. 하지만 이러한 제한은 클라우드 환경의 애플리케이션에는 더 이상 적용되지 않습니다. 애플리케이션을 수평적으로 확장할 수 있기 때문입니다. 즉, 애플리케이션이 여러 머신뿐 아니라 여러 클라우드 환경 전반에도 분산된다는 뜻입니다. 이러한 방식으로 애플리케이션은 처리량과 확장성을 달성할 수 있습니다.

개발자들이 클라우드 컴퓨팅으로 이동하는 이유는 무엇일까요?

클라우드 컴퓨팅은 여러 가지 장점을 제공합니다. 몇 가지 예는 다음과 같습니다.

간소화된 개발자 경험

개발자에게 매우 매력적인 클라우드 컴퓨팅의 한 측면은 사용이 간편해 개발자 경험을 개선할 수 있다는 것입니다. 클라우드 환경에서 개발자는 버튼만 누르면 몇 분 내로 리소스에 액세스할 수 있습니다. 게다가 클라우드 환경의 툴은 잘 통합되어 있으며 개발자는 네트워크, 스토리지, 컴퓨팅 또는 기타 서비스를 구성하지 않아도 됩니다. 이는 모두 사전 구성되어 있으며 클라우드 환경에서 즉시 사용할 수 있습니다.

고가용성

클라우드 제공업체는 과도한 지출과 상당한 노력 없이도 데이터센터에서 독보적인 서비스 수준 계약(SLA)인 99% 이상의 가동 시간을 보장합니다. 따라서 클라우드 서비스는 실행하고자 하는 클라우드 애플리케이션 유형에 관계없이 개발자에게 매력적인 옵션이 되고 있습니다.

확장성

확장성은 개발자가 클라우드 컴퓨팅으로 옮아가는 가장 중요한 이유 중 하나입니다. 클라우드 컴퓨팅을 통해 사용자는 요구 사항 변화에 맞춰 온디맨드 방식으로 리소스를 늘리거나 줄일 수 있기 때문입니다. 클라우드가 아닌 환경에서 동일한 수준의 확장성을 실현하려면 엄청난 비용이 듭니다.

글로벌 서비스

클라우드 환경은 전 세계에 대기 시간이 짧은 서비스를 제공하여 글로벌 서비스를 보장합니다. 주요 클라우드 제공업체들은 전 세계 다양한 지역(Region)에 전략적으로 배치된 물리 데이터센터인 가용 영역을 제공합니다. 따라서 클라우드 사용자는 자체 애플리케이션을 세계 각지의 고객에게 배포하여 서비스를 확장할 수 있습니다. 과거에 이와 같은 일은 실용성 또는 경제성이 부족했거나 아예 불가능한 경우도 있었습니다.

낮은 비용

클라우드 컴퓨팅의 가장 매력적인 장점은 비용일 것입니다. 데이터센터 및 애플리케이션 서버와 달리 클라우드 컴퓨팅은 막대한 초기 비용을 지불할 필요가 없으므로 전력, 공간, 유지 관리 등 하드웨어 실행에 드는 부대 비용이 절약됩니다. 고정 비용을 지불하므로 CapEx 대신에 OpEx로 자금을 조달할 수 있어 클라우드 이니셔티브를 위한 자금을 더 쉽게 조달할 수 있습니다.

새로운 애플리케이션 개발을 당장 시작하고자 하는 소기업에게는 클라우드 컴퓨팅이 특히 경제적입니다. 클라우드 배포는 신속하고 간편하며, 클라우드 환경은 쉽게 배우고 액세스할 수 있기 때문에 개발자의 생산성은 올라가고 시장 출시 기간은 단축되면서 비용이 더욱 줄어듭니다.

클라우드 컴퓨팅을 위한 현대적인 언어에 요구되는 사항

클라우드 컴퓨팅의 장점이 거절할 수 없을 만큼 크다고 판단되지만, 그럼에도 여전히 Java를 계속 사용하고 싶다면 Java가 클라우드 환경에서 어떤 방식으로 작동하는지 이해하는 것이 중요합니다. Java는 원래 클라우드 컴퓨팅 용도로 설계되지 않았기 때문에 이 새로운 분산 환경에 맞게 진화해야 합니다.

Java가 클라우드 컴퓨팅을 위한 현대적인 언어로 거듭나기 위해 충족해야 할 요구 사항은 다음과 같습니다.

내부 및 외부 루프(Inner and outer loop) 개발

전통적인 모놀리식 애플리케이션의 경우 개발자는 데이터센터를 복제하는 환경에서 로컬로 코딩하고 테스트합니다. 개발 환경을 프로덕션 환경과 유사한 상태로 유지하면 도움이 됩니다. 애플리케이션이 마침내 프로덕션으로 이동할 때 프로덕션과 유사한 환경에서 이미 테스트와 검증을 마친 상태이기 때문입니다.

클라우드 네이티브 개발이 이 프로세스를 변경한 이유는 이전과 같은 방식으로 쿠버네티스와 클라우드 환경을 로컬 개발 환경으로 복제하는 것이 불가능하기 때문입니다. 그 결과, 이제 개발 환경은 프로덕션 환경과 유사한 클라우드 환경에서 원격으로 구현됩니다. 따라서 클라우드 환경을 위한 현대적인 언어가 원격 개발을 지원해야 합니다.

클라우드 제공업체 서비스와 통합

개발자가 AI, 데이터 레이크, Apache Kafka, API 관리와 같은 클라우드 서비스를 사용할 수 있도록 클라우드 컴퓨팅을 위한 현대적인 언어와 프레임워크는 클라우드가 제공하는 서비스와 통합되어야 합니다.

모든 하이퍼스케일러 전반에서 원활한 경험

클라우드 환경으로 이전하는 많은 조직은 위험을 완화하고 벤더 종속성을 방지하기 위해 여러 클라우드 제공업체를 사용하고 있습니다. 이 환경에서 작업하는 개발자에게 요구되는 가장 중요한 사항 중 하나는 다양한 방식으로 조합된 여러 클라우드 제공업체 전반에서 일관된 경험과 함께 익숙한 플랫폼과 툴링을 제공하는 것입니다. 바로 이러한 이유로 클라우드 컴퓨팅을 위한 현대적인 언어는 클라우드 환경에 구애받지 않아야 합니다.

클라우드 아키텍처 패턴

클라우드 컴퓨팅은 애플리케이션 개발 방식을 변화시키는 몇 가지 혁신적인 아키텍처 유형을 제공합니다. 이러한 아키텍처 패턴은 분산되어 있으며 수직 스택에 의존하지 않습니다. 이를 사용하려면 개발자가 여러 클라우드 전반에서 독립적으로 커뮤니케이션하고 확장할 수 있는 소규모 구성 요소로 이루어진 애플리케이션을 모듈식으로 빌드해야 합니다.

클라우드 컴퓨팅에서 소프트웨어 개발을 나타내는 세 가지 주요 클라우드 아키텍처 패턴은 다음과 같습니다.

이벤트 기반 아키텍처

이벤트 기반 아키텍처(EDA)는 이벤트를 통해 다른 애플리케이션 및 시스템과 비동기식으로 통신하거나 통합되는 애플리케이션을 빌드하기 위한 소프트웨어 개발 방법입니다.

이벤트는 애플리케이션이 식별하는 모든 유형의 상황 발생 또는 상태 변화일 수 있습니다. '생성자'로 지정된 애플리케이션은 이벤트를 감지하여 관련 데이터를 메시지 형식으로 전송합니다. 여러 '소비자'는 동일한 메시지를 수신할 수 있으며 관련 데이터를 자체 방식으로 사용하거나 처리하여 애플리케이션 설계 목적을 달성하기 위한 특정 작업을 완료할 수 있습니다. EDA 아키텍처는 모든 유형의 통신 또는 데이터 전송에 사용할 수 있습니다.

비동기식 이벤트 기반 통신은 두 개의 애플리케이션이 직접 연결(애플리케이션 프로그래밍 인터페이스(API)를 통한 연결이 가장 일반적)되는 더 전통적인 비동기식 통신 방법과는 다릅니다. 이와 반대로 비동기식 통신은 이벤트 기반이므로 여러 애플리케이션이 동시에 실시간으로 빠르게 통신할 수 있습니다.

EDA는 서비스 간 결합을 최소화할 것을 요구하지만 이 서비스들은 여전히 서로 통신할 수 있습니다. 따라서 EDA는 클라우드 환경에서 빌드된 현대적인 분산형 애플리케이션에 최적의 아키텍처입니다.

EDA의 장점은 다음과 같습니다.

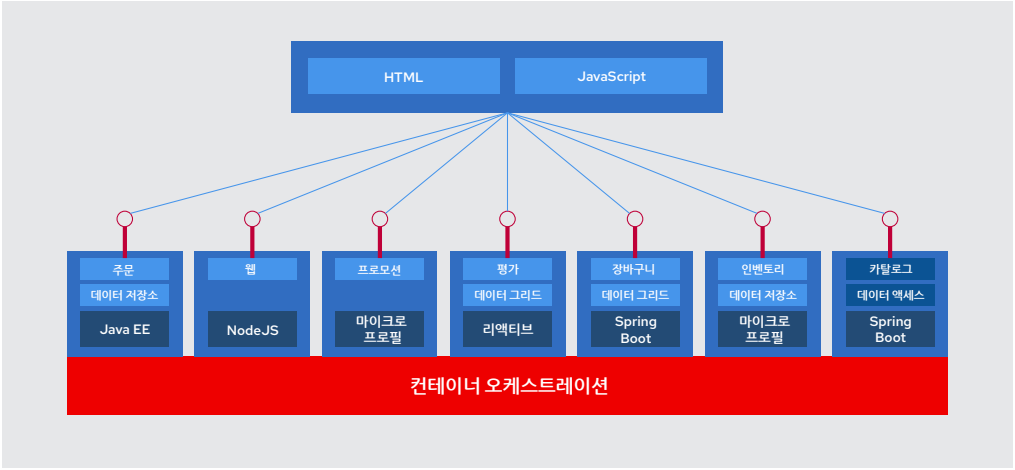
- 실시간 통신
- 짧은 대기 시간
- 확장성
- 높은 처리량
- 신뢰성

이와 반대로, 기존의 동기식 Java 애플리케이션은 클라우드 환경을 위해 설계되지 않았기 때문에 분산된 환경에서는 통신 대기 시간, 시스템 저하, 예측할 수 없는 장애와 같은 문제를 겪게 됩니다.

EDA는 스트림 처리, 데이터 통합, 웹사이트 활동 추적과 같은 확장 가능하고 신뢰할 수 있는 실시간 통신에서 장점을 끌어내는 엔터프라이즈 애플리케이션에 이상적인 아키텍처입니다. 또한 EDA는 전자 상거래 사이트나 소매점 키오스크에서 장바구니에 제품을 추가하는 것과 같이 인간이 상호 작용을 생성하는 곳에서 사용되는 경우가 많습니다.

마이크로서비스 아키텍처와 서비스 메쉬

마이크로서비스 아키텍처는 애플리케이션 빌드 및 배포 방식을 혁신한 소프트웨어 개발 방법입니다. 모든 애플리케이션의 구성 요소를 단일 배포 단위로 빌드하는 과정이 수반되는 기존의 모놀리식 개발 방식과 달리 마이크로서비스 방식은 애플리케이션을 핵심 기능이나 서비스를 대표하는 여러 개의 독립적인 구성 요소로 분할합니다. 이 서비스들은 탄력적으로 결합되어 있으므로 서로 독립적이지만 여전히 서로 통신할 수 있으며 애플리케이션으로 상호 협력할 수 있습니다.



마이크로서비스는 클라우드 네이티브 모델을 위해 애플리케이션 개발을 최적화하는 과정에서 핵심이 되는 단계를 나타냅니다. 애플리케이션은 독립적인 서비스로 구성되므로 마이크로서비스 아키텍처는 클라우드 컴퓨팅을 위한 완벽한 솔루션입니다. 애플리케이션을 여러 클라우드 환경과 온프레미스 하드웨어 전반에 배포할 수 있습니다.

마이크로서비스의 주요 장점은 개발 라이프사이클을 가속화하고 시장 출시 시간을 단축한다는 것입니다. 애플리케이션은 서로 분리된 서비스의 모음으로 구성되므로 이 모든 마이크로서비스를 여러 개발자가 독립적으로 빌드, 테스트, 배포할 수 있습니다. 이 방법은 개발자가 어떤 유형의 변경이 이루어질 때마다 전체 애플리케이션이 다시 컴파일되고 다시 배포될 때까지 기다려야 하는 경우 일반적으로 발생하는 지연 시간을 최소화합니다. 마이크로서비스를 사용하면 애플리케이션의 나머지 부분에 영향을 미치지 않고 애플리케이션의 한 가지 구성 요소를 변경할 수 있습니다. 이러한 방식으로 애플리케이션 업데이트와 수정을 훨씬 더 쉽고 빠르게 실행할 수 있습니다.

더 빠른 개발 속도 외에도 마이크로서비스는 다음과 같은 장점을 제공합니다.

- **품질:** 마이크로서비스는 소규모이므로 개발자가 문제를 찾아내 수정하기가 더 쉽습니다. 따라서 애플리케이션의 품질이 향상됩니다.
- **확장성:** 마이크로서비스는 여러 클라우드 환경 전반에서 배포 가능하므로 애플리케이션을 빠르고 경제적으로 확장할 수 있고, 이러한 확장을 통해 요구 사항을 충족할 수 있습니다.
- **복구 능력:** 독립 서비스인 마이크로서비스는 서로에게 영향을 미치지 않으므로 하나의 서비스 내부에서 장애가 발생한 경우 전체 애플리케이션이 중단되지 않습니다.
- **혁신:** 마이크로서비스를 통해 개발자는 애플리케이션에 영향을 미치고 시장 출시 시간을 지연시킬 수 있다는 염려 없이 변경과 실험을 진행할 수 있으므로 더 자유롭게 혁신을 추진할 수 있습니다.

마이크로서비스 아키텍처가 정상적인 클라우드 애플리케이션처럼 작동하려면 서비스 간에 메시징을 통해 데이터를 끊임없이 요청해야 합니다. 서비스 메쉬 계층을 구축하면 서비스 간 통신이 간소화됩니다.

서비스 메쉬는 애플리케이션 외부에 상주하는 투명한 전용 인프라 계층으로서, 애플리케이션을 구성하는 다양한 마이크로서비스가 서로 데이터를 공유하는 방식을 제어하기 위해 설계되었습니다. 서비스 메쉬의 목적은 서비스 간 통신을 최적화하고 잠재적인 통신 장애 지점을 최소화하는 것입니다.

서비스 메쉬는 서비스 간 통신을 제어하는 로직을 개별 서비스에서 제거하고 이를 인프라의 계층으로 추상화합니다. 이러한 작업은 마이크로서비스 간 네트워크 통신을 가로채는 사이드카(sidecar) 프록시를 사용하여 메쉬 내부의 네트워크화된 서비스에 대한 인사이트와 제어 기능을 제공함으로써 완료됩니다. sidecar 프록시는 마이크로서비스와 나란히 위치하며 다른 프록시로 요청을 라우팅합니다. 이러한 sidecar들이 모여 메쉬 네트워크를 형성합니다.

클라우드 네이티브 애플리케이션은 마이크로서비스를 사용해 빌드되므로 서비스 메쉬는 클라우드 환경에서 매우 유용할 수 있습니다. 또한 더 많은 마이크로서비스가 애플리케이션에 추가되면서 클라우드 환경에서 통신을 관리하는 작업이 복잡하고 까다로운 일이 될 수 있습니다. 서비스 메쉬의 장점은 마이크로서비스 간 연결성을 제공하고, 유연성, 신뢰성, 빠른 속도, 관리 용이성을 갖춘 통신을 지원한다는 것입니다.

서비스 메쉬가 없으면 각 마이크로서비스는 서비스 간 통신을 지원하는 로직으로 코딩해야 하기 때문에 개발자의 업무 부담이 늘어납니다. 서비스 메쉬는 이와 같은 추가 코딩의 필요성을 제거함으로써 개발 프로세스를 간소화합니다.

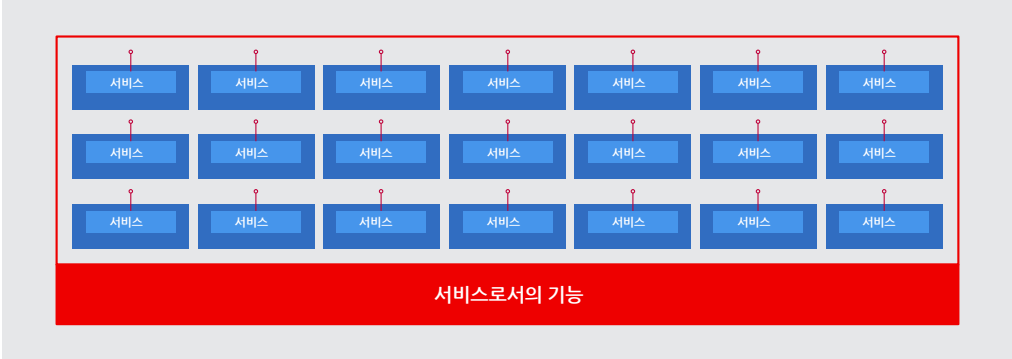
또한 서비스 메쉬를 통해 개발자는 마이크로서비스에 라우팅 제어, 내결함성, 보안과 같은 추가 기능을 도입하면서도 마이크로서비스 구성 요소 자체를 변경하지 않을 수 있습니다.

또한 서비스 메쉬는 통신 성능과 가용성을 보장하기 위해 관측성, 모니터링, 테스트 기능을 제공합니다. 문제의 원인이 마이크로서비스 내에 숨겨져 있지 않고 서비스와 병행하는 가시적인 인프라 계층에 있으므로 서비스 메쉬를 통해 통신 장애를 더 쉽게 해결할 수 있습니다. 궁극적으로 이러한 장점 덕분에 애플리케이션이 더 강력해지고 다운타임에 대한 취약성이 완화됩니다.

또한 서비스 메쉬는 스스로 학습하여 통신을 더욱 개선하는 능력이 있습니다. 서비스 메쉬는 서비스 간 통신에 관한 성능 메트릭을 수집하고, 이 데이터를 사용해 서비스 요청의 효율성과 신뢰성을 자동으로 개선합니다.

서버리스 아키텍처

서버리스 아키텍처는 개발자가 기본 인프라를 관리할 필요 없이 애플리케이션을 빌드하고 실행할 수 있는 클라우드 네이티브 개발 모델입니다. 클라우드 제공업체가 서버에 대한 프로비저닝, 유지 관리, 스케일링 등의 일상적인 작업을 처리하므로 개발자는 코딩에 집중할 수 있습니다.



서버리스 애플리케이션은 호출 시 자동으로 시작되는 컨테이너에 배포됩니다. 애플리케이션은 다른 애플리케이션에서 발생하는 이벤트, 클라우드 서비스, 서비스로서의 소프트웨어(SaaS) 시스템, 기타 서비스 등 다양한 소스로부터 트리거될 수 있습니다. 배포된 서버리스 애플리케이션은 요구 사항에 따라 자동으로 스케일업 또는 스케일다운됩니다. 따라서 서버리스 애플리케이션은 리소스가 필요할 때만 해당 리소스를 소비합니다.

서버리스는 표준 서비스로서의 인프라(IaaS) 클라우드 컴퓨팅 모델과는 다릅니다. 전통적인 시나리오에서 사용자는 상시 가동 서버가 애플리케이션을 실행하도록 하기 위해 클라우드 제공업체에 요금을 지불합니다. 애플리케이션을 실행하기 위해 필수적인 클라우드 인프라는 애플리케이션을 사용하지 않을 때도 활성화되어 있습니다.

하지만 서버리스 아키텍처에서는 애플리케이션이 필요한 경우에만 시작됩니다. 애플리케이션이 실행되면 클라우드 제공업체가 리소스를 할당합니다. 사용자는 애플리케이션이 실행 중인 동안에만 리소스에 대해 요금을 지불합니다.

전형적인 서버리스 모델은 이벤트 기반 컴퓨팅 실행 모델인 서비스로서의 기능(FaaS)입니다. 이 모델을 통해 개발자는 클라우드 제공업체가 관리하는 컨테이너에 배포된 후 온디맨드 방식으로 실행되는 로직을 작성할 수 있습니다. 애플리케이션은 이벤트로 트리거되며 필요 시 자동으로 실행됩니다.

서버리스는 수요 급증이 드물지만 예기치 않게 발생하는 애플리케이션에 적합합니다. 또한 서버리스는 수신 데이터 스트림, 채팅 봇, 예정된 태스크, 비즈니스 프로세스 자동화와 관련된 활용 사례에도 유용합니다.

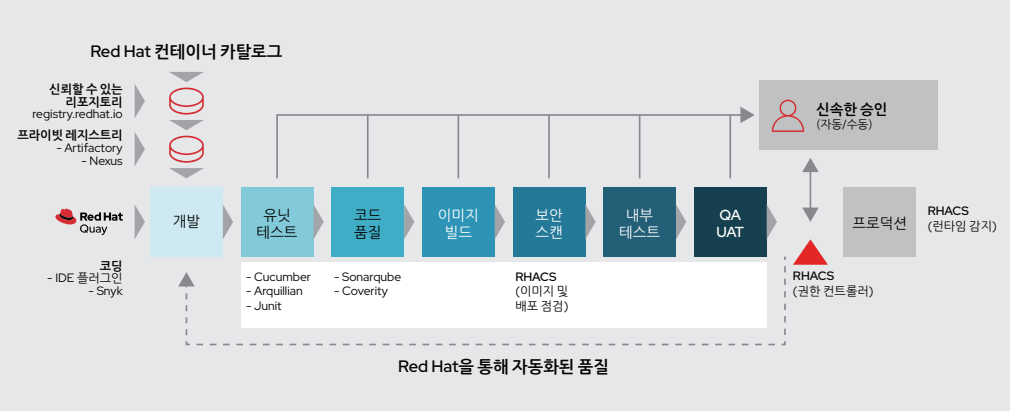
서버리스는 비용을 절감할 수 있다는 명백한 장점 외에도 개발자가 여러 클라우드 환경 전반에서 워크로드를 확장하고 배포할 수 있도록 지원합니다. 서버리스 모델은 매우 유연하여 애플리케이션을 최대한의 리소스로 확장하고 사용한 후 정기적으로 다시 규모를 축소할 수 있습니다.

또한 서버리스는 스케일링 및 서버 프로비저닝과 관련된 일상적이고 시간이 많이 소요되는 태스크 부담을 덜어 개발자가 비즈니스 가치가 높은 애플리케이션과 기능을 혁신하는 데 더 많은 시간을 할애할 수 있도록 지원합니다. 서버리스를 활용하면 운영 체제, 파일 시스템, 보안 패치, 부하 분산, 용량 관리, 스케일링, 로깅, 모니터링과 같은 태스크를 모두 클라우드 제공업체에 이관할 수 있습니다.

기존의 Java 프레임워크는 특히 쿠버네티스에 배포하는 경우 클라우드 환경에 대한 서버리스 배포를 수행하기에는 일반적으로 너무 무겁고 느립니다. Java 프레임워크가 클라우드 환경에서 효과적으로 작동하려면 더 빠른 시작 시간, 더 적은 메모리 소비, 더 작은 애플리케이션 크기를 제공할 수 있도록 진화해야 합니다.

DevOps/DevSecOps에 집중

DevOps는 개발 팀과 IT 운영 팀을 더 협업적인 엔드 투 엔드 개발 프로세스로 결합하는 IT 문화에 대한 접근 방식입니다. DevOps 접근 방식에서는 개발 팀과 IT 운영 팀이 고품질 애플리케이션 개발 및 배포에 대한 공동 책임을 집니다. 이는 많은 경우, 조기에 IT 운영 팀의 요구 사항을 개발 주기에 적용함으로써 애플리케이션이 프로덕션 요구 사항을 염두에 두고 빌드되도록 보장함을 뜻합니다. DevOps를 올바르게 수행하면 애플리케이션 개발 라이프사이클을 간소화하고 더 나은 애플리케이션을 더 빨리 출시할 수 있습니다.



DevSecOps는 이와 유사한 접근 방식으로서, 보안 팀이 평상시보다 조기에 프로세스에 참여하도록 하여 개발 및 IT 운영 팀과 협력하게 함으로써 애플리케이션의 보안을 유지합니다.

DevOps 및 DevSecOps의 핵심은 가시성, 피드백, 얻은 교훈, 기타 인사이트를 공유하는 과정을 통해 협업을 진행하는 것입니다. 또한 DevOps 및 DevSecOps 접근 방식은 시장 출시 시간을 단축하기 위해 개발 주기를 최대한 자동화하는 데 집중합니다. 이 모든 목표를 달성하려면 협업 방식의 자동화된 개발 환경을 지원할 수 있는 적합한 플랫폼과 툴이 필요합니다.

DevOps는 클라우드 네이티브 개발과 병행하여 진행됩니다. 두 이니셔티브 모두 양질의 애플리케이션을 더 빠르게 개발한다는 동일한 목표를 추구하기 때문입니다. 클라우드 컴퓨팅을 위한 현대적인 언어는 다음을 포함한 DevOps를 지원할 수 있는 기능을 기본적으로 제공해야 합니다.

CI/CD: 지속적인 통합, 지속적인 서비스 제공, 지속적인 배포는 클라우드 네이티브 개발을 지원하는 DevOps 개념입니다. 지속적인 통합은 다양한 개발자가 애플리케이션에 대해 수행한 여러 가지 변경이 공유 리포지토리에 병합된다는 뜻입니다. 이는 해당 애플리케이션에 대한 작업을 수행하는 모든 개발자가 동일한 버전의 애플리케이션에 액세스할 수 있도록 보장하기 위한 것입니다. 지속적인 서비스 제공 또는 지속적인 개발은 개발 프로세스 간소화를 위해 애플리케이션에 대한 변경 사항이 자동으로 테스트되고 배포되는 것을 뜻합니다. 클라우드 컴퓨팅을 위한 현대적인 언어는 CI/CD 파이프라인이 가속화된 개발과 향상된 품질의 장점을 누릴 수 있도록 지원해야 합니다.

자동화: 클라우드 네이티브 개발 단계에서는 클라우드 환경에 분산되어 있는 애플리케이션의 복잡성으로 인해 자동화가 매우 중요합니다. 여러 마이크로서비스로 구성된 여러 개의 애플리케이션이 여러 클라우드 환경 전반에 분산되어 있는 경우 개발하고 관리하기 까다로울 수 있습니다. 통합, 테스트, 배포와 같은 프로세스를 자동화하면 이러한 어려움을 일부 경감하고, 개발 프로세스를 크게 가속화하는 동시에 인적 오류를 줄여 품질을 높일 수 있습니다.

Quarkus를 이용해 클라우드 컴퓨팅에 Java 도입

Quarkus는 입증된 Java 라이브러리 및 표준을 기반으로 구축된 쿠버네티스 네이티브 Java 스택으로서, 컨테이너 및 클라우드 배포를 위해 설계되었습니다. Quarkus를 통해 조직은 클라우드 컴퓨팅 및 쿠버네티스를 위해 언어를 변환함으로써 수년에 걸친 투자의 효과를 극대화할 수 있습니다. 이로써 Quarkus는 개발자가 Java 관련 기존 지식과 경험뿐 아니라 과거에 사용하던 Java 프레임워크도 계속 활용할 수 있는 경로를 제공합니다.

Quarkus는 컨테이너, 마이크로서비스, 서버리스와 같은 클라우드 네이티브 애플리케이션 아키텍처와 관련해 Java가 지닌 한계를 해결하기 위해 설계되었습니다. Quarkus는 전통적인 아키텍처와 클라우드 네이티브 아키텍처를 모두 지원하므로 하이브리드 클라우드 컴퓨팅에 이상적인 Java 프레임워크입니다.

Java 개발자는 Quarkus를 사용해 기존의 Java 기반 마이크로서비스 프레임워크보다 시작 시간이 짧고 메모리를 적게 차지하는 애플리케이션을 빌드할 수 있습니다. 동일한 애플리케이션을 실행하는 데 더 적은 메모리와 CPU를 사용하므로 비용이 절감되는 효과로 나타납니다. 또한 개발자가 대규모 애플리케이션이 변경 사항 테스트를 위해 다시 빌드되고 배포될 때까지 기다리지 않아도 되므로 생산성이 향상되는 효과로도 있습니다. Quarkus를 이용해 개발자는 코드를 변경하고 애플리케이션에서 이를 즉시 확인할 수 있습니다.

Quarkus는 널리 사용되는 Java 표준, 프레임워크, 라이브러리와 즉시 연동되므로 새로운 API를 배우거나 다른 프로그래밍 언어로 완전히 전환해야 할 필요성이 줄어듭니다. 따라서 개발자는 Quarkus와 함께 사용하고 싶은 Java 프레임워크를 선택할 수 있고, 이를 통해 Quarkus는 개발자 생산성과 빠른 성능이라는 장점을 추가로 제공합니다.

다음은 Quarkus가 Java에 추가로 제공하는 기능, 특징, 툴입니다.

더 적은 코드

Quarkus를 사용하는 개발자는 더 적은 양의 코드를 작성하면서도 기존의 Java에 비해 더 많은 기능을 구축할 수 있습니다. 이를 통해 애플리케이션의 지속 가능성이 향상됩니다. Quarkus를 사용해 빌드한 초경량화 및 초고속 애플리케이션을 이용해 풋프린트를 추가하지 않고도 수백 개의 애플리케이션 사본을 배포하고 처리량을 극대화할 수 있습니다.

고성능

Quarkus는 단지 Java를 클라우드로 변환하는 데 그치지 않고 Java를 한 단계 높은 수준으로 발전시킵니다. 지난 10년 동안 Java의 속도를 높이려는 시도가 반복되었지만 이 모든 시도는 상단에 위치한 프레임워크가 아닌 기본 Java 기술인 Java VM에 집중되어 있었습니다. Quarkus는 프레임워크 자체를 최적화하여 엄청나게 빠른 성능을 달성합니다.

사용자는 Quarkus에서 동일한 리소스를 이용해 기존의 Java에 비해 더 많은 애플리케이션을 배포할 수 있습니다. Quarkus가 높은 처리량을 제공하는 이유는 네트워크 전반에 배포된 Quarkus의 복사본 여러 개를 매우 조밀하게 배포할 수 있기 때문입니다.

Quarkus는 컨테이너 우선 전략을 바탕으로 구축되었습니다. 이는 Quarkus가 더 적은 메모리 사용과 더 빠른 시작 시간을 달성하는 데 최적화되어 있음을 뜻합니다. Quarkus는 기존 Java에 비해 메모리 사용량이 1/10인 애플리케이션을 빌드하며 시작 시간이 최대 300배 더 빠릅니다.

라이브 코딩

Quarkus는 '라이브 코딩'으로 개발 중에 신속하게 반복할 수 있도록 지원합니다. 코드 변경 사항은 실행 중인 애플리케이션에 즉시 자동으로 반영됩니다.

기존 Java 워크플로우는 변경 사항이 있을 때마다 개발자가 애플리케이션을 다시 컴파일하고 배포해야 하는데, 이 작업에 최대 1분 이상이 소요됩니다. 이로 인해 Java 개발자가 겪는 지연 시간이 상당히 늘어납니다. 개발자는 라이브 코딩을 통해 이동 중에도 변경할 수 있고 매번 전체 애플리케이션을 다시 컴파일하고 배포할 필요 없이 브라우저를 새로 고치기만 하면 되므로 생산성이 향상됩니다. 일반적으로 Quarkus는 이러한 변경을 1초도 안 되는 짧은 시간에 구현합니다.

지속적인 테스트

Quarkus에서 단순한 키 조작으로 사용할 수 있는 지속적인 테스트 기능을 통해 개발자는 테스트 기반 개발을 계속할 수 있습니다.

기존 Java 개발 라이프사이클에서 개발자는 코드와 테스트를 작성하고, 테스트를 실행하며, 테스트를 통과하는지 여부를 확인한 다음, 변경합니다. 개발자는 Quarkus에서 지속적인 테스트 기능을 이용해 유닛 테스트를 계속 실시간으로 실행하고 이와 동시에 코드를 작성할 수 있습니다. 테스트는 백그라운드에서 자동으로 실행되며, 지속적으로 피드백을 제공합니다. 키보드 입력을 하나라도 잘못하면 테스트가 실패하게 되며, 이를 즉시 알 수 있습니다. 이러한 기능 덕분에 개발 주기가 크게 단축됩니다.

개발 서비스

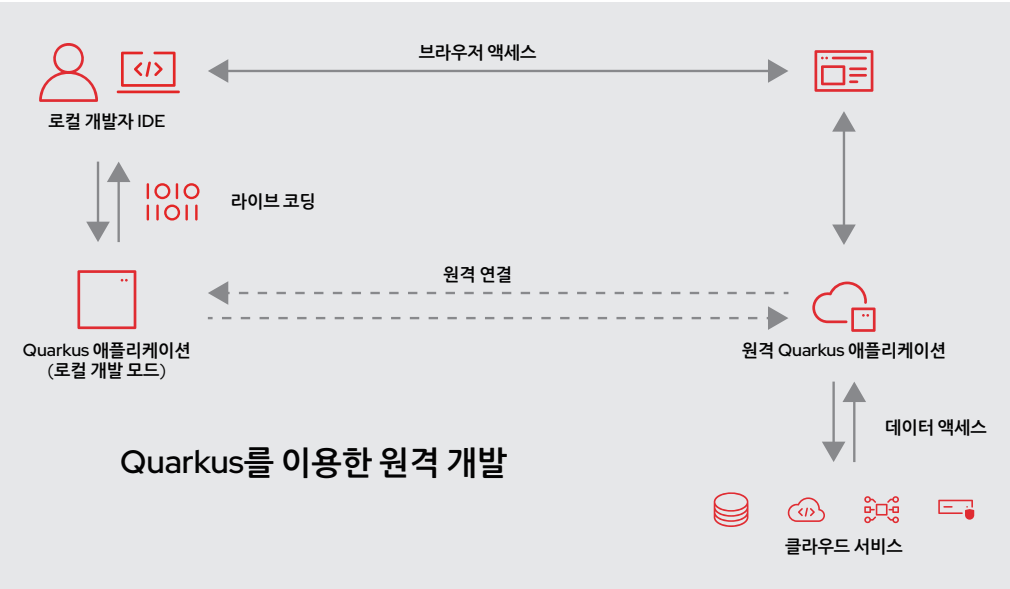
Quarkus 개발 서비스를 통해 개발자는 애플리케이션 종속성을 손쉽게 테스트할 수 있습니다.

모든 애플리케이션은 다른 애플리케이션 및 서비스에 대한 종속성이 있습니다. 기존 Java 개발에서 개발자는 애플리케이션을 완벽하게 테스트하기 위해 연결된 모든 서비스의 복사본을 가동해야 합니다. 하지만 Apache Kafka, 메시지 브로커, Identity 관리 시스템과 같은 일부 서비스는 복제하기 매우 까다로울 수 있습니다. Quarkus는 애플리케이션을 테스트하는 데 필요한 서비스를 자동으로 제공하는 개발 서비스로 이러한 문제를 해결합니다.

예를 들어, 애플리케이션에 데이터베이스가 필요한 경우 Quarkus는 이러한 요구 사항을 식별하고, 적절한 데이터베이스를 알아낸 후 데이터베이스를 가동하고, 애플리케이션을 연결합니다. 이 모든 작업은 자동으로 이루어집니다.

원격 개발

클라우드 네이티브 Java 런타임에서 원격 개발을 수행할 수 있으므로 코드 작성에서부터 마이크로서비스를 신속하게 빌드, 실행, 디버깅, 배포하는 데 이르는 개발 워크플로우가 간소화됩니다.



Quarkus 원격 개발을 통해 개발자는 원격 컨테이너 환경에서 애플리케이션을 실행하면서도 로컬 노트북을 통해 액세스할 수 있습니다. 로컬 개발 머신의 변경 사항은 실행 중인 원격 Quarkus 애플리케이션으로 실시간으로 자동 푸시됩니다.

원격 개발의 장점은 원격 환경이 애플리케이션의 프로덕션 환경과 더 가까워 테스트의 정확도가 높아진다는 것입니다. 이와 관련된 원격 개발의 장점은 개발자의 로컬 머신에서는 사용할 수 없거나 다시 생성하기 쉽지 않은 클라우드 환경에서 서비스에 액세스할 수 있다는 것입니다. 대체로 원격 개발을 통해 개발자는 애플리케이션이 프로덕션 단계에서 실행되고 변경 사항을 개발하고 테스트하는 데 필요한 시간을 획기적으로 단축할 것이라고 확신할 수 있습니다.

Red Hat OpenShift에 기반한 클라우드 및 쿠버네티스 네이티브 개발

과거에는 애플리케이션을 쿠버네티스로 배포하는 것은 어려운 일이었습니다. 하지만 Quarkus는 쿠버네티스와 Red Hat의 기본 쿠버네티스 기반 플랫폼인 Red Hat® OpenShift®에 대해 기본적으로 잘 알고 있습니다. Quarkus는 쿠버네티스와 Red Hat OpenShift에서 애플리케이션을 배포하는 과정을 자동화하여 모든 구성 작업과 스크립트 작성을 처리함으로써 사용자에게 쿠버네티스 네이티브 개발 경험을 제공합니다.

Quarkus 쿠버네티스 확장이나 Quarkus OpenShift 확장을 추가하기만 하면 프로젝트에 따라 쿠버네티스 또는 OpenShift 리소스가 자동으로 생성됩니다.

클라우드 서비스와 통합

Quarkus는 주요 클라우드 제공업체가 제공하는 다양한 서비스도 기본적으로 인식하고 있으므로 개발자는 애플리케이션을 이러한 서비스와 손쉽게 통합할 수 있습니다.

예를 들어, Quarkus Funqy는 이동형 Java API로서, AWS Lambda, Azure Functions, Knative, Knative Events와 같은 서비스로서의 기능(FaaS) 환경에 배포할 수 있는 기능을 작성하는 데 사용됩니다.

또 하나의 예로, Quarkus를 통해 개발자는 S3, SNS, Alexa와 같은 AWS 서비스뿐 아니라 기타 클라우드 제공업체 서비스도 사용할 수 있습니다.

기존 Java 프레임워크와 통합

Quarkus는 개발자가 잘 알고 선호하는 Java 프레임워크를 계속 사용하도록 허용합니다. Quarkus는 Eclipse MicroProfile과 Spring뿐 아니라 Apache Kafka, RESTEasy(JAX-RS), Hibernate ORM(JPA), Infinispan, Camel, 그 외 다수의 널리 사용되는 Java 표준, 프레임워크, 라이브러리와 연동되도록 설계되었습니다.

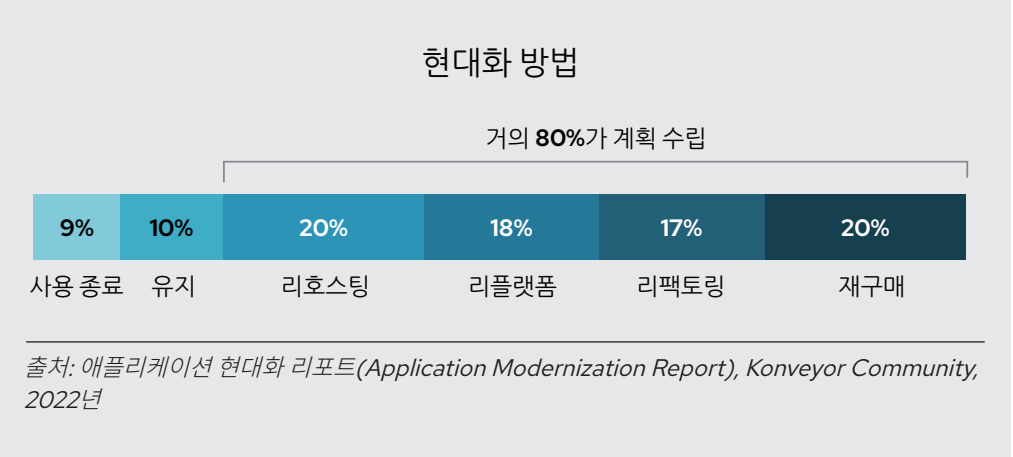
CI/CD, 관측성, 추적, 텔레메트리와 통합

기존 Java에서 마이크로서비스와 같은 분산형 아키텍처로 이동하는 과정에서 애플리케이션이 더 복잡해집니다. 분산된 서비스와 EDA를 사용하면 관측, 추적, 디버깅에 수반되는 문제점이 기하급수적으로 증가합니다. Quarkus는 관측성, 추적, 텔레메트리, CI/CD 시스템에 즉각적인 지원을 제공함으로써 이러한 고충을 최소화합니다.

클라우드를 위한 경로

Java 애플리케이션을 현대화하고, 모놀리식 아키텍처에서 벗어나고, 클라우드 배포 모델을 활용하려면 리호스팅(rehost), 리플랫폼, 리팩토링이라는 세 가지 주요 경로를 통해 Java 애플리케이션을 현대화할 수 있습니다.

모든 애플리케이션을 똑같은 경로로 현대화해야 하는 것은 아닙니다. 각 애플리케이션의 특성을 비롯하여 조직의 현재 및 향후 요구 사항에 가장 적합한 경로를 선택할 수 있습니다.



출처: 애플리케이션 현대화 리포트(Application Modernization Report), Konveyor Community, 2022년

경로 1: 리호스팅

'리프트 앤 시프트'라고도 하는 리호스팅 경로는 기존 애플리케이션을 가상 머신(VM) 내부에 있는 것처럼 배포하는 방법입니다.

리호스팅에는 기존 애플리케이션 서버에서 실행되는 Java 애플리케이션을 하이브리드 클라우드 플랫폼에서 실행되는 VM으로 리프트 앤 시프트하는 작업이 포함됩니다. 모놀리식 애플리케이션은 애플리케이션 서버에서 변함없이 유지되며 기존의 모든 통합 및 종속성을 그대로 유지합니다. 외부 데이터와 통합을 계속해서 레거시 플랫폼을 기반으로 유지할 수 있습니다.

리호스팅은 일반적으로 짧은 시간 내에 완료되므로 마이그레이션 비용이 적게 들지만 다른 현대화 경로보다는 장점이 적습니다. 이를 감안해도 리호스팅은 애플리케이션을 일관된 플랫폼으로 이전하는 데 여전히 도움이 되며 향후 클라우드 네이티브 운영에 대비하는 데도 도움이 될 수 있습니다.

기존 애플리케이션 서버 중 일부는 VM에서 작동하지 않을 수 있습니다. 이 경우 VM으로 이전하기 전에 현대적인 실행(runtime) 환경에서 애플리케이션을 다시 배포해야 합니다. 실행(runtime) 환경을 변경해야 하는 경우에는 애플리케이션의 플랫폼을 재구성(경로 2)하여 컨테이너에 배포함으로써 현대화 노력을 최적화하는 방안을 고려해 보세요.

경로 2: 리플랫폼

리플랫폼은 쿠버네티스 기반 클라우드 플랫폼에서 애플리케이션을 컨테이너에 배포하는 방법입니다.

리플랫폼에는 Java 애플리케이션을 리프트 및 수정하고 하이브리드 클라우드 플랫폼의 컨테이너에서 실행되는 현대적인 실행(runtime) 환경으로 시프트하는 과정이 수반됩니다. 기본적인 Java 애플리케이션은 거의 변경하지 않고도 OpenJDK와 같은 컨테이너화된 Java 런타임의 장점을 누릴 수 있습니다.

엔터프라이즈 애플리케이션은 컨테이너에서 현대적인 실행(runtime) 환경(예: Red Hat JBoss Enterprise Application Platform, IBM WebSphere Liberty 또는 Red Hat JBoss Web Server)을 배포하기 전에 이러한 환경으로 마이그레이션됩니다. 이 경로는 일반적으로 리호스팅보다 시간이 더 오래 걸리지만 더 많은 장점을 제공합니다. 단일 하이브리드 클라우드 플랫폼에서 애플리케이션을 통합하면 운영이 간소화되고 셀프 서비스 기능을 제공할 수 있습니다. 리플랫폼된 애플리케이션은 하이브리드 클라우드 플랫폼의 모든 네이티브 기능도 이용할 수 있습니다.

경로 3: 리팩토링

리팩토링 경로는 애플리케이션을 마이크로서비스로 다시 빌드하고, 새로운 기술을 통합하며, 클라우드 플랫폼에 배포하는 방법입니다.

리팩토링에는 하이브리드 클라우드 플랫폼 상의 서비스 메쉬 내에 배포된 마이크로서비스로서의 Java 애플리케이션을 다시 개발하는 과정이 포함됩니다. 시간이 지나면 서비스를 재구축할 수 있으므로 기존 애플리케이션 아키텍처의 기능을 점차 새로운 애플리케이션 아키텍처로 이동할 수 있습니다.

이 프로세스 중에 기본 기술을 업그레이드하고 인공지능 및 머신 러닝(AI/ML), 분석, 자동 스케일링, 서버리스 기능, 이벤트 기반 아키텍처와 같은 새로운 클라우드 네이티브 기능에 추가할 수도 있습니다.

리팩토링은 시간이 가장 많이 소요되지만 가장 큰 장점을 제공합니다. 리팩토링은 리호스팅 및 리플랫폼의 장점을 모두 제공하면서도 혁신적인 신기술을 이용해 비즈니스 민첩성 및 가치를 증진하는 데 도움이 됩니다.

고객 사례

Quarkus를 이용해 Java를 클라우드 환경으로 이전하는 데 성공한 Red Hat 고객의 사례를 몇 가지 소개합니다.

Asiakastieto Group

Asiakastieto Group은 북유럽 지역에서 혁신적인 디지털 비즈니스와 소비자 정보 서비스를 제공하는 선도적인 업체입니다. 오픈 뱅킹으로의 전환을 지원하고, 새로운 데이터 프라이버시 및 보안 요구 사항을 준수하고, 북유럽의 높은 부채율을 해결하기 위해 이 회사는 신용 평가 솔루션을 구축했습니다. Asiakastieto는 개인 부채와 채무 불이행을 줄이고 개인의 상환 능력을 더 정확히 평가하기 위해 Red Hat OpenShift, Red Hat Integration, Quarkus를 사용해 자체 계정 인사이트 애플리케이션을 개발했습니다. 또한 이 회사는 Thorntail에서 Quarkus로 마이그레이션하여 마이크로서비스 시작 시간, 메모리 사용량, 서버 밀도를 개선함으로써 하드웨어 리소스를 최적화했습니다.

Bankdata

덴마크의 여러 대규모 은행에 IT 서비스를 제공하는 Bankdata는 고품질 IT 솔루션을 빌드, 구현, 실행하고 있습니다. 최상의 Java 프레임워크를 찾기 위해 Bankdata는 Quarkus 대비 자체 Spring Boot Java 프레임워크의 성능과 효율성을 테스트했습니다. 그 결과, 테스트 애플리케이션의 Quarkus 네이티브 버전은 더 빠른 부팅 시간(Spring의 경우 3분 걸리는 데 비해 1초도 안 걸림), 처리되는 호출당 57% 적은 메모리 사용량, 더 적은 CPU 사용량을 제공하는 것으로 나타났습니다. Bankdata는 Quarkus로 마이그레이션함으로써 테스트 시간이 단축되어 새로운 서비스를 더 빠르게 출시할 수 있었습니다.

Lufthansa Technik

Lufthansa Technik은 항공사가 유지 보수 체계화 및 일정 수립을 개선함으로써 비행 지연과 취소를 미연에 방지하도록 지원하는 AVIATAR라는 디지털 플랫폼을 운영하고 있습니다. 빠른 성장과 고객의 요구 사항 증가에 대처하기 위해 이 회사는 Microsoft Azure Red Hat OpenShift에 기반을 둔 마이크로서비스 아키텍처로 이전하기로 결정했습니다. AVIATAR 팀은 클라우드 리소스 사용량을 줄이기 위해 Quarkus를 배포했습니다. Lufthansa Technik은 팀이 서비스 가용성과 대응 시간을 희생하지 않고도 3배 더 조밀한 배포를 실행할 수 있었습니다. 또한 Quarkus는 개발 라이프사이클을 가속화하는 데 도움이 되었습니다. 예를 들어, 두 명으로 구성된 팀이 단 3주 내에 '고객 구성' 서비스라고 하는 새로운 마이크로서비스를 개발했습니다.

자세히 알아보기

이 토픽에 관한 보충 정보는 Red Hat OpenShift에서 Quarkus로 Java 애플리케이션을 개발해야 하는 이유를 참조하세요.

f www.facebook.com/redhatkorea
구매문의 080 708 0880
buy-kr@redhat.com

www.redhat.com/ko
#F31646_0622