Microservices Architecture

Nedir?

Bir uygulama geliştirme mimarisidir.

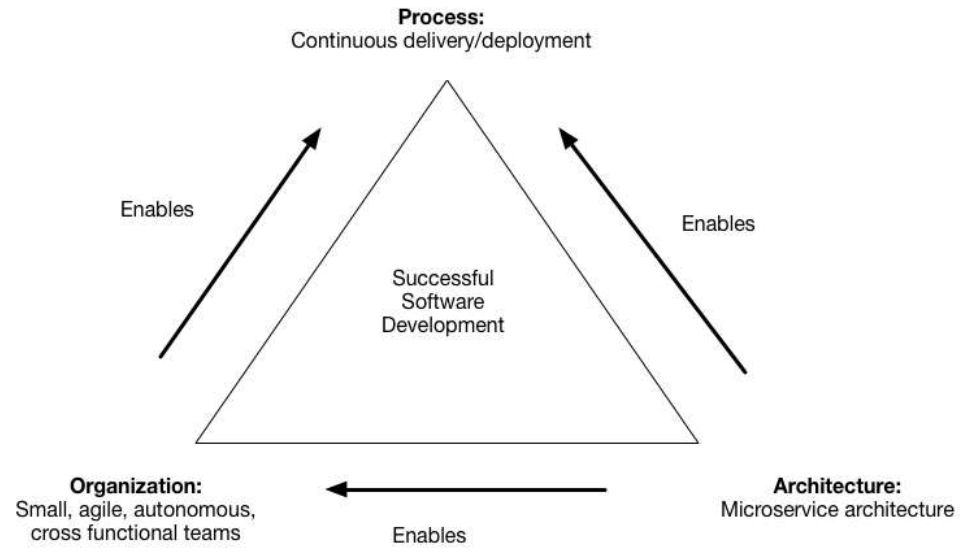
Birbirleriyle gevşek bağ ile bağlı uygulama parçacıklarının oluşturduğu mimari olarak da tanımlanabilir.

Microservices mimarisi büyük, karmaşık uygulamaların sürekli delivery/deploy yapılabilmesini mümkün kılar. Ayrıca şirketin ileriye dönük teknoloji gelişimini kolaylıkla yapılabilmesine imkan sunar.

Microservislerin mimarisel bazı problemleri vardır. Microservices mimarisini anlamak ve uygulamak için çeşitli patternler vardır. Bunun amacı vardır:

1- Uygulamanız için microservices mimarisinin uygun olup olmadığına karar vermenize yardımcı olmak
2- Microservices mimarisinin başarıyla kullanımanıza yardımcı olmak

Microservisin Amacı/Özellikleri



En temel amaç geliştirmeyi hızlandırmaktır. Bunu sağlayan en önemli unsur ise continuous delivery/deployment'tır.

Microservices başarılı/hızlı uygulama geliştirmeyi iki şekilde sağlar;
1- Testi basitleştirip ve servislerin bağımsız olarak konumlanmasını sağlayarak
2- Takım organizasyonunu, her biri bir veya daha fazla hizmetten sorumlu küçük, özerk takımlar şeklinde yapılandırarak sağlar.

Bu otomatik olarak sağlanan bir şey değildir. Bunu sağlamadaki en büyük unsur uygulamayı servislere dikkatli bir şekilde ayırmaktır.

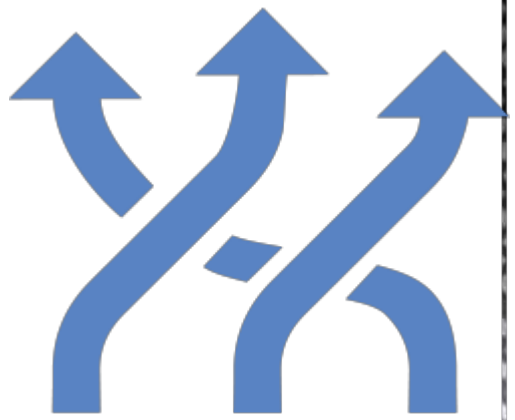
Microservisin Amacı/Özellikleri

Bir servis küçük bir takımın geliştirip test edebileceği kadar küçük olmalıdır. Bunu sağlamak için «Single Responsibility Principle»(Tek Sorululuk Prensibi) bize yol gösterici olabilir.

Uygulama aynı zamanda yeni eklenen ya da değişikliğe sebep olan gereksinimlerin çoğunu tek bir servisi etkileyecek şekilde parçalanmalıdır. Bunun nedeni birden çok ekip üzerinde koordinasyon gerektiren birden fazla servisi etkileyen değişikliklerin geliştirmeyi yavaşlatmasıdır.

Patternler

- Decomposition – Ayrıştırma
- Deployment
- Cross Cutting Concerns
- Communication Style
- External API
- Service Discovery
- Reliability – Güvenilirlik
- Data Management
- Security
- Testing
- Observability
- UI



Decomposition – Ayrıştırma

- Decompose by business capability
  - İş sorumluluklarına göre ayırma. Birçok organizasyonda bu konu multi-level hiyerarşi ile ayrılmıştır. Örneğin ; Ürün/Hizmet geliştirme, Ürün/Hizmet teslimi, Talep Yönetimi vb.
- Decompose by subdomain
  - Domain-Driven-Design yaklaşımı ile ayrıştırma yapılır. DDD uygulamayı domain olarak varsayar. Bir domain birden fazla alt domainlerden oluşur. Her bir sub domain uygulamanın modüllerini ifade eder. Alt domainler aşağıdaki yaklaşımla tasarlanır;
    - Core : İşin en değerli ve en önemli bölümlerini ifade eder
    - Supporting : Uygulamanın ana konularıyla alakalı süreçlerdir.
    - Generic : İşe özgü olmayan ve temel olması gereken konulardır

Deployment

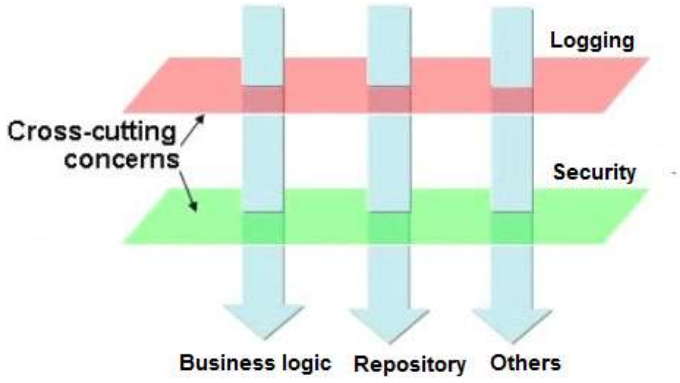
- İhtiyaçlar
  - Servisler birçok farklı dil,framework ve framework versiyonu ile yazılmıştır
  - Servisler verim ve kullanılabilirlik açısından birden fazla servisten oluşur
  - Servisler birbirinden bağımsız olarak deploy edilebilmeli ve scale edilebilmeli
  - Servis instansları diğerlerinden izole olmalı
  - Bir servisi hızlıca build edip yükleyebilmelisiniz
  - Bir hizmetin kullanabileceği sistem kaynalarını sınırlayabilmelisiniz
  - Her servis instansını izleyebilmelisiniz
  - Deploymentı güvenli yapabilmelisiniz
  - Uygulamayı olabildiğince düşük maaliyetle deploy edebilmelisiniz
- Yöntemler
  - Multiple service instances per host(physical or virtual)
  - Service instance per host
  - Service instance per VM
  - Service instance per Container
  - Serverless deployment
  - Service deployment platform

Cross Cutting Concerns

- Örnekler:
  - Externalized configuration
  - Logging
  - Health checks
  - Metrics
  - Distrubuted tracing

- Bunlar gibi generic CCC'lerin yanında uygulamaya özel CCC konuları da çıkabilir. Örneğin RDBMS kullanan bir uygulamada connection pool kullanmak gibi.
- Bu konuların uygulamanın tasarım aşamasında oturtulması gereklidir ileriye dönük refactoring yapmak maliyetli olabilir.

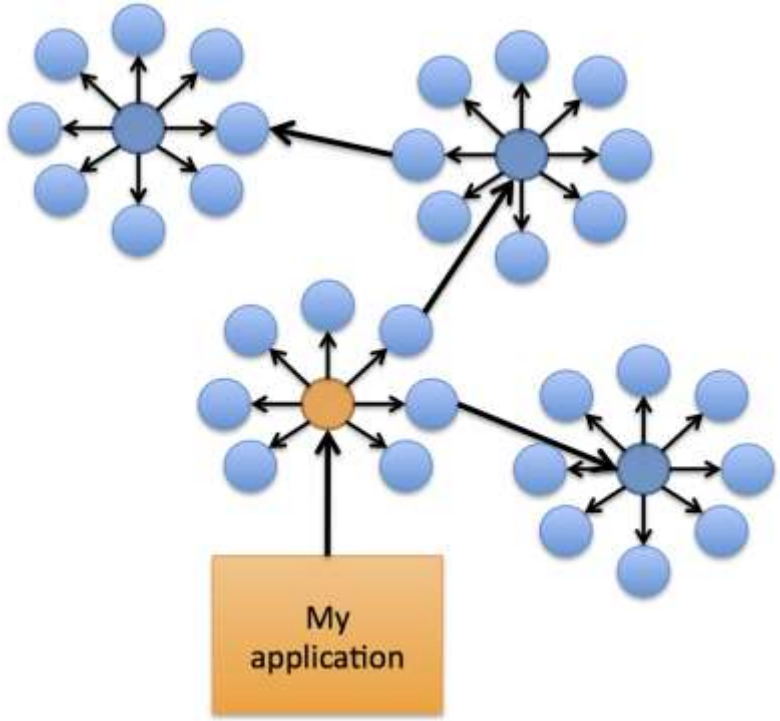
- Bunlara bağlı olarak:
  - Yeni bir microservices oluşturmak hızlı ve kolay olmalıdır
  - Yeni microsevice CCC'leri kapsayacak şekilde olmalıdır



Communication Style

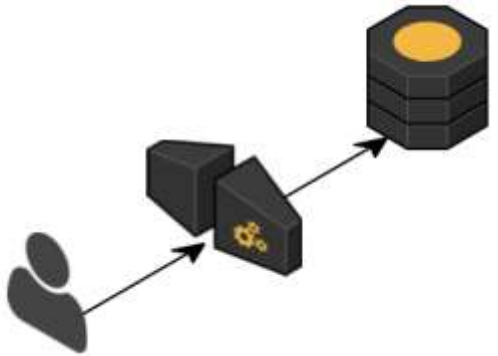
- Servisler istemcilerden gelen istekleri yerine getirmelidir. Hatta bu istekleri yerine getirirken diğer servisler ile çalışması gerekebilir. Bu durumda süreçler arası iletişim yapmaları gerekmektedir.

- Remote procedure invocation
  - REST,gRPC,Apache Thrift
- Messaging
  - Apache Kafka, RabbitMQ
- Domain-specific protocol
  - Email : SMTP, IMAP
  - Media Streaming : RTMP, HLS, HDS



External API - API Gateway / Backend for Front-End

- Faydaları
  - Encapsulate internal services
  - Load Balancing
  - Authentication
  - Monitoring
  - Caching
- Örneğin microservices mimarisi ile online store geliştiriyoruz ve ürün detay sayfası implemente ediyoruz. Bu sayfanın iki farklı arayüz ile hizmet vermesi isteniyor:
  - HTML5/Javascript based web page
  - Rest API ile çalışan Native Android ve Iphone client
- Ayrıca uygulama 3.parti uygulamaların kullanması için ürün detaylarını Rest API ile servis etmelidir.
- Ürün detay sayfasında birçok bilgiye ihtiyaç vardır:
  - Temel bilgiler : Başlık, yazar, fiyat
  - Satış geçmişi
  - Stok durumu
  - Satın alma opsiyonları : hard copy/e-book
  - Genellikle bu kitapla birlikte satın alınan ürünler
  - Kitap yorumları
  - Satıcılar



External API - API Gateway / Backend for Front-End

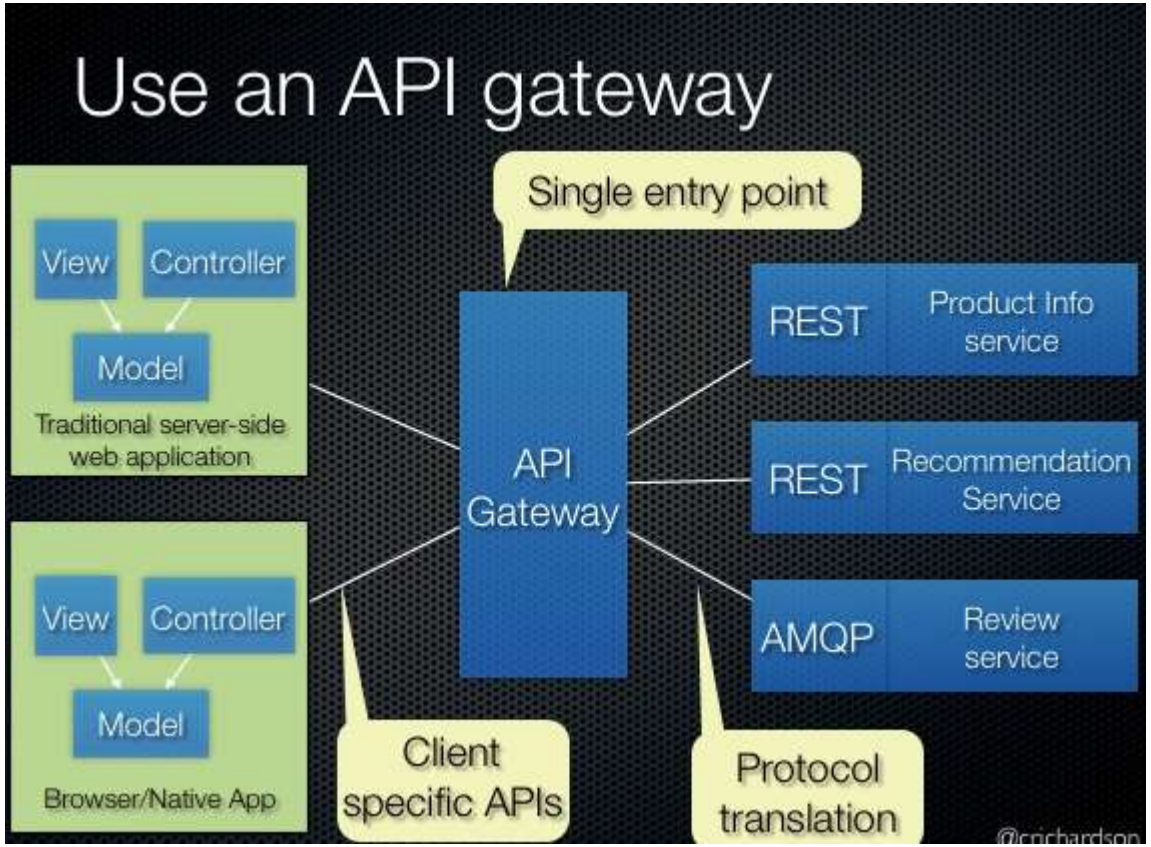
- Microservices mimarisi kullanıldığı için ürün bilgileri birden fazla hizmete yayılmıştır. Örneğin;
  - Product Info Service – basic information about the product such as title, author
  - Pricing Service – product price
  - Order service – purchase history for product
  - Inventory service – product availability
  - Review service – customer reviews
- Sonuç olarak ürün ayrıntılarını gösteren uygulama bu servislerin tümünden bilgi almak zorundadır.

External API - API Gateway / Backend for Front-End

- Microservices mimarisi kullanan bir uygulamanın clientleri servislere nasıl erişir?
  - Farklı clientler farklı datalara ihtiyaçları vardır. Örneğin masaüstü bilgisayardan açılan ürün detayları sayfasında mobil uygulamadan daha çok veri vardır.
  - Farklı clientler için farklı network performans problemleri oluşabilir.
  - Servis instanslarının sayıları dinamik olarak değişebilir(host+port)
  - Servislerin ayrıştırılması zamanla değişebilir ve bunun clientlerdan bağımsız olması gerekir
  - Servisler farklı iletişim protokolleri kullanabilir

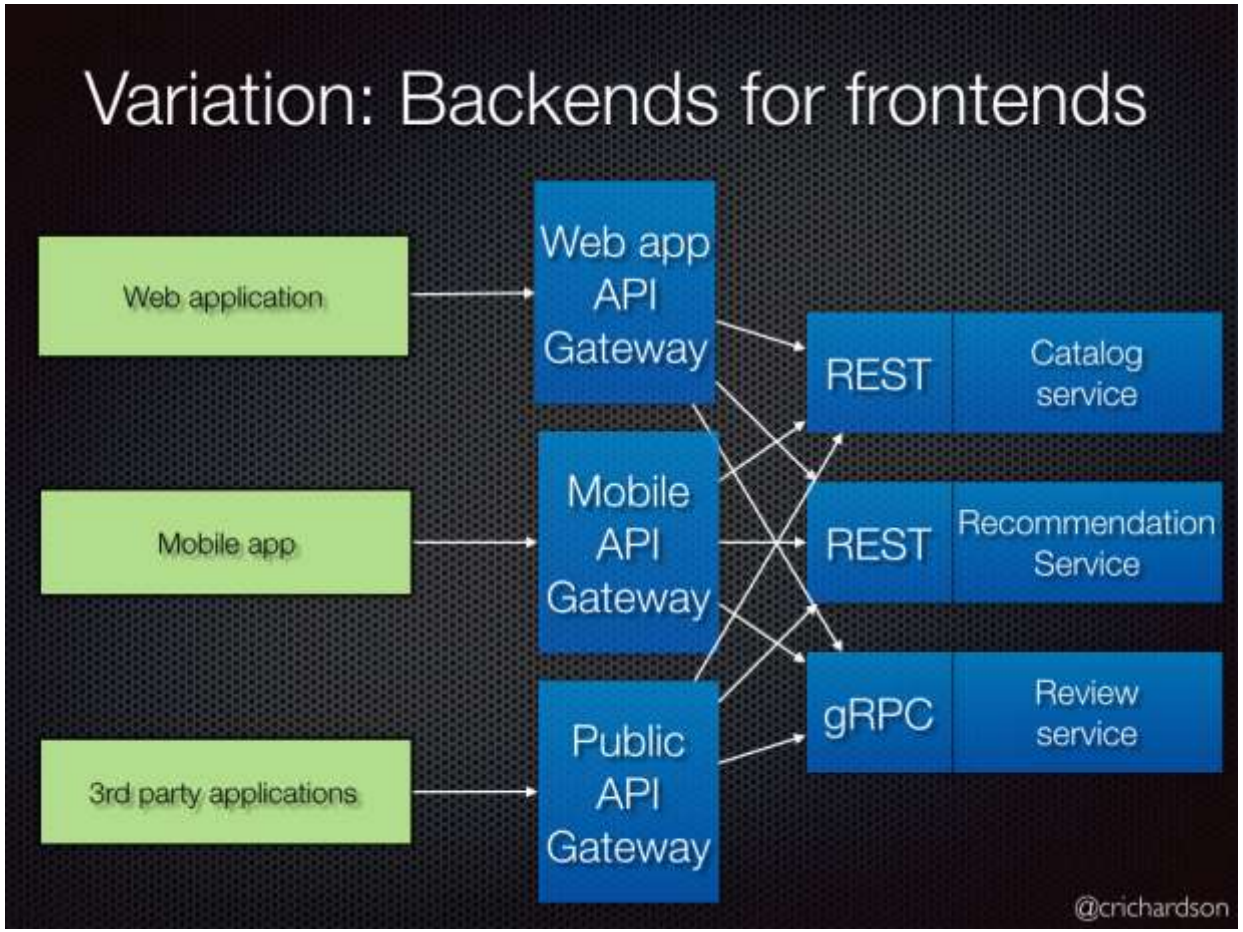
External API - API Gateway / Backend for Front-End

- Bu durumlar için clientlerin apilere erişirken tek giriş noktasını bilmesi gerekir. Gateway istekleri iki şekilde karşılayabilir; proxy ya da route



External API - API Gateway / Backend for Front-End

- Gateway'ler her cliente ortak bir apı sunmak yerine cliente uygun apileri sunabilir.
- Gatewayler ayrıca güvenlik katmanı olarak da kullanılabilir.



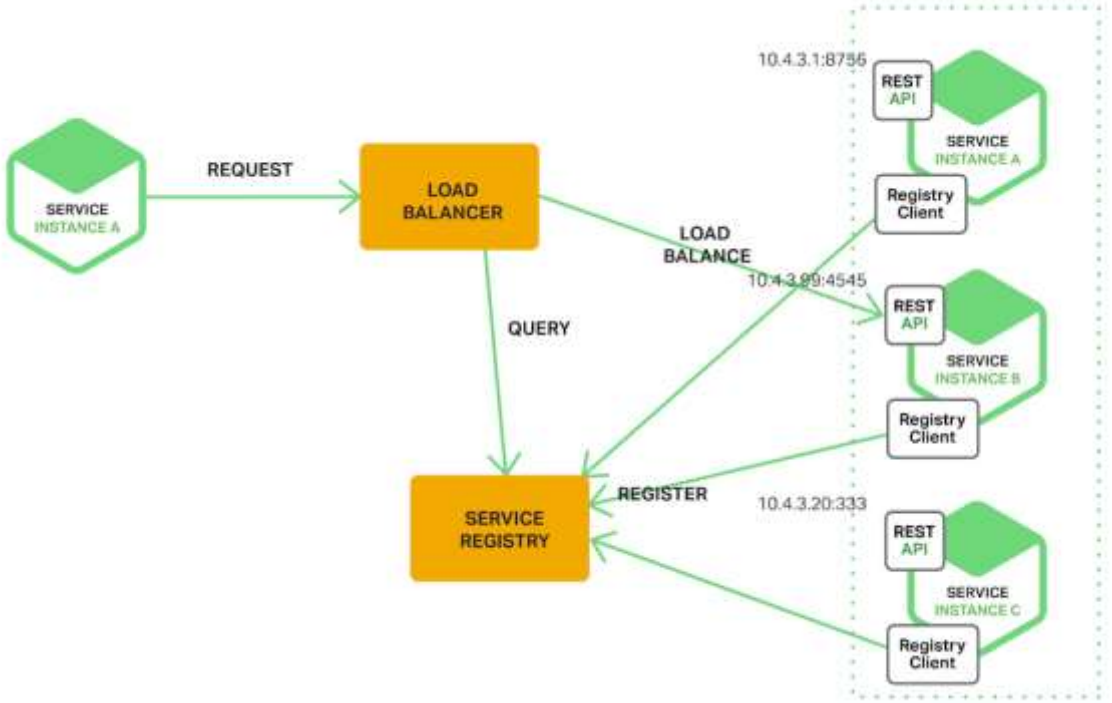
External API - API Gateway / Backend for Front-End

- Clientler servislerin nasıl bölümlendiğiniden izole eder
- Clientlerin servis adreslerini bilmesini gerekliliğini ortadan kaldırır
- Her istemci için uygun API'yi sağlar

- Complexitiyi artırabilir, doğru yönetilmesi gereken bir modeldir
- Request/response süreleri artabilir. Ama bu süre ihmal edilebilecek kadar azdır.

Service Discovery

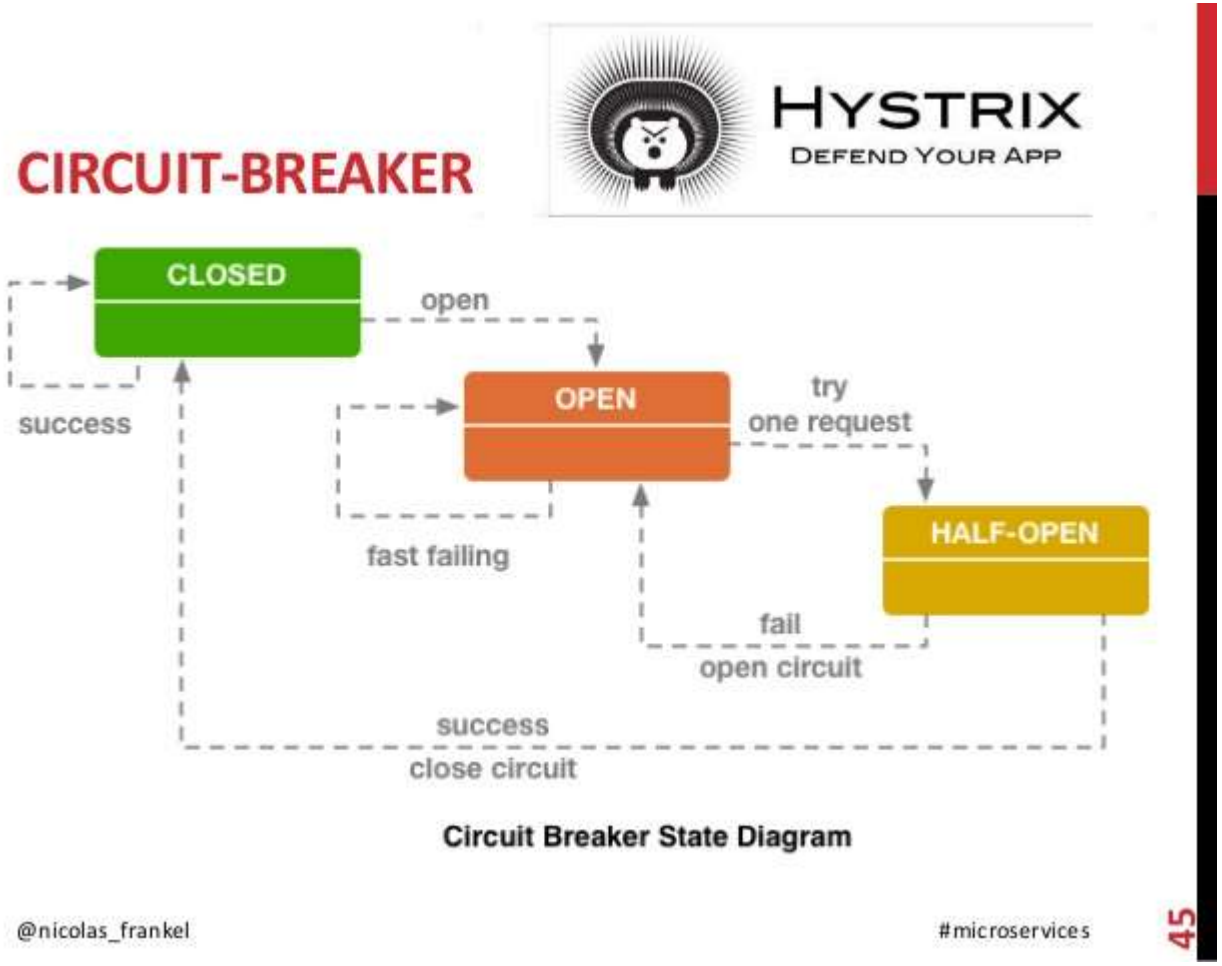
- Client-side discovery
- Server-side discovery
- Service registry
- Self registration
- 3rd party registration



Reliability

- Circuit Breaker
  - Servislerde endpointlerde oluşacak hataların sistemi tamamen kilitlemesini önlemek için kullanılmaktadır.
  - Eğer servis enpointinde belirli sayıda hata gerçekleşirse o endopointe gelen requestleri belirli süre engelliyoruz, o süre sonunda tekrar kontrol ediliyor eğer hata düzelmişse normal akışına devam ediyor.
  - Bu sayede endpointin kullandığı sistem kaynaklarını gereksiz kullanmayı ve sistem içerisindeki cascade hatalar oluşmasının önüne geçiyoruz
  - Retry : Genelde farklı sistemlere bağlı işlem yaparken bazı durumlarda geçici hatalar oluşabilir. Bu hataları elemine etmek için retry mekanizması kullanılarbilir. Örneğin ilk hatadan sonra 3 defa daha deneme yaparak 3.nü n sonunda başarılı cevap alabiliriz.
  - Fallback : Retry sürecini işlettik ve hala hata alıyorsak ve daha önce bir alternatif yöntem (vazgeçip başka bir yol izlemek) stratejisi belirlediysek bunu devreye sokabiliriz.

Reliability



Data Management

- Servisler serbestçe geliştirilebilecek ve ölçeklendirilebilecek yapıda olmalıdır. Bunun için servisler gevşek bir şekilde (loosely coupled) olmalıdır.
- Bazı işlemler birden fazla serviste veri değişikliğini zorunlu kılmaktadır.
- Bazı işlemler birden fazla servisin sahip olduğu veriyi sorgulamalıdır.
- Bazı işlemler birden fazla servisin sahip olduğu veriyi sorgulayıp birleştiriyor olabilir.
- Veritabanları bazen çoğaltılıp ölçeklenebilir.
- Farklı servislerin farklı depolama gereksinimleri olur. Bazı servisler RDBMS, bazıları NoSQL, bazıları ise GraphDB'ye ihtiyaç duyabilir.

Data Management

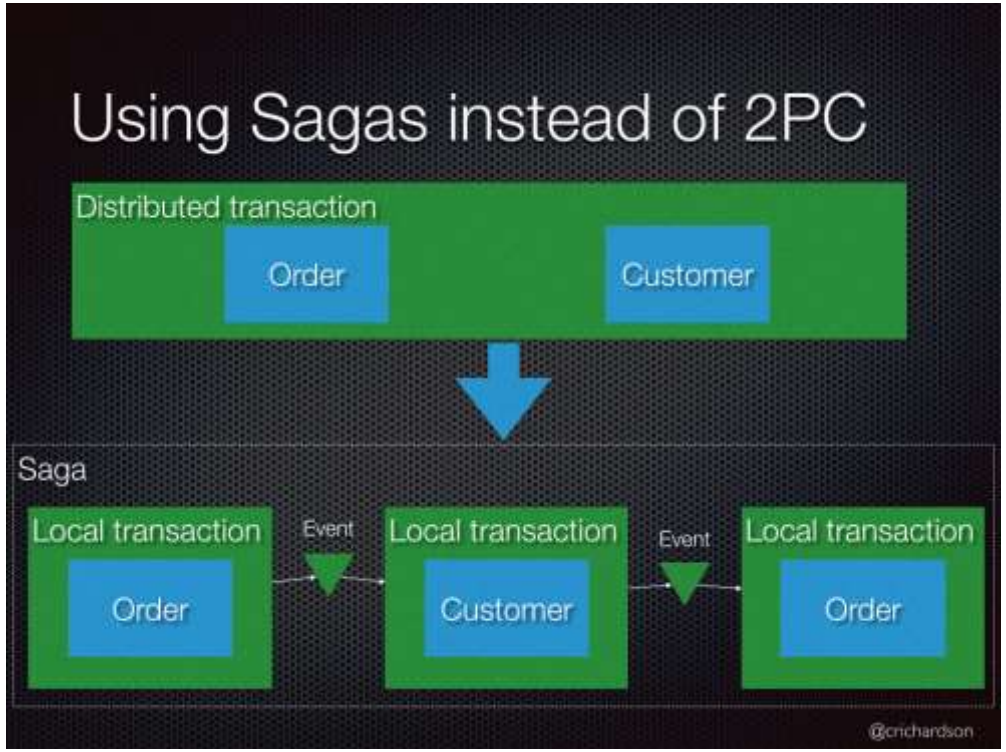
- Solutions
  - Database per Service
  - Shared database
  - Saga
  - API Composition
  - CQRS
  - Event sourcing
  - Transaction log tailing
  - Database triggers
  - Application events

Data Management - SAGA

- SAGA – Distrubuted Transaction
  - Her servisin local transactionalarının bir sırasıdır.
  - Her servis kendi verisini güncller ve bir sonraki işlemi tetikleyecek message veya event yayınlar.
  - Herhangi bir servisin işlemi başarısız olursa SAGA önceki işlemleri geri alacak bir dizi telafi işlemi gerçekleştirir.

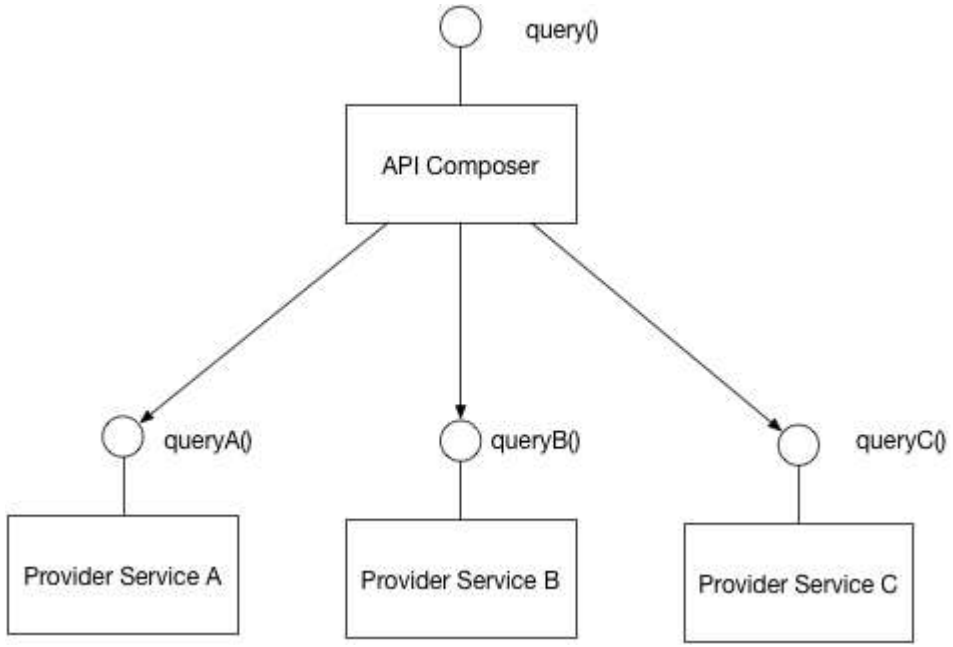
- Order Service bekleme durumunda bir sipariş oluşturur
- Customer Service sipariş için kredi reserv eder
- Order Service reservasyon işlemi sonucuna göre siparişi onaylar ya da iptal eder

- CreateOrder saga içinde OrderCreated eventi yayınlanır, CustomerService CreditReserved ya da CreditLimitExceeded eventi yayınlar.



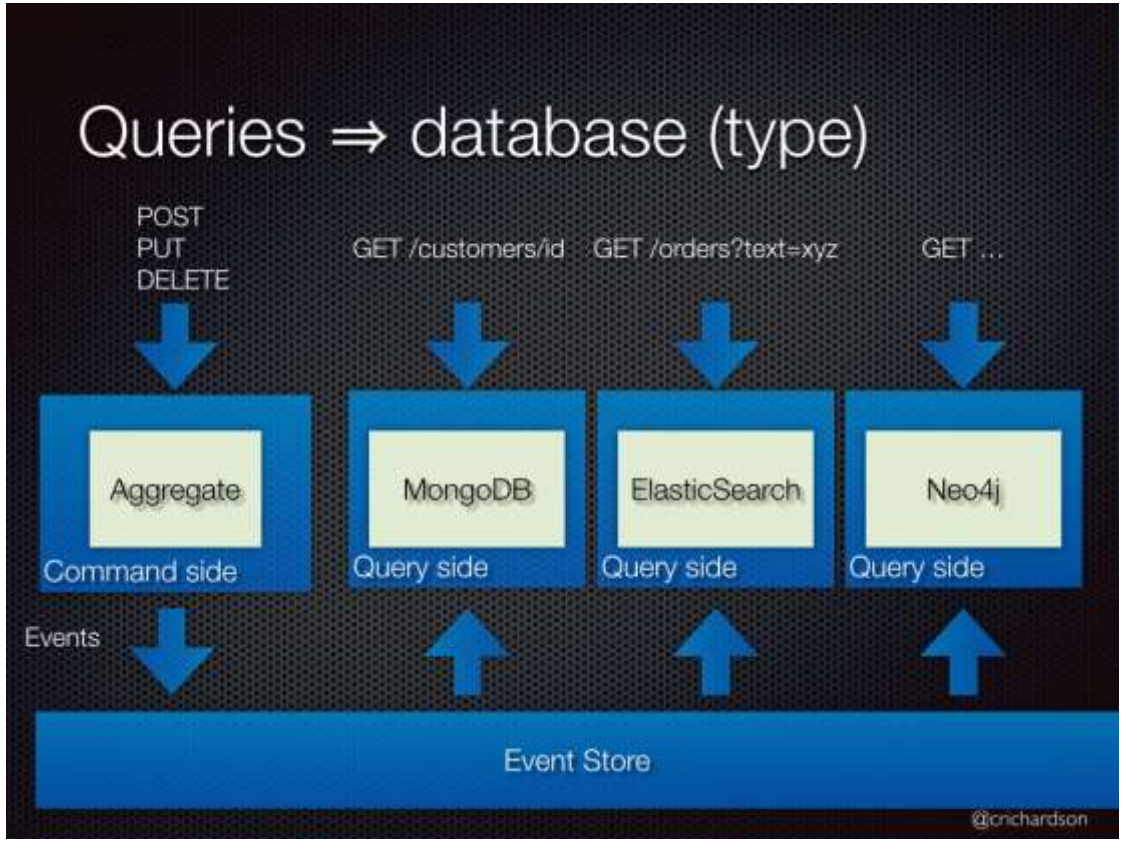
Data Management – API Composition

- Birden farklı servisten gelen dataları barındıran sorgulara ihtiyaç olduğunda bunu en efektif olarak yapmak için API Composer pattern uygulanabilir.



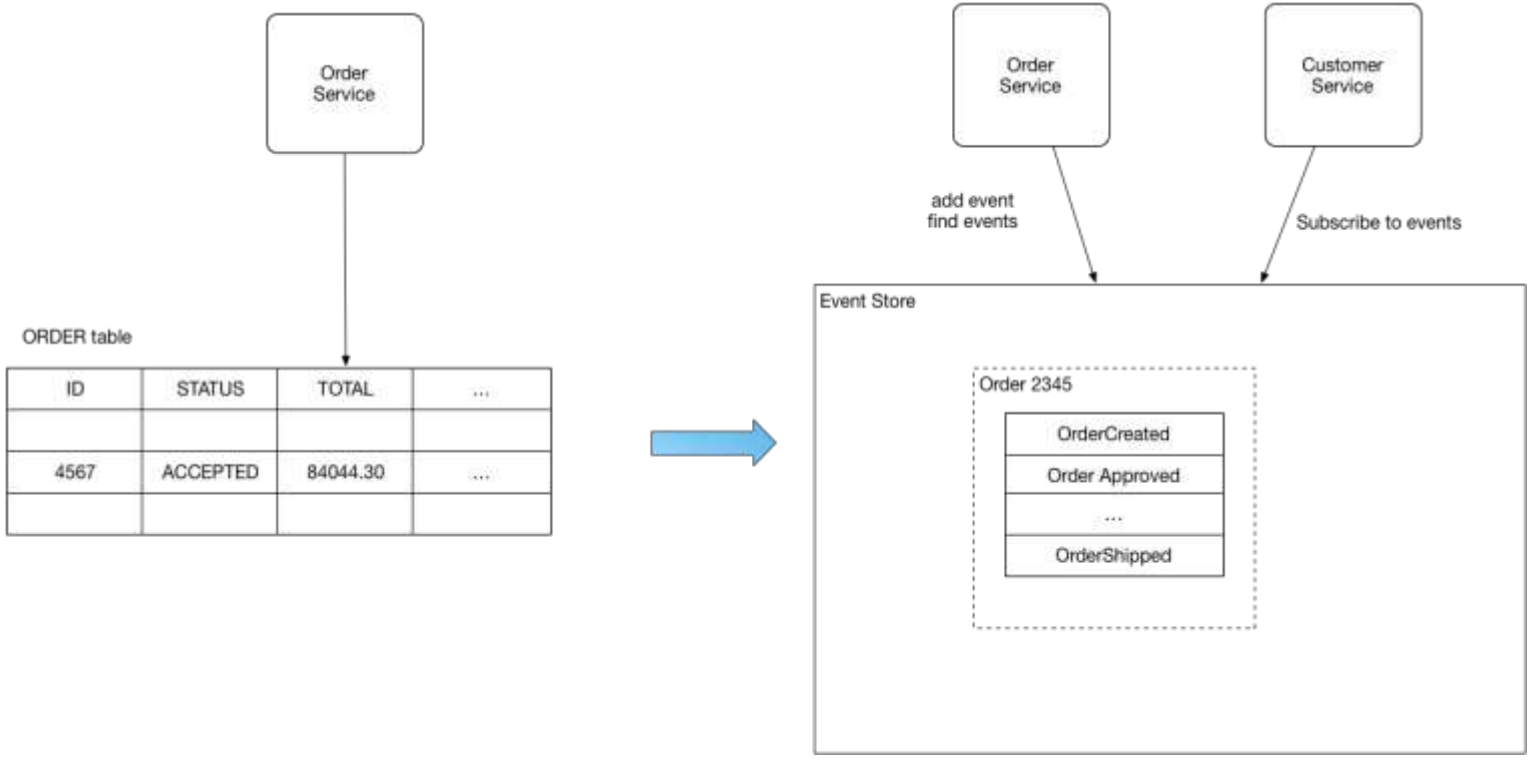
Data Management – CQRS

- CQRS – Command Query Responsibility Segregation
  - Uygulama command-side ve query-side olmak üzere ikiye ayrılır. Command-side insert,update vb. işleri yapar. Query-side ise commandlar sonucu oluşan eventler ile sorgu datalarını günceller.



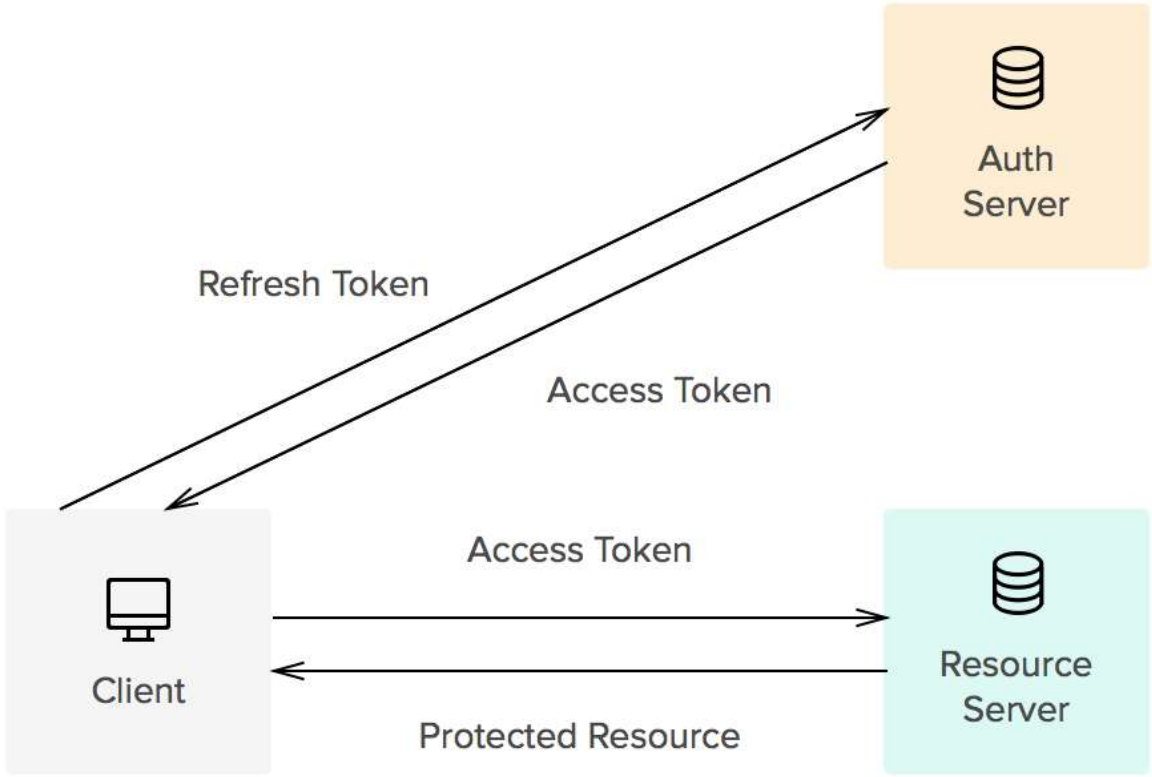
Data Management – Event Sourcing

- Temel prensip olarak doğrudan tablolarda entity olarak verileri saklamak yerine domainler üzerinde gerçekleşen olayları saklamaktır. Yani sipariş oluşturma sonucunda sipariş için bir tabloya kaydetmek yerine sipariş oluşturuldu eventini sipariş bilgisi ile birlikte saklar.
- Bu model özellikle dağıtık sistemlerde veri tutarlılığını saklamak için önemli bir çözümdür. Verinin eventlerini replay ettiğinizde hangi aşamada ne durumda olduğunu her zaman elde edebilirsiniz.



Security

- Access Token

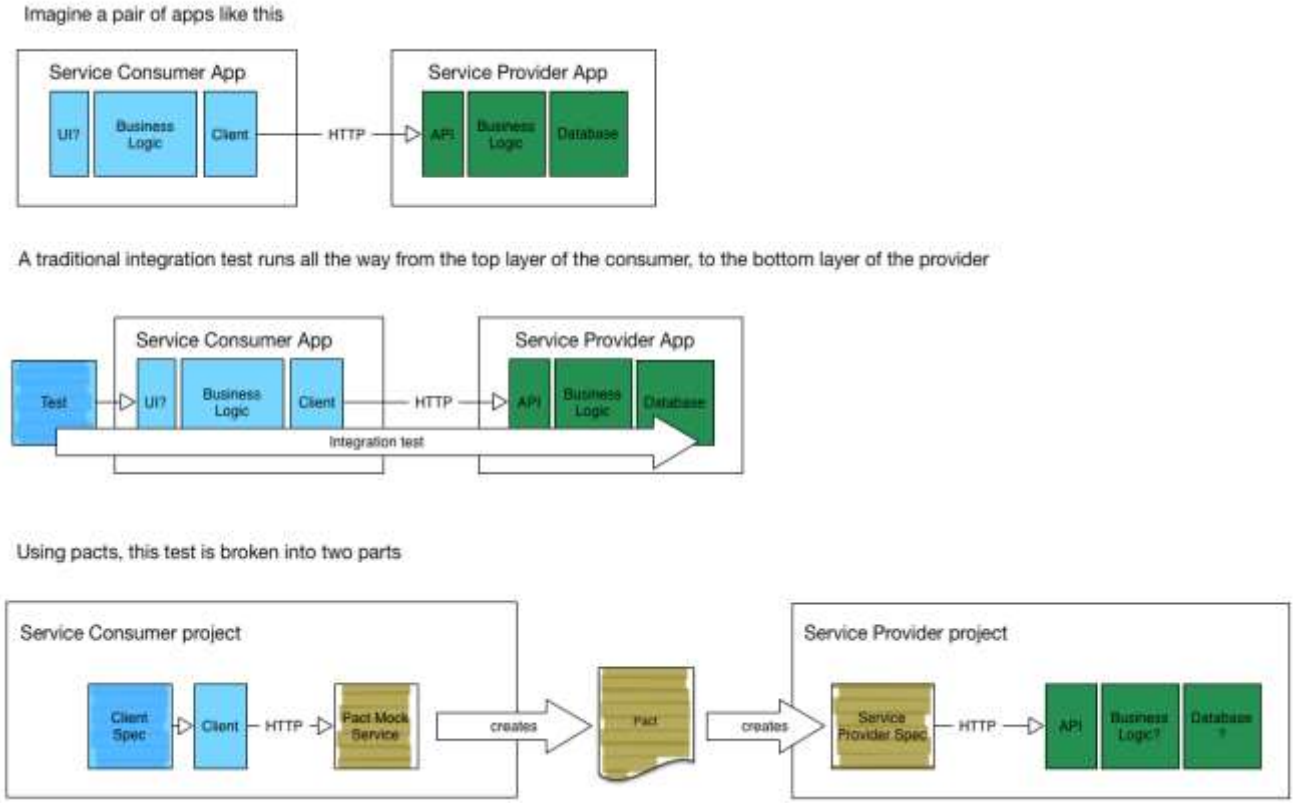


Testing

- Service Component Test
  - Uçtan uca servisleri test etmek zor ve yavaştır. Bunun için servisleri yalıtılmış olarak kendi başlarına test etmek gerekir.
  - Bu kolay ve hızlıdır.
  - Bu bazı durumlarda testlerin development ortamında başarılı fakat production ortamında hatalı olmasına sebep olabilir.

Testing

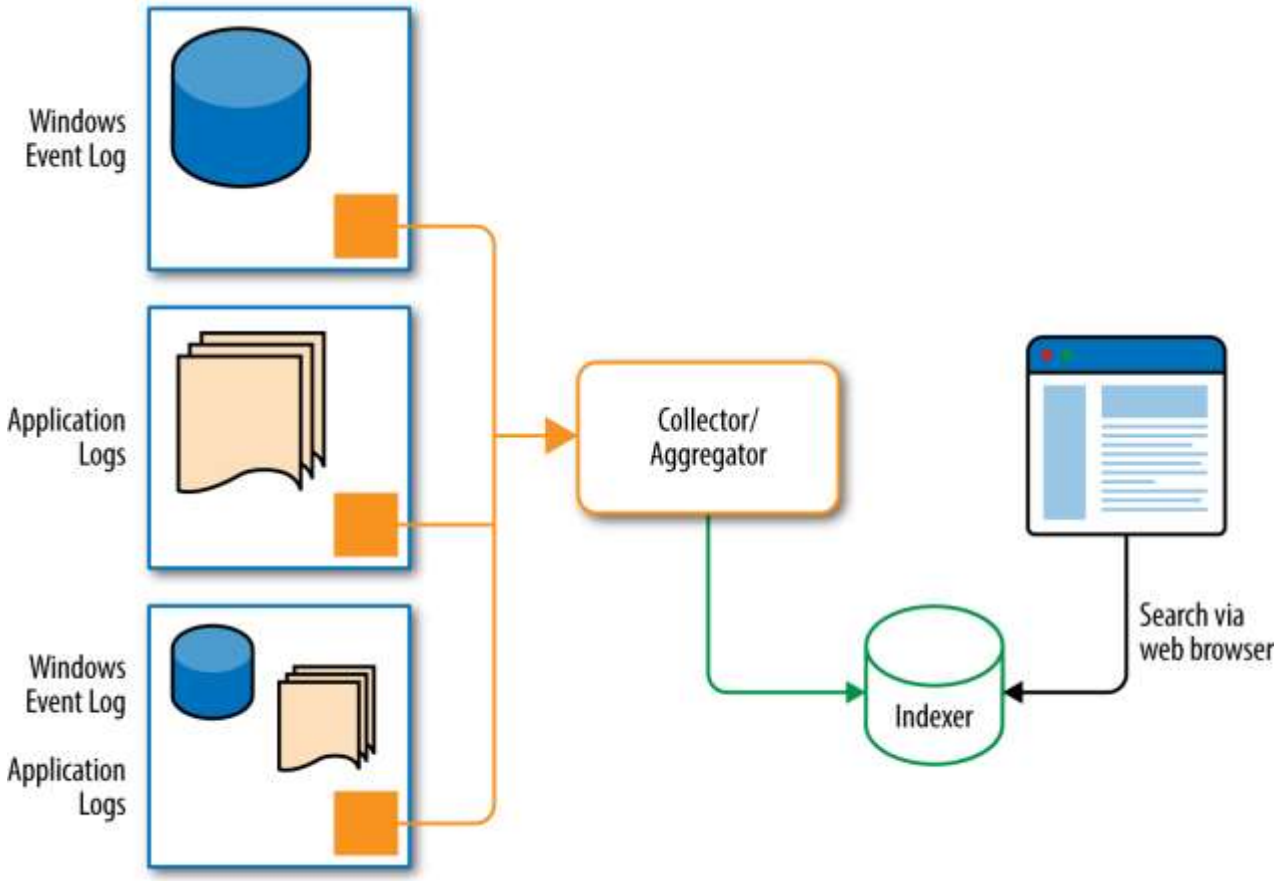
- Service Integration Contract Test
  - Bu patternde her servis kendini kullanan servislerin ona sağladığı test paketlerini uygulayarak entegrasyon testini sağlamış olur.



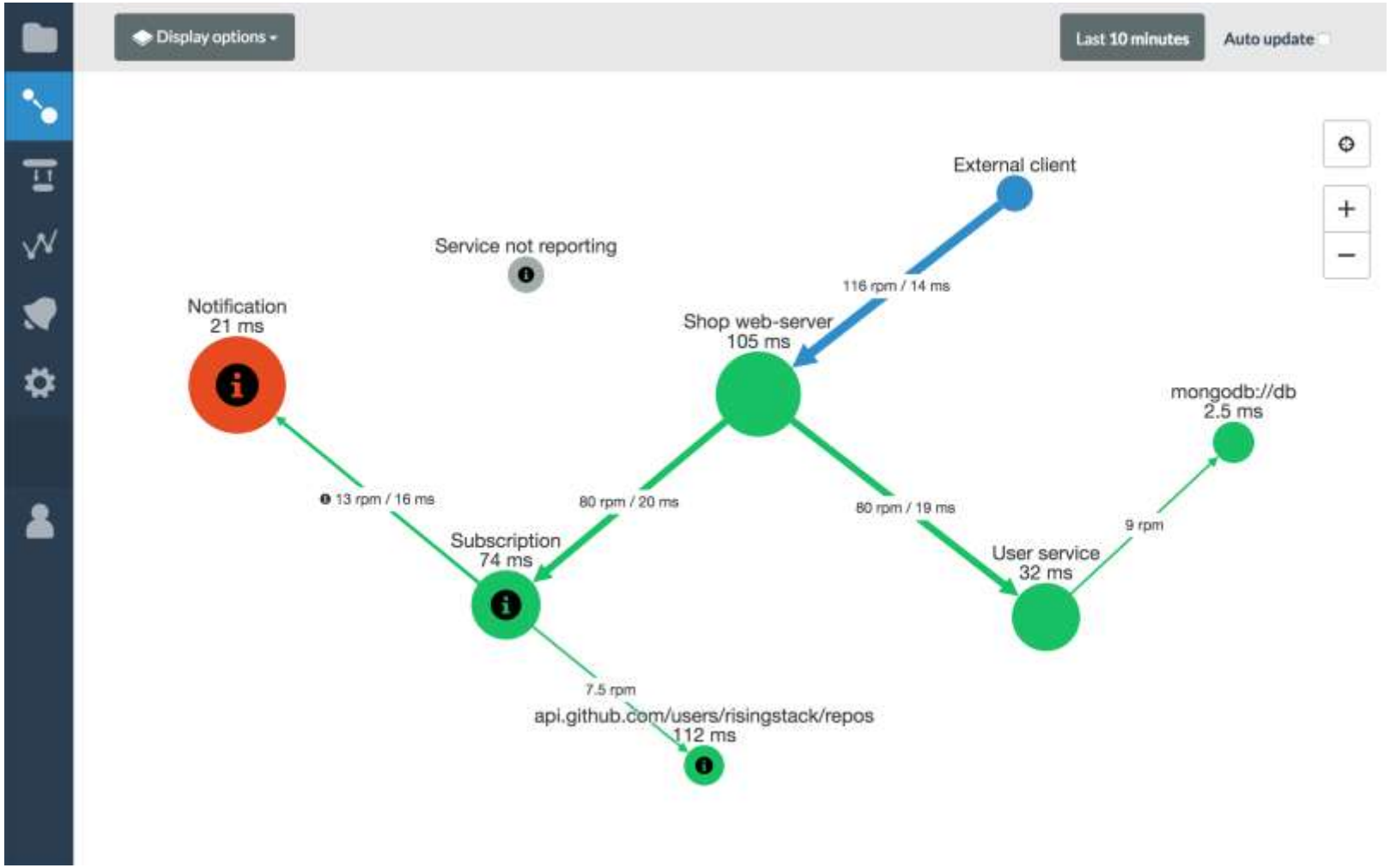
Observability

- Log aggregation
- Application metrics
- Audit logging
- Distributed tracing
- Exception tracking
- Health check API
- Log deployments and changes

Observability – Log aggregation

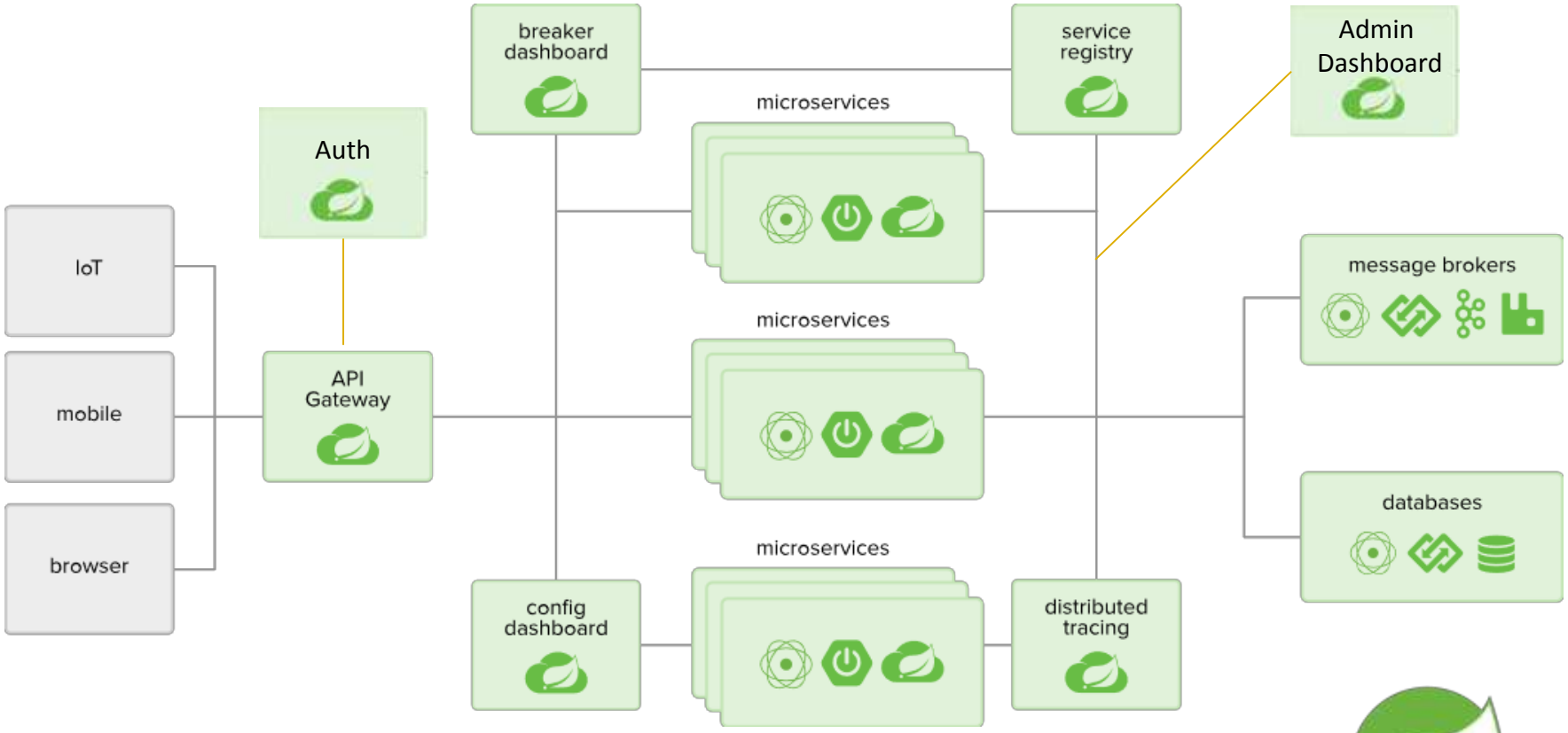


Observability – Distributed tracking



Microservices with Spring Cloud

- Microservices uygulamalarında ihtiyaç duyulan önceki slaytlarda belirttiğimiz gereksinimleri Spring Boot uygulamalarında kolayca kullanabilmeyi sağlayan kütüphanedir.



cloud

Spring Cloud Demo

Kaynak

- http://microservices.io
- https://cloud.spring.io