이벤트 기반 마이크로서비스

응답성, 유연성, 탄력성의 마이크로서비스 아키텍처를 구축하기 위한 개발자용 가이드

solace.

“

마이크로서비스를 통해 민첩성을 기하급수적으로 향상하기 위해서는 우리의 정적이고 순차적이며 단일구조적인 사고를 바꾸어야 한다

조나단 샤보우스키

솔라스
캐나다 온타리오 주 오타와시 535 레겟 드라이브 3 층

전화: +1 613-271-1010
웹사이트: solace.com

목 차

solace.

머리말

많은 기업들이 마이크로서비스 아키텍처로 전환하는 이유는 한 가지 바로, 민첩성 때문입니다. 클릭 한 번이면 경쟁업체들을 찾을 수 있고 제품을 개발해서 출시하기까지 걸리는 시간이 모든 것을 결정하게 된 세상에서 기업의 수익을 지킬 수 있는 방식으로 신속하게 컴포넌트들을 만들고 수정하는 능력은 무엇보다 중요한 요소입니다. 그러나 컴포넌트들을 개발하는 속도는 퍼즐 조각들 중의 하나일 뿐입니다.

당신은 얼마나 빠르게 그 컴포넌트들을 시스템의 다른 부분들과 통합할 수 있습니까? 당신은 얼마나 완전하게 혁신적인 신기법과 신기술을 포용할 수 있습니까?

마이크로서비스 계획을 성공시키기 위해서는 민첩성이 지속적 통합/지속적 배포(CI/CD), 자동화된 인프라, 데브옵스(DevOps) 및 애자일 방식 개발과 같은 실제적 방법들을 활용하는 전체론적인 접근법에 달려 있다는 것을 당신의 회사가 반드시 알고 있어야 합니다. 일반적인 마이크로서비스 개발자는 이런 컨셉들을 간소하고 일반적인 개발 머신으로 결합합니다. 뛰어난 마이크로서비스 개발자는 고도 분산시스템들에 수반하는 부정할 수 없는 (그러나 예측가능한) 문제들을 이해하고 극복할 수 있는 방법을 찾아냄으로써 앞으로 더 나아갑니다.

마이크로서비스 계획을 확실히 성공시키기 위해서는 민첩성에 대한 중대한 위험요인들과 장애물들을 이해하는 것이 가장 중요합니다.

- 분산 프로세싱의 불안정성
- 에코시스템 통합의 문제
- 서비스 조화의 부족

만약 우리가 분산시스템의 현실을 인정하지 않는다면, 지난 2000 년대에 겪었던 서비스지향 아키텍처(SOA)의 과대광고와 같은 과거의 실패를 되풀이하게 될 것입니다. 다행이도 이러한 현실을 극복할 수 있는 매우 효과적인 방법이 하나 있다. 이벤트 기반(반응형이라고 부르기도 하는) 아키텍처가 그것입니다.

솔라스 기술담당최고책임자실의 수석개발자인 조나단 샤보우스키는 애플리케이션 분할이 여러 면에서 유익하지만 삶을 좀 더 복잡하게 만들 수 있다는 사실부터 이벤트 기반 아키텍처와 마이크로서비스를 결합하여 얻게 되는 커다란 이점까지 설명할 것입니다.

“디지털 전환 계획을 이끄는 애플리케이션 리더들은 자신들의 기술적, 조직적 그리고 문화적 전략들에 ‘이벤트적 사고’를 더해야 한다.

에핌 나티스
가트너

출처: 가트너 “디지털 비즈니스에 있어서 비즈니스 이벤트, 비즈니스 모멘트 그리고 이벤트적 사고” 에핌 나티스, 2017. 8. 4

solace.

서비스 분할 패러독스

마이크로서비스의 이론은 간단합니다. 모노리식 애플리케이션들을 작고 특정목적에 맞는 마이크로서비스들로 분할하여 더 나은 민첩성과 확장성 그리고 재사용성을 얻는 것입니다.

그러나, 현실적으로 마이크로서비스를 구현하는 것은, 항상 그렇듯, 그보다는 좀 더 복잡합니다. 그 이유로는 분산 컴퓨팅의 오류가 큰 부분을 차지하고 있는데 이것은 프로그래머들과 개발자들이 분산 애플리케이션의 세계에 들어갈 때 내리는 일련의 잘못된 추정들이다. 아래 목록은 1994 년 선마이크로시스템스의 엘. 피터 도이치와 그의 동료들이 만든 것인데 지금도 여전히 유효합니다.

분산 컴퓨팅의 오류

- **네트워크는 신뢰할 수 있습니다.**
- **지연속도가 없습니다.**
- 대역폭은 무한합니다.
- **네트워크는 안전합니다.**
- 망 형태는 변하지 않습니다.
- 관리자는 1 명입니다.
- **전송 비용이 들지 않습니다.**
- **네트워크는 동질입니다.**

이러한 오류들이 모두 마이크로서비스 세계와 관련되어 있지만 그 중에서도 **굵은 활자체**로 된 항목들은 특히 중요한 것들입니다.

당신이 각 마이크로서비스를 더 작게 만들수록 서비스 횟수는 더 늘어나게 되고, 분산 컴퓨팅의 오류가 안정성과 사용자 경험/시스템 성능에 더 큰 영향을 주게 됩니다. 이로 인해 네트워크와 서비스 마비 상태를 처리하는 동안 지연속도를 최소화하는 아키텍처를 구축하고 실행하는 것이 무엇보다 중요한 요소가 됩니다.

마이크로서비스와 연관된 대부분의 툴링에는 지속적 통합-지속적 배포(CI/CD), 자동화된 인프라, 데브옵스(DevOps) 그리고 애자일 소프트웨어 개발을 사용해야 합니다. 그 좋은 예로 피보탈(Pivotal)의 클라우드 파운드리(Cloud Foundry)를 들 수 있는데, 이 플랫폼으로 개발자들은 현대적이고 클라우드 환경에 최적화된 기술을 사용하여 쉽게 마이크로서비스를 개발, 테스트, 배치, 업데이트 할 수 있습니다.

문제는 마이크로서비스가 그 기능을 수행하고 기업가치를 만들기 위해서는 접속성/데이터가 필요한데 그동안 데이터 수집/통신이 매우 소홀히 여겨져 와서 툴링이 뒤쳐져 있을 정도라는 것입니다. 예를 들어, API 관리/게이트웨이 제품들은 오로지 동기적 요청/응답 교환 패턴만을 지원하는데 이것이 분산 컴퓨팅의 문제들을 더욱 악화시킵니다. 또한 이 제품들은 레거시 시스템에 있는 데이터들을 통합하거나 수집할 수 없습니다.

그와 동시에 이벤팅/메시징 툴 역시 시대에 뒤진 민첩하지 않은 세상에 갇혀 있어서 데브옵스(DevOps) 그리고 셀프서비스와 같은 마이크로서비스의 많은 기본원리들과 호환되지 않습니다. 그러나 분산 컴퓨팅의 특이성들을 가장 잘 다룰 수 있는 것이 이벤팅/메시징이며 또한 그것이 마이크로서비스 아키텍처의 잠재력을 실현시켜줄 열쇠입니다.

"이벤팅/메시징 툴은 시대에 뒤진 민첩하지 않은 세상에 갇혀 있습니다. 많은 툴들이 데브옵스와 같은 마이크로서비스의 기본원리들과 호환되지 않습니다."

TWEET

서비스들이 더 작아지고 용도가 단일화되면서 재사용도의 가능성이 커지고 있지만 이는 서비스들이 협업할 수 있는 능력에 달려있습니다.

서비스지향 아키텍처(SOA)의 시대에는 유즈케이스의 모든 패싯들을 BPEL 엔진이나 ESB 를 이용, 한 개의 서비스 패키지로 결합하여 직접적으로 구현한 커다란 모노리식 서비스를 개발할 수 있었습니다. 하지만 이것은 재이용할 수 없었고 확장하기도 어려웠는데 그 이유는 ESB/BPEL 의 구성이 네트워크에 너무 논리적으로 이동되어 "덤 엔드포인트-스마트 파이프"가 되어 버리기 때문이었고 이는 고비용이고 복잡하며 고장수리가 거의 불가능했습니다.

이제 우리는 가야할 방향이 작고, 단일목적의 마이크로서비스이며 또한 접속성과 커뮤니케이션에 "스마트 엔드포인트, 덤 파이프" 접근법을 사용해야 한다는 것을 알고 있습니다. 그러나 여전히 해결되지 않은 질문들이 남아 있습니다.

어떻게 이미 실패한 모노리식 또는 오케스트레이션 기법으로 다시 후퇴하지 않고 기업가치를 만들어 내는 서비스 협업을 가능하게 할 것인가?

알고 계셨나요?

로직모니터의 클라우드 비전 2020 설문조사에 따르면 2020 년에는 기업 업무량의 83%가 클라우드에 있을 것입니다

solace.

통합의 난제

모든 마이크로서비스는 프로세스용 데이터가 필요한데 12 팩터 앱은 무상태이기 때문에 어디에서 데이터를 가져와야 합니다. 그린필드 시스템에 필요한 데이터 수집은 쉽지만, 그러나 마이크로서비스는 거의 언제나 디지털 전환과 현대화 또는 더욱 빠른 속도로 새로운 캐퍼빌리티를 구축해야 할 필요에 따르는 부작용과 같이 됩니다. 그래서 당신이 거의 항상 레거시 시스템의 에코시스템을 다루고 있는 것인데 그 중 일부는 현대화될 것이고 나머지는 가까운 시기동안 지금과 같이 남아있을 것입니다.

기존의 비즈니스 에코시스템은 대부분의 비즈니스가 마이크로서비스로의 여정을 시작할 때 반드시 대처해야 하는 피할 수 없는 부담입니다. 마이크로서비스가 프라이빗 클라우드와 퍼블릭 클라우드를 기반으로 하는데 반해 대부분의 기존 시스템은 온프레미스를 기반으로 합니다. 종종 불안정하고 예측할 수 없는 광역통신망(WAN)으로 데이터를 전송하도록 하는 것은 까다롭고 시간이 많이 걸리는 일입니다. 또한 그 후에는 사물인터넷(IoT)과 모바일 디바이스 그리고 빅데이터가 초래하는 새롭고 특화된 요인들도 감안해야 합니다. 이 시스템의 크기와 다양성으로 인해 마이크로서비스 계획에 매우 많은 선결 문제들이 생깁니다.

이것들을 모두 종합해보면 임피던스 불일치들을 우리 주변에서 어렵지 않게 찾아볼 수 있습니다.

1. 레거시 시스템으로의 업데이트는 느린데 마이크로서비스는 빠르고 민첩해야 합니다.
2. 레거시 시스템은 오래된 통신 매질을 사용하지만 마이크로서비스는 최신의 오픈 프로토콜과 API 를 사용합니다.
3. 레거시 시스템은 거의 언제나 온프레미스 기반이고 잘해야 가상화를 이용하지만 마이크로서비스는 클라우드와 IaaS 추상화를 기반으로 합니다.
4. 사물인터넷(IoT) 시스템은 고도로 특화된 프로토콜을 사용하지만 대부분의 마이크로서비스 API 와 프레임워크는 본질적으로 그것을 지원하지 않습니다. 또한,
5. 모바일 디바이스는 REST 를 이용할 수도 있지만 또한 비동기식 커뮤니케이션을 필요로 하는데 비해 대부분의 API 게이트웨이는 오로지 동기식 RESTful 인터렉션만을 지원합니다.

어떻게 한 조직이 이러한 불일치들을 모두 해결할 수 있을까요?

놀라운 무용수 마이크로서비스

앞서 설명한 것처럼 서비스가 작아질수록 각 서비스가 엔드유저에게 별개로 제공하는 가치도 작아집니다-가치는 서비스의 오케스트레이션으로 만들어진다. 과거에는 이 오케스트레이션을 BPEL 엔진 또는 ESB 와 같은 센트럴 컴포넌트로 실행하였고, 오늘날에는 API 게이트웨이를 이용합니다.

나는 이와 같은 접근법이 갖는 문제점을 음악적 비유로 설명하고자 합니다.

대부분의 작곡가들은 시행착오를 거쳐 곡을 완성한다(소프트웨어 개발자와 똑같습니다). 그들의 산출물? 각각의 악기들이 연주할 악보들을 담고 있는 하나의 악보입니다. 이 비유에서 각 뮤지션들은 마이크로서비스와 같습니다. 만약 작곡가가 헤비메탈 밴드의 곡을 쓴다면 그 곡에는 단지 몇 개의 악기들(기타, 베이스, 드럼, 보컬리스트) 밖에 없을 것이고 지휘자는 필요하지 않을 것입니다.

그러나 심포니 오케스트라는 다양한 악기들을 연주하는 백여명의 뮤지션들로 구성되어 있습니다. 이 경우, 모든 뮤지션들이 제때에 연주를 시작하고, 박자를 유지하고, 필요할 때는 더 빠르거나 느리게 연주하기 위해서, 또는 더 크게 연주하거나 부드럽게 연주하기 위해서 절대적으로 지휘자가 필요하게 됩니다.

만약 불행히도 지휘자가 뮤지션들과 커뮤니케이션 할 수 없다면 – 혹은 단지 한 섹션의 뮤지션들이나 단 한 명의 뮤지션과 의사소통을 할 수 없게 되더라도 – 상황이 빠르게 나빠져서 연주를 완전히 망치지는 않는다 하더라도 상처를 입게 될 것입니다.

춤은 다릅니다. 안무가는 음악을 듣고 그 안에 들어있는 이벤트들을 바탕으로 춤의 루틴을 만듭니다. 무용수들은 서로 완전히 다른 몸동작이나 스텝을 할 수 있지만, 그러나 그것이 오디오 큐(혹은 이벤트)에 맞추어 미리 안무 되어 있는 것이라면 그 춤의 루틴은 성공한 것이 됩니다. 심지어 누군가 스텝을 잘못 밟거나 박자를 놓쳐 망치게 되더라도 쇼는 계속될 수 있는데 그 이유는 각 무용수들이 편곡자로부터 무엇을 할 것인지 지시를 받는 것이 아니라 음악을 들으면서 자신의 특정한 이벤트를 기다리기 때문입니다.

마이크로서비스로 되돌아가서 말하자면, 하나의 마이크로서비스는 마이크로-오케스트레이션의 예라고 할 수 있는 코드 안에서 일련의 스텝들을 수행합니다. 마이크로서비스의 입력물 혹은 출력물은 도메인 시그니피컨스를 가지고 있는 데이터 이벤트입니다. 마이크로서비스는 단지 이벤트를 생산만 하기 때문에 그것을 프로세스 해야 할지 혹은 언제 해야 할지 알지 못한다는 것이 핵심입니다. 다른 서비스들은 하나의 이벤트 또는 한 묶음의 이벤트에 대한 인터레스트를 등록하고 그에 맞게 대응합니다. 마치 루틴의 스텝을 수행하는 무용수와 같습니다.
아이러니하게도 마이크로서비스의 실행과 춤 동작 사이의 공통된 신호 테마는 이벤트(데이터 이벤트 또는 오디오 이벤트)입니다.

이벤트 중심으로의 대전환

데이터 중심에서 이벤트 중심으로 IT 우선순위 – 코페르니쿠스적 대전환

> **"코페르니쿠스 이전 천문학자들의 착각과 비슷하게, 많은 개발자들과 공학자들이 데이터가 컴퓨팅 우주의 중심이라는 생각에 사로잡혀 있다."**
>
> TWEET

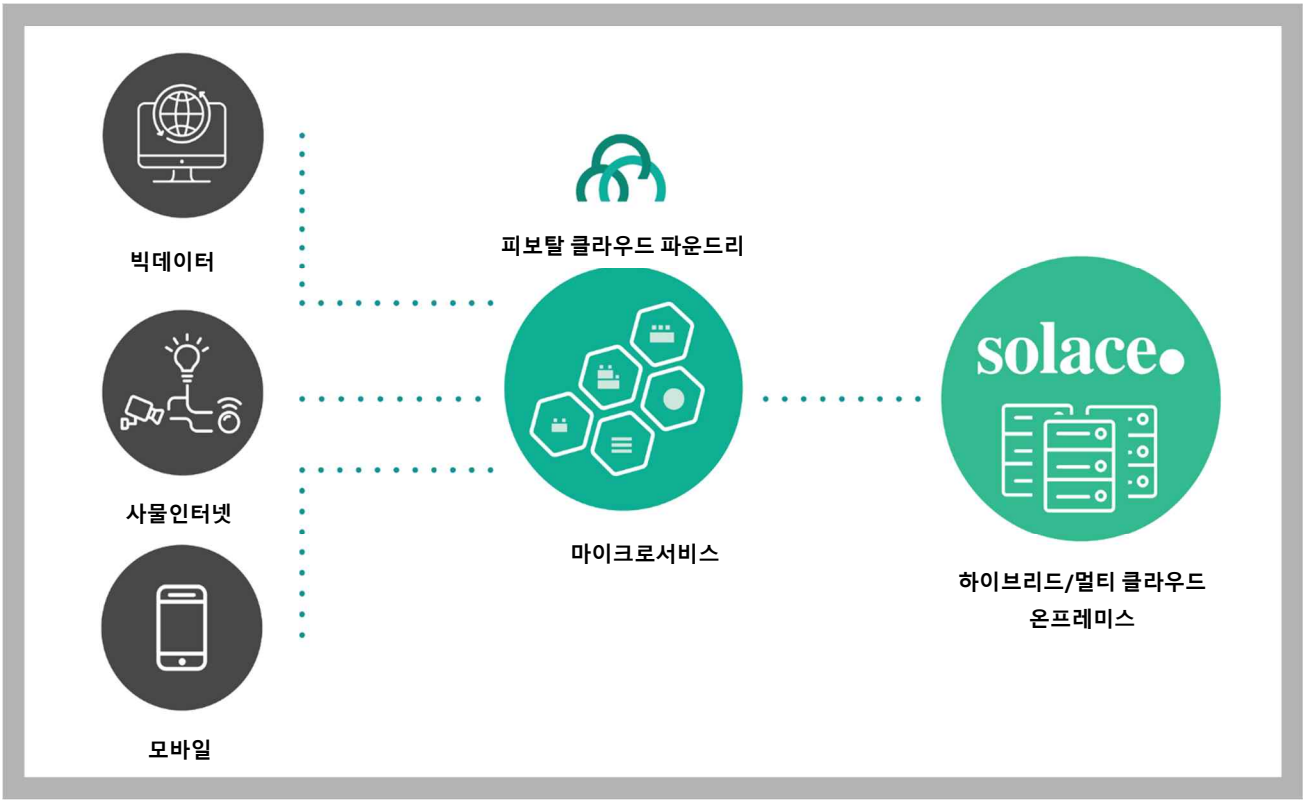
코페르니쿠스는 르네상스 시대의 수학자이자 천문학자로 지구가 아닌 태양이 우주의 중심에 있는 우주모델을 정립하였다. 당시로서는 획기적이었습니다.

코페르니쿠스 이전 천문학자들의 착각과 비슷하게, 많은 개발자들과 공학자들이 데이터가 컴퓨팅 우주의 중심이라는 생각에 사로잡혀 있습니다.

이러한 전통적인 관점은 데이터가 가장 우선이며 또한 데이터가 일단 보관되면 커맨드 스타일의 요청/응답 인터렉션을 통해 데이터를 검색하고, 업데이트하고, 삭제할 수 있다는 믿음에 근거한 것입니다.

이런 데이터에 대한 극단적인 집중의 이유는 간단하다. 데이터베이스와의 모든 인터렉션이 커맨드 스타일의 인터렉션으로 실행되는 것입니다! 이 관점의 문제는 기업들이 데이터베이스를 중심으로 한 많은 마이크로서비스들을 만들어 내게 되고 궁극적으로 데이터를 빠르고 유연하게 공유할 수 없는 많은 순차적 애플리케이션 엔클레이브가 만들어지게 되는 것입니다. 다른 말로 하자면, 똑같이 민첩성 부족으로 실패할 운명의 다른 종류의 모노리식 애플리케이션을 만드는데 돈을 쓰는 것입니다.

그러면 마이크로서비스 우주의 중심에는 무엇이 있어야 할까요? 쉽습니다. 이벤트입니다.



마이크로서비스를 통해 민첩성을 기하급수적으로 향상시키기 위해서는 우리의 정적이고 순차적이며 단일구조적인 사고를 바꾸어 알맞은 이벤트를 알맞은 때에 알맞은 서비스로 전달하려고 해야 합니다.

당신의 몸을 생각해 보라. 우리는 촉각, 시각, 미각, 청각, 후각의 감각을 통해 전달되는 이벤트들에 대해 끊임없이 반응하고 행동합니다. 이 이벤트들은 우리에게 정보를 전달하는데 이 정보는 우리의 기억속에 저장되어 있다가 생각으로 재생되고 그리고 필요할 때는 우주가 반응하는 새로운 이벤트들을 만들어내는 행동의 형태로 우리가 행동하게 합니다.

이벤트 기반 방식의 사고는 컴퓨팅 우주에서 조직들을 하나의 감각기관으로 바꾸어 놓습니다. 웹/모바일 애플리케이션들과 사물인터넷(IoT)의 센서들 또는 데이터 레거시 시스템이 새로운 이벤트들을 감지하고 이벤팅/메시징 플랫폼으로 이동시키면 이 플랫폼은 다시 이벤트들을 마이크로서비스 플랫폼으로 분배합니다.

마치 우리의 감각체계가 이벤트들을 중추신경계에 전달해 해석하도록 하는 것과 똑같습니다. 마이크로서비스의 세계에서 우리는 다음과 같이 할 수 있는 중추신경계가 필요합니다.

- 다른 종류의 자극들을 잘 수용합니다.
- 전송이 빠르고, 든든히 믿을 수 있습니다. 또한,
- 이벤트를 수용할 때 변화에 잘 적응합니다.

이벤트 기반의 사고를 적용하지 않는다면 비용 증가와 생산성 저하로 인해 디지털 전환과 마이크로서비스 계획을 성공시키는데 걸림돌이 될 것입니다. 그렇다면 당신과 당신의 조직이 IT 우주의 중심에 있는 이벤트들로 성공을 거둘 수 있는 패턴과 접근법은 무엇일까요?

현대적 중추신경계가 제공하는 민첩성의 실현

가트너가 2017 년 8 월 자사의 <*디지털 비즈니스에 있어서 비즈니스 이벤트, 비즈니스 모멘트 그리고 이벤트적 사고*>라는 제목의 보고서에서 밝히고 있듯이, "디지털 전환 계획을 이끄는 애플리케이션 리더들은 자신들의 기술적, 조직적 그리고 문화적 전략들에 '이벤트적 사고'를 더해야 합니다".

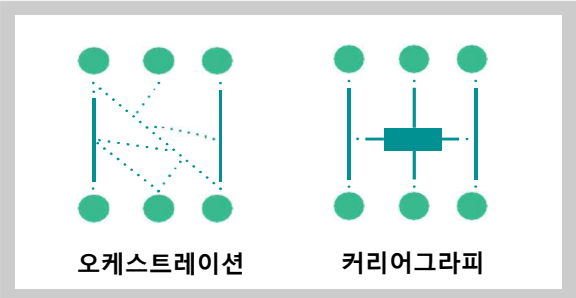
그것은 대담한 말이지만, 나는 그 말에 완전히 동의한다. 결국 이 글의 목적은 이벤트 기반 마이크로서비스의 민첩성을 실현시키기 위해서 당신이 꼭 거쳐야 하는 실행가능한 단계들을 보여주려는 것이다.

이벤트 기반을 생각하고 서비스 실행을 커리어그라프 하십시오

이벤트 기반의 사고방식을 적용하기 위한 첫번째 단계는 솔루션 디자인과 설계에 대한 당신의 사고방식을 바꾸는 것입니다. 보통 처음에는 서비스들 사이의 모든 인터렉션을 일련의 요청/응답 서비스 콜 시퀀스로 생각합니다. 실제로 만일 당신이나 당신의 팀이 "호출", "요청" 또는 "발신"과 같은 용어를 사용한다면 당신이 여전히 커맨드 스타일의 패러다임 안에서 사고하고 있다는 확실한 신호입니다.

대신에 다음과 같이 해보십시오. "당신의 서비스가 어떤 이벤트들을 프로세스 해야 할까요?" 그리고 "당신의 서비스가 어떤 이벤트들을 내보낼 것일까요?"

이벤트 기반의 사고방식을 적용하고 난 다음에는 오케스트레이션에서 커리어그라피로 전환해야 합니다.



흔히 개발자들은 "서비스가 A 가 서비스 B 에 발신하고 서비스 B 는 서비스 C 에 발신할 것"이라는 관점에서 생각하고 그 모델을 연쇄적 호출(a->b->c)을 통해서나 또는 x->a, x->b, x->c 의 차례대로 오케스트레이터 서비스를 만들어 구현합니다.

그 두가지 접근법들은 모두 분산 컴퓨팅의 현실과 마주할 때, 특히 당신이 스케일을 시작할 때에 혼란을 일으킵니다.

대안은 커리어그라피의 이론을 따르는 것입니다. 춤의 비유로 돌아가서 말하자면, 무용수가 음악적 신호에 반응하는 것처럼 서비스는 자신의 환경변화에 반응해야 합니다. 그에 따른 이점은 막대합니다.

- **향상된 민첩성.** 애자일 개발팀들은 더 독립적이고 다른 서비스의 변화들에 의한 영향을 현저하게 덜 받습니다.
- **서비스가 더 작고 간단합니다.** 각각의 서비스는 다운스트림 서비스 또는 네트워크 장애를 대처하기 위한 복잡한 에러처리가 없어도 됩니다.
- **줄어든 서비스 커플링.** 서비스는 다른 서비스들의 존재를 모릅니다.
- **정교한 스케일링을 가능하게 한다.** 각 서비스를 수요에 따라 독립적으로 스케일업 또는 스케일다운 할 수 있다. 이로써 좋은 사용자 경험을 유지하고 컴퓨터 리소스의 낭비를 줄인다.
- **새로운 서비스를 추가하기 쉽다.** 줄어든 커플링으로 인해 새로운 서비스가 온라인으로 제공되고 이벤트를 사용하며 다른 서비스를 변경하지 않고 새로운 기능을 실행한다.

위와 같이 많은 이점들을 아무런 대가 없이 얻을 수는 없다. 세상에 공짜로 주는 밥이란 없는 법이니까.

이제 상태의 정합성이 논의의 초점이 되는데 왜냐하면 서비스가 일시적으로 정지되었다는 것은 이벤트의 상태 변화가 즉시 프로세스 되지 않을 수도 있다는 것이기 때문이다. 근본적으로 어떻게 이런 부작용에 대처할 것인가?

궁극적인 정합성을 포용하십시오

궁극적인 정합성이란 정합성이 미래에 일어날 것이라는 생각이고, 어떤 것들이 한동안 동기화되지 않을 수도 있음을 인정한다는 의미입니다. 이 패턴과 컨셉으로 인해 개발자들이 값비싼 XA 트랜잭션을 믹스에서 제거하기도 합니다. 서비스가 적절히 처리하고 확인하기 전에는 도메인 변경 이벤트가 절대로 유실되지 않도록 하는 것은 이벤팅/메시징 플랫폼이 해야 할 일입니다.

어떤 사람들은 궁극적인 정합성의 유일한 이점이 성능이라고 생각하지만, 각기 다른 서비스는 그것이 관심을 갖는 이벤트들에 따라 동작할 뿐이기 때문에 마이크로서비스들의 커플링을 해체하는 것이 실제 장점입니다.

> 마이크로서비스로 가는 길은 좋은 의도로 가득 차 있습니다. 그러나 적지 않은 팀들이 무엇이 필요한지 먼저 분석하지도 않고 시류에 편승하고 있습니다.

나다니엘 티. 슈타

데이터베이스+CQRS

이벤트 사용에 대한 사고과정은 대개의 경우 흥미로운 질문으로 이어집니다. 데이터가 더 이상 내 우주의 중심이 아니라면, 이제 이벤트를 어디에 존속시켜야 할까요?

데이터베이스 때문에 우리의 생각은 다시 커맨드(생성, 읽기, 갱신, 삭제) 스타일의 인터렉션으로 되돌아갑니다. 이 데이터베이스에 관한 딜레마는 극히 흥미로운 것인데, 명령 및 조회 책임의 분리(CQRS)라고 불리는 패턴을 사용하면 큰 도움을 받을 수 있습니다. 핵심은 CQRS 가 아키텍처가 아니라는 것입니다. 그것은 단순한 패턴이고 이벤트 기반 아키텍처가 가능하도록 도와줄 수 있는데 그 이유는, 그렇다, 우리 우주의 중심에 있는 이벤트들은 어디인가에서 반드시 존속되어야 하기 때문입니다.

궁극적으로, 대부분의 경우 어디인가에 데이터베이스가 있을 것입니다.

그러면, CQRS 란 정확히 무엇인가요?

우리가 이야기하고 있는 CQRS 의 예로 간단한 은행업무 유즈케이스를 살펴보십시오. 일반적으로 우리는 모든 어카운트 인터렉션을 이용하여 어카운트 서비스를 처리한다. 그 API 는 다음과 같이 정의됩니다.

```
AccountService

{

Public void createAccount(name)

Public Account getAccount(name)

Public  void  debitAccount(name,
amount)

Public void creditAccount(name,
amount)

Public AccountList

getInactiveAccounts()

Public AccountList

getOverdrawnAccounts()

}
```

CQRS 패턴은 이 하나의 서비스를 단순히 서로 다르면서 독립적으로 확장가능한 두 개의 서비스들로 분리합니다.

```
AccountChangeService

{

Public void createAccount(name,
acctMetadata)

Public void debitAccount(name,
amount)

Public void creditAccount(name,
amount)

}
```

그리고

```
AccountReaderService

{

Public Account getAccount(name)

Public AccountList
getInactiveAccounts()

Public AccountList
getOverdrawnAccounts()

}
```

커맨드와 쿼리의 수행은 근본적으로 다릅니다. 예를 들어, 커맨드와 쿼리는 언제나 다르게 스케일 되는데 그 이유는 서로 다른 속도로 스케일 되고 또한 서로 다른 오버헤드 페널티를 가지고 있기 때문입니다.

그러면 왜 우리는 이벤트 기반 아키텍처를 심플한 CQRS 와 결합해야 하고 또 이점은 무엇일까요?

이것을 생각해 보자. 일반적으로 데이터베이스를 가지고 데이터 디자인과 데이터 모델 작업을 시작합니다. 디자인할 때 내리는 결정이 모든 업스트림 서비스와 프로세싱에 영향을 미치게 되는데 왜냐하면 그것이 데이터 모델이 갖는 제약들 안에서 구동되어야 하기 때문이다. 이러한 생각때문에 이벤트가 아닌 데이터를 아키텍처의 중심에 사방으로 채우게 됩니다.

만약 당신이 생성, 갱신, 삭제와 같은 커맨드 액션들을 쿼리로부터 분리해내면, 본질적으로 도메인 상태를 변경하는 이벤트를 도메인 상태를 변경하지 않는 쿼리에서 분리한 것입니다. 당신이 이벤트를 아키텍처의 중심으로 다시 생각하도록 하는 것이 바로 이런 이벤트이고 또한 그것이 도메인을 모델링 하는 가장 자연스러운 방식입니다.

이 방식의 큰 장점은 이런 도메인 변경 이벤트를 새로운 마이크로서비스(간단하게 새로운 캐퍼빌리티와 기능을 추가)로 옮기거나 또는 빅데이터의 세계(분석작업을 수행할 수 있고 새로운 발견이 이루어지는)로 쉽게 옮길 수 있다는 것이다.

앞서 예로 들었던 은행의 경우에서 추가 기능의 예로 들 수 있는 것이 새로운 은행신용카드 상품의 마케팅 광고를 구현하는 것이다. EDA 와 CQRS 를 사용하면 어카운트 생성 이벤트들을 소비하기도 쉽고 또한 어카운트 메타데이터(예를 들어 개시잔액, 주소, 직업 등)를 기초로 신규고객에게 은행의 신용카드를 홍보하는 것도 쉬워진다.

CQRS 를 사용하면 정합성을 잃게 되지만, 성능과 가용성을 모두 향상시키고 또한 앞서 언급했듯이, 궁극적인 정합성을 포함함으로써 대부분의 유즈케이스에서 이에 대한 우려를 덜어낼 수 있다.

이벤트 기반 아키텍처를 통합하십시오

기업은 획기적인 가능성을 가진 이벤트들과 데이터 저장소들을 가득 가지고 있습니다. 가장 중요한 것은 이벤트 기반 아키텍처 바깥세상이 당신을 커맨드 스타일 인터렉션으로 혹은 그보다 더 나쁜 (끔찍한) 일괄처리방식으로 되돌아가지 않도록 하는 것입니다.

이런 원칙은 이미 이벤트 기반 방식의 시스템과 디바이스에서는 당연한 것이지만, 데이터베이스 중심의 시스템에서는 그렇지 않습니다.

마이크로서비스가 데이터 중심 시스템의 전통적 세상과 공존할 수 있게 하는 방법은 기초 데이터베이스에 변경이력추적(CDC)을 구현하는 것입니다.

오라클 데이터베이스에서 이 기법의 예가 CDC 유틸리티인 골든 게이트를 활용하는 것입니다. 이벤트가 발생하면 이것이 데이터베이스에 기록됩니다. 이런 변경을 추적하고 그것을 이벤트로 처리하여 마이크로서비스 플랫폼에서 활용할 수 있습니다. 이 방법은 기존의 시스템에 영향을 주지 않으며 또한 고비용의 코드변경을 하지 않아도 됩니다. 이때 해야 하는 작업의 대부분이 CDC 이벤트를 도메인 데이터 구조로 전환하는 것입니다.

최신의 시스템과 디바이스를 통합하는 것은 더 쉽다. 예를 들어, 사물인터넷(IoT) 디바이스는 본질적으로 이벤트 기반이고 소셜 미디어 플랫폼은 스트림이 가능하며, 심지어 많은 JEE

애플리케이션이 활용하기 쉬운 메시지 중심 빈(MDB)을 이용하고 있습니다. 핵심은 통합 때문에 다시 비 이벤트 기반의 세상으로 돌아가지 않도록 하는 것입니다.

알고 계셨나요?

IDC 의 연구에 따르면, 2021 년까지 엔터프라이즈 앱들이 하이퍼 애자일 아키텍처로 전환하고, 앱 개발의 80%는 마이크로서비스와 기능을 이용하여 클라우드 플랫폼에서 이루어지며, 새로운 마이크로서비스의 95%가 컨테이너로 배치될 것입니다.

UI 에 이벤트를 활용하십시오

이제 EDA 가 백엔드 마이크로서비스 프로세싱 아키텍처에 부여하는 효력에 대해 알았으니 그와 동일한 효력을 사용자 경험에 적용하는 것을 생각해 보아야 합니다.

직접적으로 말하자면, AJAX 는 사용자에게 비동기적으로 보이는 사용자 경험을 주지만, 사실은 비밀스럽게 웹 리소스들을 폴링하고 있을 뿐입니다. 만약 폴링 간격이 너무 길면 사용자 경험이 나빠지고 또한 '새로고침'을 시도하여 시스템에 불필요한 부하를 줄 수 있습니다.

이와 반대로, 폴링 간격이 너무 짧으면 업데이트가 실제로 이루어졌을 가능성이 낮아지고 그리고 리소스들은 다시 낭비됩니다. 사용자가 이 웹 애플리케이션을 사용하면 할수록, 이 업데이트 서비스를 요청하는 트래픽이 급격하게 증가하고(대부분은 새로운 결과가 없지만) 또한 비즈니스 비용을 증가시킵니다.

이 보다 나은 접근방식이 메시지 큐잉 텔레메트리 트랜스포트(MQTT) 혹은 웹소켓(WebSocket)과 같은 비동기식 프로토콜을 사용하는 것일 수 있습니다. 웹 애플리케이션은 접속연결 후 유저가 원하는 데이터를 구독하고 이벤트가 브라우저로 스트림 되기를 기다립니다. 이벤트가 실시간으로 도착하기 때문에 사용자 경험이 더 나아지고 또한 무의미한 요청으로 대역폭과 컴퓨팅 리소스를 낭비하지 않아도 되기 때문에 비즈니스 비용을 줄일 수 있습니다.

재해복구를 위한 토대로서의 이벤트

IT 재해에 대처하기 위해 애쓰고 있는 좋은 예로 항공산업을 살펴보십시오.

과거 수년 동안, 수만명의 여행객들이 테러나 지리정치학적 요인 또는 질병발생과 같은 이유들이 아닌 IT 장애의 케스케이딩 효과 때문에 발이 묶였었습니다. 그 중 많은 경우는 시스템이 부적절하게 디자인되었기 때문이었습니다.

다른 경우는 재해복구 시스템이 잘못 계획되었거나 복구를 실행하는데 시간이 너무 오래 걸렸습니다.

이것은 이벤트 기반 마이크로서비스와 직접적인 관련이 있습니다. 이벤트를 다른 액티브 또는 재해복구 사이트에 쉽게 복제할 수 있으므로 시스템은 항상 동기화 상태이며 구동준비가 되어 있습니다.

지금까지 이것은 각 데이터베이스 타입이 자신만의 고유한 복제 메커니즘을 사용하는 데이터 저장소 레벨에서 이루어졌습니다.

이것은 많은 컴포넌트들과 다른 전략들이 필요했기 때문에 복잡했습니다. 또한 많은 경우 동일한 이벤트가 여러 장소에 저장되었기 때문에 비용이 굉장히 높기도 했습니다.

데이터 대신 이벤트를 중심으로 시스템을 디자인하여 얻게 되는 좋은 부작용은 잠재적으로

재앙이 될 수 있는 상황에서도 마치 아무 일도 일어나지 않았다는 듯 비즈니스를 계속 할 수 있다는 것이다.

이벤트에 벤더불문 스탠다드 API 또는 와이어라인 프로토콜을 이용하십시오

"현재의 비즈니스 에코시스템은 대부분의 비즈니스에 있어 마이크로서비스로의 여정을 시작할 때 반드시 대처해야 하는 피할 수 없는 부담입니다"

TWEET

AWS 의 SQS/SNS, 구글 클라우드 Pub/Sub, Azure Service Bus 와 같은 클라우드 제공자의 서비스나 Apache Kafka, IBM MQ 그리고 TIBCO EMS 와 같은 이벤트 전송을 위한 독점 솔루션을 활용하는 것이 매력적으로 보일 수 있습니다. 이 방법에는 몇 가지 문제점들이 있습니다.

- **고정화.** 기술적인 또는 비즈니스적인 이유로 어떤 독점 솔루션을 더 이상 사용하지 않고자 할 때 문제가 발생합니다. 거의 모든 서비스와 통합에 영향을 미치기 때문에 잘못을 다시 바로잡기 위해서는 막대한 비용이 들어갑니다.
- **API 의 부족.** 마이크로서비스가 갖는 핵심적인 이점들 중의 하나가 어떤 프로그래밍 언어를 사용해야 하는지 강요하지 않는다는 점입니다. 독점 API 와 와이어라인은 보통 일부 언어만을 지원하기 때문에 민첩성과 선택의 범위를 제한합니다.
- **혁신의 중단.** 마이크로서비스의 또 다른 이점은 업계와 제품들의 진화에 따른 새로운 기술과 기법들을 사용할 수 있도록 하는 기능입니다. 벤더의 독점 API 에 갇혀 있으면 전환비용이 너무 높기 때문에 기술혁신에서 소외될 수 있다. 모노리스 애플리케이션의 문제와 같습니다

이벤팅/메시징 플랫폼의 구현방식을 선택할 때 그것이 Spring, JMS, NMS, Paho 그리고 Qpid 같은 스탠다드 API 나 또는 MQTT 와 AMQP v1.0 과 같은 와이어라인을 지원하도록 하는 것이 중요합니다.

solace.

이벤트 기반 마이크로서비스와의 장애물 극복하기

EDA 는 설계자들과 개발자들이 성공에 필요한 사고방식을 포용하는 한 엄청난 잠재력을 가지고 있습니다 문제는 마이크로서비스의 환경에서 EDA 를 구현할 툴링이 부족하다는 것입니다.

툴링은 현대적이지 않고, 고성능이 아니며, 오픈 되어있지 않고, 안정적이지 않습니다. 게다가 온프레미스 또는 다양한 퍼블릭 클라우드와 같이 비즈니스가 요구하는 형태로 배치될 수도 없습니다.

광역통신망을 통한 연계, 동기화 또는 복구 기능을 제공하지 않습니다. 마지막으로, API 관리/게이트웨이 솔루션은 이벤트 기반 마이크로서비스에 적합한 툴이 아닙니다. 그 일곱가지 이유는 다음과 같습니다.

알고 계셨나요?

2017 년 레드햇의 마이크로서비스 설문조사에 따르면 미들웨어 응답자의 67% 그리고 오픈시프트 응답자의 79%가 마이크로서비스를 새로운 프로젝트에 사용하는 정도만큼 기존 애플리케이션 재설계에도 사용하고 있다고 답했습니다

1

부족한 현대화. 메시징 자체는 오래전부터 있어 온 것이다. 15 년 전에는 애자일 개발이 없었습니다– 당시에는 폭포수 모델이 지배적이었고 전문가 팀들이 메시징 인프라를 관리하였습니다. 오늘날에는 데브옵스(DevOps)의 컨셉과 이와 연관된 민첩성으로 개발자들이 메시징 인프라의 설치와 환경설정, 그리고 데이터를 이용할 수 있을 때까지 기다릴 수가 없게 되었습니다. 모든 것이 자동화되어야 하고 데브옵스(DevOps) 친화적이어야 하며 또한 셀프서비스이어야 합니다. 이것은 메시징이 올바른 툴이기는 하지만 툴들이 오늘날의 민첩성의 필요 및 조건과 더 이상 양립할 수 없다는 것을 의미합니다. 이런 상황은 마이크로서비스 아키텍처가 제공하는 속도와 확장성 그리고 민첩성으로 인해 더욱 증폭됩니다.

2

높고, 예측할 수 없는 지연속도. 모노리식 애플리케이션들이 이산형 서비스들로 나누어지면서 지연속도가 증가하고 성능이 저하되는 한 지점이 생기는데 이로 인해 사용자 경험과 서비스수준협약을 충족하는 기능에 영향을 미치게 됩니다. 이벤팅/메시징 플랫폼의 지연속도를 최대한 낮추어야 할 필요성이 구현을 궁극적으로 성공시키는데 결정적인 영향을 미칩니다. 이 같은 일이 일어나면 첫 반응이 전단계로 돌아가서 더 크고 더 모노리식하게 그리고 네트워크 홉 수를 줄이기 위해 서비스의 재가용성과 민첩성을 저하시켜 결국 지연속도를 낮추는 것입니다. 이런 대응은 마이크로서비스가 가진 많은 이점들을 완전히 없애 버리는 것이지만, 많은 경우 기업이 독점 API 와 와이어라인 때문에 고정화 되어 결국 가장 쉬운 방법을 택하게 됩니다.

3

독점 프로토콜과 API. 수년 전, 당시에는 스탠다드 와이어라인이나 프로토콜이 없어서 IBM 과 TIBCO 그리고 다른 기업 메시징 회사들이 자체적인 서비스를 구현하였습니다. 이 회사들은 지금도 그렇게 하고 있는데 그 이유는 오픈 스탠다드가 경쟁을 심화시키고 또한 경쟁업체들이 혁신을 통해 제품을 차별화함으로써 고객들이 어렵지 않게 타사의 제품으로 바꿀 수 있다고 우려하기 때문입니다. 정확하다! 대부분의 벤더들은 많아야 한개의 스탠다드 와이어라인 혹은 오픈 API 를 실행함으로써 마이크로서비스를 구현할 수 있는 언어를 제한하고 또한 사물인터넷(IoT) 디바이스로부터 전송되는 것과 같은 이벤트의 더 큰 에코시스템을 배제합니다. 이것은 마이크로서비스가 제공하는 전체적인 가치와 민첩성을 저하시키게 되어 결국 데이터 이동을 일체화하기 위한 통합 심이 되어버립니다.

4

낮은 안정성. 앞으로도 가용성 백퍼센트의 서비스는 없을 것이고 또한 항상 완벽하게 구동되는 시스템도 없을 것입니다. 서비스의 버그, 네트워크의 마비 그리고 오염된 메시지와 같은 것들은 서비스가 어떻게 마이크로서비스 에코시스템을 둔화시키는지 혹은 어떤 부담을 주는지 그 실제적인 예들입니다. 이와 같은 비정상적인 상황이 발생할 경우, 개발자로서는 메시징과 같은 기본 서비스가 위험을 미리 알려주고, 리퀘스트들을 완충해 주고, 그리고 데이터 손실이 일어나지 않도록 해주기를 기대합니다. 하지만 실제로는 많은 메시징 시스템이 이런 일들이 일어날 때까지 잘 구동됩니다. 다시 말하자면, 상황이 어려워지면 우리를 실망시킨다는 얘기입니다. 아이러니하게도 이때가 우리가 이 시스템을 가장 필요로 하는 때입니다!

5

구축능력 리스크. 모든 클라우드 서비스 제공자들의 메시징/이벤팅 플랫폼은 다른 경쟁사의 클라우드나 많은 기업들이 핵심 업무기능을 관리하고 있는 온프레미스에 배치될 수 없습니다. 레거시 메시징 시스템은 클라우드 환경에서 쉽게 배치되지 않으며 특히 일부 시스템은 멀티캐스트 사용으로 인해 아예 배치되지 않습니다. 오늘날 끊임없이 진화하는 클라우드 전략 환경에서 배치성은 중요한 고려사항입니다. 하이브리드나 멀티 클라우드 아키텍처 방향으로 가고자 할 때 전환에 따르는 대가는 더욱 커질 수 있습니다.

6

광역통신망(WAN) 약점. 운영의 경제성 및 연속성 면에서 하이브리드 클라우드와 멀티 클라우드 아키텍처를 사용하는 것이 매우 일반적입니다. 이 모든 전략들을 위해서는 광역통신망(WAN)을 통해 신뢰할 수 있고, 안전하며 고성능인 방식으로 이벤트를 전송해야 합니다. 레거시 메시징 제품들은 본래 데이터센터 환경에서 사용되었기 때문에 불안정하고 장치간 왕복시간이 느린 광역통신망(WAN)의 일반적인 속성에는 잘 대응하지 못합니다.

7

API 관리/게이트웨이 – 잘못된 툴. API 관리/게이트웨이 스페이스가 호황을 누리고 있습니다. 많은 경우 B2B 와 웹 애플리케이션이 REST/HTTP 와 같은 웹 친화적인 API 를 이용하여 통합해야 하기 때문에 이 컴포넌트가 필요합니다. 도메인 변경 이벤트를 나타내는 이런 인터렉션은 다운스트림 프로세싱을 위해 이벤트로 신속하게 전환되어야 합니다. API 관리 및 게이트웨이는 이벤트 기반 마이크로서비스들 사이의 인터렉션이 가능하도록 시도하지 않으며 또한 마찬가지로 많은 비동기식 프로토콜이 이벤트를 웹 애플리케이션에 스트리밍할 수 있도록 하지 않습니다.

맺음말

처음에는 이벤트 기반 마이크로서비스가 어려워 보일 수도 있지만 대다수 마이크로서비스와 IT 전략의 미래이다. 적합한 이벤팅/메시징 플랫폼을 선택하는 것이 마이크로서비스의 막대한 이점을 실현하기 위한 가장 결정적인 단계 중의 하나입니다.

설계자와 개발자는 오늘날의 현대적이고 빠르게 진화하는 세상에서도 성공할 수 있도록 설계된 플랫폼이 필요합니다.

최신의 이벤팅 플랫폼으로 설계된 솔라스는 모든 클라우드와 플랫폼에 하나의 서비스로 쉽게 배치할 수 있으며 또한 광역통신망에도 쉽게 연결됩니다. 솔라스는 데브옵스(DevOps) 자동화를 지원하며 개발자들에게 거의 완전한 셀프서비스 경험을 제공해줍니다. 솔라스는 안정적이면서 고성능인 유일한 솔루션으로 개발자의 특별한 니즈를 충족시킵니다.

조나단 샤보우스키 솔라스 기술담당최고책임자실의 수석개발자. 그의 전문지식은 미국연방항공청(FAA), 위성지상시스템(GOES-R) 및 보건의료와 같이 다양한 분야에서 대규모의 미션 크리티컬 기업 시스템들을 포함하고 있습니다. 최근에는 마이크로서비스를 위한 이벤트 기반 아키텍처의 이용 및 퍼블릭 클라우드 구동 PaaS(Platform-as-a-Service)에서의 배치에 집중해 오고 있습니다.

solace.

솔라스에 대하여

당사는 이벤트 분배 메쉬를 만드는데 사용할 수 있는 고성능 메시지 브로커 제품 펍섭+(PubSub+)을 개발하였습니다. 하이브리드 클라우드와 IoT 환경 모든 곳에서 오픈 프로토콜과 API 를 이용하여 Pub/Sub, Queueing, Request/Reply 그리고 스트리밍을 지원하는 유일한 통합형 메시지 브로커 제품으로 당사는 애플리케이션, 디바이스 그리고 사람 사이의 정보를 클라우드를 통해 빠르고 신뢰할 수 있게 전송합니다. VoiceBase 그리고 Jio 와 같이 빠르게 성장하고 있는 기업들 뿐만 아니라 SAP, Barclays 와 같은 유수기업들 역시 레거시 애플리케이션의 현대화를 위해 그리고 데이터분석, 하이브리드 클라우드 및 IoT 전략의 성공적인 추진을 위해 당사의 스마트 데이터 이동 테크놀로지를 사용하고 있습니다. 더 자세한 내용은 홈페이지 solace.com. 에서 확인해 주십시오.

주요 고객 업체

주요 파트너 업체

solace.

solace.com