RED HAT®
CONSULTING

redhat.

백서

컨테이너 기반 애플리케이션 설계의 원칙

작성자: Bilgin Ibryam

핵심 요약

클라우드 네이티브는 클라우드 기반 인프라에서 실행되도록 특별히 설계된 애플리케이션을 표현하는 용어입니다. 일반적으로 클라우드 기반 애플리케이션들은 플랫폼이 관리하는 컨테이너 내에서 실행되는 유연한 마이크로서비스로서 개발되었습니다. 이 애플리케이션들은 장애를 예측하며 기본 인프라가 중단되는 상황에서도 안정적으로 실행되고 확장합니다. 이와 같은 기능을 제공하기 위해 클라우드 기반 플랫폼은 이를 기반으로 실행되는 애플리케이션에 일련의 계약과 제약 조건을 부여합니다. 이 계약들은 애플리케이션들이 특정 제약 조건을 준수하도록 하며 플랫폼이 컨테이너화된 애플리케이션의 관리를 자동화할 수 있도록 보장합니다.

많은 기업들은 클라우드 네이티브 환경을 구현해야 한다는 필요성과 중요성을 인식하고 있지만, 어디서부터 시작해야 할지 확신하지 못하고 있습니다. 클라우드 네이티브 플랫폼과 여기에서 실행되는 컨테이너화된 애플리케이션이 원활하게 함께 실행되도록 보장함으로써 장애를 예측할 수 있는 것은 물론, 기본 인프라가 중단되는 상황에서도 실행하고 확장할 수 있는 신뢰성을 확보하게 됩니다. 이 백서에서는 컨테이너화된 애플리케이션들이 적합한 클라우드 네이티브 구성 요소가 되기 위해서 준수해야 하는 많은 원칙들을 설명하고 있습니다. 이들 원칙을 준수함으로써 기업 애플리케이션들을 Kubernetes와 같은 클라우드 네이티브 플랫폼의 자동화에 적합하게 만들 수 있습니다.

소프트웨어 설계의 원칙:

- 간단하고 쉽게 유지하기(KISS, Keep It Simple, Stupid)
- 동일 작업 반복 하지 않기(DRY, Don't Repeat Yourself)
- 필요 없는 기능 추가 하지 않기(YAGNI, You Aren't Gonna Need It)
- 관심사 분리(SoC, Separation Of Concerns)

클라우드 네이티브 컨테이너에 대한 RED HAT의 접근 방식:

- 단일 관심사 원칙(SCP, Single Concern Principle)
- 고 관측 원칙 (HOP, High Observability Principle)
- 라이프사이클 준수 원칙(LCP, Life-cycle Conformance Principle)
- 이미지 불변성 원칙(IIP, Image Immutability Principle)
- 프로세스 폐기 가능성 원칙 (PDP, Process Disposability Principle)
- 독립성 원칙(S-CP, Self-Containment Principle)
- 런타임 견제 원칙 (RCP, Runtime Containment Principle)

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

www.redhat.com/ko

컨테이너 관련 공통 모범 사례:

- 작은 이미지 구축
- 임의의 사용자 ID 지원
- 중요 포트 표시
- 퍼시스턴트 데이터를 위한 볼륨 사용
- 이미지 메타데이터 설정
- 호스트와 이미지 동기화

소프트웨어 설계의 원칙

원칙은 삶의 여러 분야에 존재하며, 일반적으로 다른 사람들로부터 이끌어낸 근본적인 진실 또는 믿음을 나타냅니다. 소프트웨어에서 원칙은 소프트웨어를 설계할 때 따라야 하는 매우 추상적인 지침입니다. 이 원칙들은 프로그래밍 언어에 적용되고 다양한 패턴을 사용해 구현되며 다양한 실행 방식에 따라 달성될 수 있습니다.

일반적으로 패턴과 실행 방식은 원칙이 목표로 하는 결과를 달성하기 위해 사용되는 도구이며, 다른 모든 원칙들에서 도출해낸 우수한 소프트웨어 작성을 위한 기본 원칙들이 있습니다. 이러한 원칙들은 다음과 같습니다.

- **KISS** - 간단하고 쉽게 유지하기(Keep it simple, stupid)
- **DRY** - 동일 작업 반복 하지 않기(Don't repeat yourself)
- **YAGNI** - 필요 없는 기능 추가 하지 않기(You aren't gonna need it)
- **SoC** - 관심사 분리(separation of concerns)

이 원칙들이 구체적인 규칙들을 명시하지는 않지만, 다수의 개발자들이 이해하고 꾸준히 참고하는 언어와 상식을 대변하고 있습니다.

Robert C. Martin이 처음 소개한 SOLID(**S**ingle responsibility: 단일 책임, **O**pen/closed: 개방/폐쇄, **L**iskov substitution: 리스코프 치환, **I**nterface segregation: 인터페이스 분리, **D**ependency inversion: 의존성 주입) 원칙은 보다 우수한 객체 지향 소프트웨어 작성을 위한 지침을 제시합니다. 이 프레임워크는 상호 보완적인 원칙들로 구성되어 있는데, 이는 일반적이고 해석에 있어서 개방적인 반면 더 우수한 객체 지향 설계를 작성하는 데 충분한 지침을 제공합니다. SOLID 원칙을 적용하면 더욱 향상된 품질의 속성을 보유하고 더 오랫동안 유지할 수 있는 시스템을 구축할 수 있습니다.

SOLID 원칙은 객체 지향 설계를 추론하기 위해 클래스, 인터페이스, 상속 등과 같은 객체 지향 프리미티브(Primitive)와 개념을 사용합니다. 더불어, 주요 프리미티브가 클래스 보다는 컨테이너 이미지인 클라우드 네이티브 애플리케이션을 설계하기 위한 원칙들이 있습니다. 이 원칙들을 따르면, Kubernetes와 같은 클라우드 네이티브 플랫폼에 보다 적합한 컨테이너화된 애플리케이션을 개발할 수 있습니다.

클라우드 네이티브 컨테이너에 대한 RED HAT 접근 방식

현재 거의 모든 애플리케이션을 컨테이너화하고 실행하는 것이 가능합니다. 하지만, Kubernetes와 같은 클라우드 네이티브 플랫폼에 의해 효과적으로 자동화되고 오케스트레이션될 수 있는 컨테이너화된 애플리케이션을 개발하기 위해서는 추가적인 작업을 수행해야 합니다.

아래의 아이디어는 소스 코드 관리에서 애플리케이션 확장성 모델에 이르는 "Twelve-Factor App" 등과 같은 다른 많은 작업들에서 영감을 받았습니다. 하지만 다음 원칙들의 범위는 Kubernetes와 같은 클라우드 네이티브 플랫폼을 위한 컨테이너화된 마이크로서비스 기반 애플리케이션의 설계로 제한됩니다.

다음에 나열된 컨테이너화된 애플리케이션 개발 원칙들은 컨테이너 이미지를 기본 프리미티브로 사용하고 목표 실행 시간 환경으로서 컨테이너 오케스트레이션 플랫폼을 사용합니다. 이 원칙들을 준수함으로써 최종 컨테이너들은 대부분의 컨테이너 오케스트레이션 엔진에서 우수한 클라우드 네이티브 구성 요소로서 작동하도록 보장되며 자동화된 방식으로 스케줄링되고, 확장되며, 관리됩니다. 아래에서 이 원칙들을 더 자세히 소개해 드립니다. (나열 순서 없음)

단일 관심사 원칙(SCP, SINGLE CONCERN PRINCIPLE)

많은 측면에서, 이 원칙들은 하나의 클래스가 단 하나의 책임을 가져야 한다고 조언하는 SOLID의 SRP(Single Responsibility Principle, 단일 책임 원칙)와 유사합니다. SRP의 핵심은 각 책임이 변화의 축이며 하나의 클래스는 변화해야 하는 단 하나의 이유만을 가져야 한다는 것입니다. SCP 원칙에서 "관심사(concern)"는 책임보다 한 단계 더 높은 추상적인 관심사를 뜻하며 이는 클래스와는 대조적으로

컨테이너로서의 범위를 보다 정확하게 설명합니다. SRP의 핵심이 변화를 위한 단 하나의 이유만을 가지는 것이라면, SCP의 핵심은 컨테이너 이미지의 재사용과 교체 가능성입니다. 기능 구현을 완료하는(feature-complete) 방식으로 단일 관심사를 해결하는 컨테이너를 생성한다면, 다양한 애플리케이션 맥락에서 보다 높은 수준의 컨테이너 이미지를 재사용할 수 있습니다.

따라서 SCP 원칙은 모든 컨테이너가 단일 관심사를 효과적으로 처리해야 한다고 명시하고 있습니다. 컨테이너는 일반적으로 단일 프로세스를 관리하고, 대부분 단일 프로세스는 단일 관심사만을 해결하기 때문에 객체 지향 환경에서 SRP를 달성하는 것보다 SCP 원칙을 달성하는 것이 훨씬 간단합니다.

그림 1. 하나의 배포 단위 내 다수의 단일 관심사(Single Concern) 컨테이너들

컨테이너화된 마이크로서비스가 여러 관심사를 해결해야 한다면, sidecar와 init-container와 같은 패턴을 이용해 여러 컨테이너를 단일 배포 단위(pod)로 통합할 수 있으며 여기에서 각 컨테이너는 여전히 단일 관심사를 처리합니다. 마찬가지로 동일한 관심사를 처리하는 컨테이너를 교환할 수도 있습니다. 예를 들어, 웹 서버 컨테이너나 큐 구현 컨테이너를 보다 확장성이 뛰어난 새로운 컨테이너로 교체할 수 있습니다.

고 관측 원칙 (HOP, HIGH OBSERVABILITY PRINCIPLE)

컨테이너는 애플리케이션을 블랙 박스처럼 취급함으로써 이를 패키징하고 실행하는 통일된 방법을 제공합니다. 클라우드 네이티브 구성 요소가 되려는 컨테이너는 실행 시간 환경을 위한 API(Application Programming Interfaces)를 제공해 컨테이너 상태를 관찰하고 이에 따라 조치를 취할 수 있도록 해야 합니다. 이는 통일된 방식으로 컨테이너 업데이트와 라이프사이클을 자동화하는 기본 전제 조건일 뿐만 아니라, 시스템의 복원력과 사용자 경험을 향상시킵니다.

그림 2. 다수의 관측 API를 포함한 하나의 컨테이너 (컨테이너들은 관측을 하기 위해 다수의 API를 보유함)

현실적으로 컨테이너화된 애플리케이션은 최소한의 수준으로 활동(liveness)과 준비 상태(readiness) 와 같은 다양한 유형의 상태 확인을 위한 API를 제공해야 합니다. 보다 효과적으로 실행되는 애플리케이션들도 컨테이너화된 애플리케이션의 상태를 관찰하기 위한 수단을 제공해야 합니다. 애플리케이션은 Fluentd와 Logstash 등과 같은 툴로 로그 집계를 위해 STDERR(standard error)와 STDOUT(standard output)에 주요 이벤트를 로깅해야 하며 OpenTracing, Prometheus 등의 추적 및 측정 지표 수집 라이브러리를 통합해야 합니다.

애플리케이션을 블랙 박스처럼 다루는 한편, 플랫폼이 가능한 최적의 방식으로 애플리케이션을 관찰하고 관리할 수 있도록 모든 필수 API를 구현해야 합니다.

라이프사이클 준수 원칙(LCP)

HOP는 컨테이너가 플랫폼의 읽기 작업을 위한 API를 제공하도록 요구합니다. LCP는 애플리케이션이 플랫폼에서 입력되는 이벤트를 읽는 방법을 갖도록 요구합니다. 뿐만 아니라, 컨테이너는 이벤트를 가져오는 것 이외에도 이 이벤트들을 따르고 대응해야 합니다. 이 원칙의 명칭이 바로 여기에서 유래되었습니다. 이는 플랫폼과 상호 작용하기 위해 애플리케이션에서 "API를 작성" 하는 것과 유사합니다.

그림 3. API를 제공하고 플랫폼 이벤트를 수신하는 컨테이너

여러분이 컨테이너의 라이프사이클을 관리할 수 있도록 돕는 관리 플랫폼은 모든 유형의 이벤트들을 제공합니다. 어떤 이벤트를 처리할 것인지, 그리고 해당 이벤트들에 대응을 할 것인지 여부는 애플리케이션에 의해 결정됩니다.

이벤트마다 그 중요도는 다릅니다. 예를 들면, 클린 셧다운(clean shutdown) 프로세스를 필요로 하는 모든 애플리케이션은 신호를 포착하고 메시지를 종료하며(SIGTERM) 가능한 한 빨리 셧다운합니다. 이는 신호를 통해 강제 셧다운을 피하기 위함입니다. 즉, SIGTERM에 따른 kill(SIGKILL)을 뜻합니다.

PostStart, PreStop과 같은 이벤트들은 여러분이 애플리케이션 라이프사이클을 관리할 때에 중요한 경우가 있습니다. 예를 들어, 일부 애플리케이션들은 서비스 요청 이전에 준비되어야 하며 일부는 셧다운되기 전에 리소스를 릴리즈해야 합니다.

이미지 불변성 원칙(IIP, IMAGE IMMUTABILITY PRINCIPLE)

컨테이너화된 애플리케이션은 변경 불가능하다는 것을 의미하며 한번 구축되면 다른 환경에서 변경되지 않습니다. 이는 환경별로 컨테이너를 개발하고 수정하는 것이 아니라 환경에 따라 달라지는 외부화된 설정에 의존하는 외부 수단과 런타임 데이터를 저장하는 외부 수단을 사용 하는 것을 의미합니다. 컨테이너화된 애플리케이션의 모든 변경은 새로운 컨테이너 이미지를 구축하고 이를 모든 환경에서 재사용하는 결과로 이어져야 합니다. 동일한 원리가 불변의 서버/인프라(immutable server/infrastructure)에서도 많이 사용되고 있으며 서버/호스트 관리에도 사용됩니다.

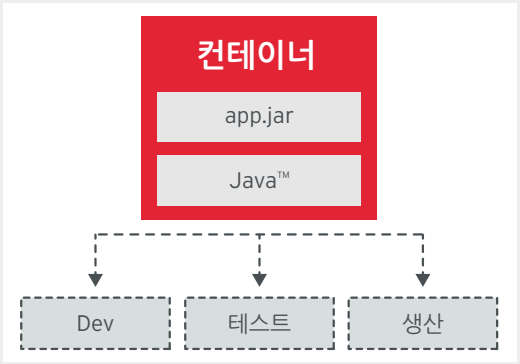

그림 4. 변경할 수 없는 컨테이너 이미지가 모든 환경 전반에서 사용됨

IIP 원칙에 따라 다양한 환경에서 유사한 컨테이너 이미지를 생성하는 것을 방지하면서, 각 환경을 위해 설정된 하나의 컨테이너 이미지를 고수해야 합니다. 이 원칙은 애플리케이션 업데이트 중 자동 롤백(roll-back)과 롤 포워드(roll-forward) 등과 같은 작업을 지원하며 이는 클라우드 네이티브 자동화의 중요한 측면입니다.

프로세스 폐기 가능성 원칙 (PDP, PROCESS DISPOSABILITY PRINCIPLE)

컨테이너화된 애플리케이션으로 전환해야 하는 주요 원인 중 하나는 컨테이너가 가능한 한 일회적이어야 하며 원하는 모든 시점에 다른 컨테이너 인스턴스로 즉시 교체될 수 있어야 한다는 것입니다. 상태 체크 실패, 애플리케이션 축소, 컨테이너를 다른 호스트로 마이그레이션, 플랫폼 리소스 부족 또는 다른 이슈 등 컨테이너를 교체해야 하는 많은 이유들이 있습니다.

그림 5. 손쉬운 교체를 위한 빠른 스타트업과 셧다운을 지원하는 컨테이너화된 애플리케이션

이는 컨테이너화된 애플리케이션이 그 상태를 외부 또는 분산 그리고 중복으로 유지해야 한다는 것을 의미합니다. 또한 애플리케이션을 빠르게 스타트업하고 셧다운할 수 있어야 하며, 갑작스러운 전체 하드웨어 장애에도 대비해야 합니다.

이 원칙을 구현하는 또 다른 유용한 방법은 소형 컨테이너를 만드는 것입니다. 클라우드 네이티브 환경의 컨테이너는 여러 다른 호스트에서 자동으로 스케줄링되고 시작될 수 있습니다. 재시작 전에 컨테이너가 호스트 시스템으로 물리적으로 복사되기 때문에 소형 컨테이너를 통해 시동 시간을 단축할 수 있습니다.

독립성 원칙(S-CP)

이 원칙은 컨테이너가 구축 시간(빌드 타임)에 필요한 모든 것을 포함하고 있어야 한다는 것입니다. 컨테이너는 Linux® 커널만을 의존해야하며 컨테이너가 구축될 때 다른 라이브러리가 추가됩니다. 라이브러리 이외에도, 실행 환경, 요구되는 애플리케이션 플랫폼, 그리고 컨테이너화된 애플리케이션을 실행하는 데 필요한 여러 종속 요소 등을 포함해야 합니다.

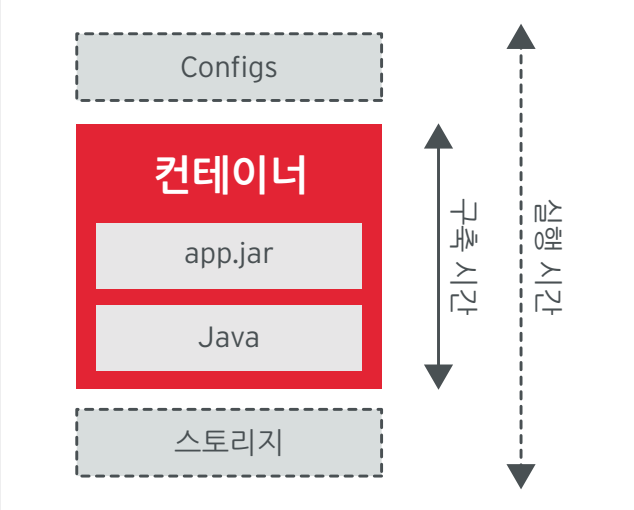

그림 6. 특정 환경에 속한 경우를 제외하면 컨테이너는 구축 시간에 모든 종속 항목 내에 포함됨

환경에 따라 다를 수 있으며 런타임에 제공되어야 하는 설정 등이 유일한 예외입니다 (예를 들면, Kubernetes ConfigMap을 통한 설정).

일부 애플리케이션은 여러 개의 컨테이너화된 구성 요소들로 구성되어 있습니다. 예를 들어 컨테이너화된 웹 애플리케이션은 데이터베이스 컨테이너도 필요로 할 수 있습니다. 이 원칙은 컨테이너 두개를 합치는 것을 권장하지 않습니다. 그 대신, 데이터베이스 컨테이너는 데이터베이스를 실행하는 데 필요한 모든 것을 포함하도록 권장하며, 웹 애플리케이션 컨테이너는 웹 서버와 같이 웹 애플리케이션을 실행하는 데 필요한 모든 것을 포함할 것을 제안합니다. 런타임동안 웹 애플리케이션 컨테이너는 필요에 따라 데이터베이스 컨테이너를 이용하고 의존합니다.

런타임 견제 원칙(RCP, RUNTIME CONTAINMENT PRINCIPLE)

S-CP는 빌드 타임 관점과 해당 컨텐츠가 있는 최종 바이너리 위주로 컨테이너를 인식합니다. 하지만 컨테이너를 디스크에 존재하는 일정한 크기의 평면적인 블랙 박스라고만 인식할 수는 없습니다. 컨테이너에는 메모리 및 CPU 사용 규격(Dimension)과 여타 리소스의 소모 규격 등 여러가지 규격들이 존재합니다.

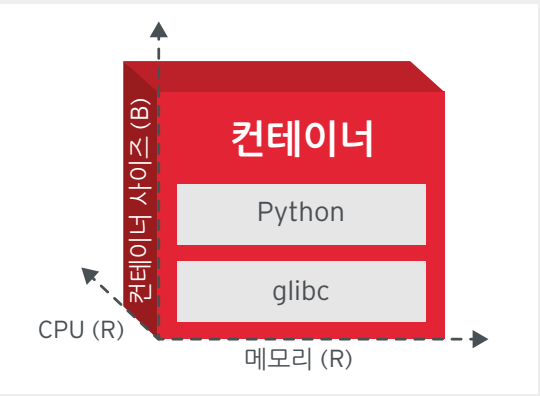

그림 7. 런타임 리소스 요구 사항을 선언하고 런타임에 이를 적용(Respect)한 컨테이너

이 RCP 원칙은 각 컨테이너가 해당 리소스 요구 사항을 선언하고 정보를 플랫폼으로 전달할 것을 제안합니다. 또한, 플랫폼은 컨테이너의 스케줄링, 자동 확장, 용량 관리, 일반 SLA(서비스 수준 계약) 등을 실행하는 방식에 대해 CPU, 메모리, 네트워킹, 디스크 등이 미치는 영향 측면에서 컨테이너의 리소스 프로필을 공유해야 합니다.

컨테이너의 리소스 요구 사항을 전달하는 것 이외에도, 애플리케이션이 지정된 리소스 요구 사항을 준수하도록 하는 것이 중요합니다. 애플리케이션이 이를 따른다면, 플랫폼은 리소스 부족 문제가 발생했을 때 종료와 마이그레이션을 고려하지 않아도 됩니다.

결론

클라우드 네이티브는 최종 상태가 아니라 운영 방식입니다. 이 백서는 컨테이너화된 애플리케이션이 클라우드 네이티브의 바람직한 구성 요소가 되기 위해서 반드시 준수해야 하는 여러 가지의 원칙들을 다루었습니다.

컨테이너화된 우수한 애플리케이션을 개발하기 위해서는 이 원칙들 이외에도 다양한 컨테이너 관련 모범 사례와 기술들을 많이 접해야 합니다. 이 백서에서 설명한 원칙들은 매우 기본적이며 대부분의 사용 사례에 적용할 수 있습니다. 그러나 다음에 나열된 모범 사례들에 대해서는 언제 적용하는 것이 적합한지 또는 아예 적용하지 않아야 하는지를 판단해야 합니다. 컨테이너 관련 모범 사례들은 다음과 같습니다.

- **작은 이미지 구축:** 불필요한 패키지의 설치를 막고 임시 파일을 정리하여 작은 이미지를 생성합니다. 이는 컨테이너 이미지를 복사할 때 컨테이너의 크기, 빌드 타임, 그리고 네트워킹 시간을 줄입니다.
- **임의의 사용자 ID 지원:** sudo 명령어를 사용하거나 컨테이너를 실행하기 위한 특정 userid를 요구하는 것을 막습니다.
- **주요 포트 표시:** 런타임에 포트 번호를 지정하는 것이 가능한 동시에, EXPOSE 명령어를 사용하여 이를 특정화함으로써 사람과 소프트웨어 모두 이미지를 보다 쉽게 이용할 수 있습니다.
- **퍼시스턴트 데이터를 위한 볼륨 사용:** 컨테이너 제거 후 보존해야 하는 데이터는 볼륨에 작성해야 합니다.
- **이미지 메타데이터 설정:** 태그, 라벨, 주석 등의 형태로 작성된 이미지 메타데이터는 컨테이너 이미지를 보다 효율적으로 사용할 수 있도록 하므로 이미지를 사용하는 개발자들의 경험을 향상시킵니다.
- **호스트와 이미지 동기화:** 일부 컨테이너화된 애플리케이션들은 컨테이너가 시간과 머신 ID와 같은 특정한 속성을 기준으로 호스트와 동기화되어야 한다고 요구합니다.

다음의 참고 링크에서는 이 백서의 원칙들을 더욱 효과적으로 구현할 수 있도록 돕는 모범 사례들과 패턴을 제공합니다.

- https://www.slideshare.net/luebken/container-patterns
- https://docs.docker.com/engine/userguide/eng-image/dockerfile_best-practices
- http://docs.projectatomic.io/container-best-practices
- https://docs.openshift.com/enterprise/3.0/creating_images/guidelines.html
- https://www.usenix.org/system/files/conference/hotcloud16/hotcloud16_burns.pdf
- https://leanpub.com/k8spatterns/
- https://12factor.net/

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

www.redhat.com/ko
#f8808_1017