Dynatrace API v2 ile Servis Topolojisi Otomasyonu: Teknik Uygulama Kılavuzu

Dynatrace API v2 ile Servis Topolojisi Otomasyonu: Teknik Uygulama Kılavuzu

1. Yönetici Özeti

Bu kılavuz, modern ve dinamik mikroservis ortamlarında servisler arası bağımlılıkların otomatik olarak çıkarılmasına yönelik teknik bir strateji sunmaktadır. Sürekli entegrasyon ve teslimat (CI/CD) süreçlerinin hakim olduğu günümüz mimarilerinde, servis topolojisini manuel olarak takip etmek pratik olarak imkansızdır. Bu otomasyon projesi, operasyonel verimliliği artırmak, Konfigürasyon Yönetimi Veritabanı (CMDB) sistemlerini güncel tutmak ve ağ segmentasyonu gibi güvenlik uyumluluğu denetimlerini doğrulamak için zorunlu bir gerekliliktir. Programatik olarak çıkarılan bu topoloji haritası, yalnızca gözlemlenebilirlik sağlamakla kalmaz, aynı zamanda temel bir denetim ve uyumluluk kanıtı olarak da hizmet eder. Bu, örneğin 'Ödeme' yönetim bölgesindeki (Management Zone) bir servisin, 'Dahili İK' yönetim bölgesindeki beklenmedik bir servisi çağırdığının anında tespit edilmesini sağlayarak potansiyel bir güvenlik ihlalini veya hatalı bir ağ kuralını anında ortaya çıkarır.

Projenin temel hedefleri şunlardır:

- Dynatrace tarafından izlenen tüm servis varlıkları arasındaki çift yönlü `calls` (çağıran) ve `called_by` (çağrılan) ilişkilerini programatik olarak çıkarmak.
- Elde edilen ham veriyi, CMDB sistemleri, otomasyon betikleri ve görselleştirme araçları gibi diğer kurumsal sistemler tarafından kolayca işlenebilecek standart ve yapısal bir formata dönüştürmek.

Bu projenin hedeflenen nihai çıktısı, servis bağımlılık ağını temsil eden **"Düzleştirilmiş Edge Listesi" (Flattened Edge List)** formatında bir veri setidir. Bu

format, her bir servis bağımlılığını `(Kaynak Varlık ID, Hedef Varlık ID, İlişki Tipi)` şeklinde tekil ve atomik bir kayıt olarak temsil eder. Bu yapı, topoloji verilerinin analizini, sorgulanmasını ve diğer sistemlere entegrasyonunu büyük ölçüde basitleştirir.

Bu kılavuzun devamında, bu hedeflere ulaşmak için kullanılacak olan Dynatrace API v2'nin yapılandırma detayları, parametre seçimlerinin stratejik gerekçeleri ve veri işleme mantığı ayrıntılı olarak ele alınacaktır.

2. API Konfigürasyonu ve Parametre Seçim Gerekçeleri

Servis topolojisi otomasyonunun temeli, Dynatrace API'sine yapılan çağrının doğru ve verimli bir şekilde yapılandırılmasına dayanır. Doğru endpoint'in ve sorgu parametrelerinin seçimi, elde edilen verinin kalitesi, bütünlüğü ve sistem performansı üzerinde doğrudan bir etkiye sahiptir. Bu bölümde, veri çıkarma işleminin temelini oluşturan API isteğinin teknik özellikleri ve bu seçimlerin arkasındaki stratejik gerekçeler detaylandırılmaktadır.

Kullanılacak olan API endpoint'i, **Monitored Entities API v2** olarak belirlenmiştir. Bu endpoint, Dynatrace versiyon 1.263'ten beri resmi olarak kullanımdan kaldırılmış olan eski Smartscape API'sinin yerini alan modern ve desteklenen tek yöntemdir.

- **HTTP Metodu:** `GET`
- **Endpoint:** `/api/v2/entities`

API'ye yapılacak çağrılar için temel URL yapısı aşağıdaki gibidir. `{your-environment-id}` kısmı, kuruma özel Dynatrace ortam kimliği ile dinamik olarak doldurulmalıdır: `https://{your-environment-id}.live.dynatrace.com/api/v2/entities`.

Aşağıdaki tablo, topoloji verilerini çekmek için kullanılacak zorunlu sorgu parametrelerini ve bu parametrelerin seçilme nedenlerini analitik bir dille açıklamaktadır.

Parametre	Değer	Stratejik Gerekçe
`entitySelector`	`type("SERVICE")`	Yalnızca servis tipi varlıkları hedefleyerek sorgu kapsamını daraltır, gereksiz altyapı bileşenlerini (host, process vb.) filtreleyerek

		veri işleme yükünü en aza indirir.
fields	+fromRelationships.calls,+toRelationships.called_by	Servisler arası çift yönlü (giden ve gelen) bağımlılıkları getirmek için zorunludur. Bu parametre olmadan topoloji haritası eksik kalır. Veri yükünü optimize etmek için sadece gerekli ilişki alanlarını talep eder.
pageSize	500	Tek bir yanıtta alınacak maksimum varlık sayısını belirler. Bu değeri API tarafından izin verilen en yüksek değer olan 500'e ayarlamak, toplam API çağrı sayısını en aza indirerek veri çekme sürecini hızlandırır ve rate limit'e takılma riskini azaltır.

Bu parametrelerin dikkatli bir şekilde birleştirilmesi, binlerce servisin bulunduğu kurumsal ortamlarda bile verimli ve ölçeklenebilir bir veri çekme stratejisi sağlar. Bir sonraki bölümde, bu konfigürasyonla elde edilen verilerin büyük veri setlerinde eksiksiz olarak nasıl çekileceğini yöneten algoritma ve akış mantığı incelenecektir.

3. Algoritma ve Akış Mantığı (Flow Logic)

Büyük ölçekli kurumsal ortamlarda, binlerce servis varlığının topoloji verisini güvenilir ve kesintisiz bir şekilde çekmek, sağlam bir algoritma mantığı ve proaktif hata yönetimi stratejileri gerektirir. Bu mekanizmalar, otomasyon betiğinin yalnızca küçük ortamlarda değil, aynı zamanda kurumsal düzeyde de ölçeklenebilir ve dayanıklı olmasını garanti altına almak için kritik öneme sahiptir. Bu bölümde, veri bütünlüğünü sağlayan sayfalama mantığı ve API kullanım limitlerine uyumu sağlayan hata yönetimi ele alınmaktadır.

Durum Tabanlı Sayfalama (Stateful Pagination) Stratejisi

Dynatrace API v2, büyük veri setlerini yönetmek için imleç tabanlı (cursor-based) bir sayfalama mekanizması kullanır. Tüm servis verisini eksiksiz bir şekilde çekmek için betiğin aşağıdaki durum tabanlı döngü mantığını uygulaması zorunludur:

1. **Başlangıç İsteği:** İlk API çağrısı, bölüm 2'de tanımlanan `entitySelector`, `fields` ve `pageSize` parametrelerinin tümünü içerir. Bu istek, veri setinin ilk sayfasını getirir.
2. **Yanıtın Değerlendirilmesi:** API yanıtının kök (root) seviyesinde `nextPageKey` alanının varlığı kontrol edilir. Bu alan, alınacak bir sonraki veri sayfasının "adresi" olan bir imleç (cursor) anahtarı içerir.
3. **Döngü Koşulu:** `nextPageKey` alanı dolu (null değil veya mevcut) olduğu sürece döngüye devam edilir.
4. **Sonraki İstekler (Kritik Kural):** Döngünün sonraki tüm adımlarında yapılacak API istekleri, sorgu parametresi olarak **sadece ve sadece** bir önceki yanıttan alınan `nextPageKey` değerini içermelidir. Orijinal `entitySelector` veya `fields` gibi parametrelerin bu takip isteklerinde tekrar kullanılması, API sözleşmesinin doğrudan ihlalidir ve API'nin `HTTP 400 (Bad Request)` hatası döndürmesiyle sonuçlanacaktır.
5. **Döngünün Sonu:** API yanıtında `nextPageKey` alanı artık dönmediğinde (null veya eksik olduğunda), tüm veri sayfalarının başarıyla çekildiği anlaşılır ve döngü güvenli bir şekilde sonlandırılır.

Rate Limit Yönetimi (HTTP 429) ve Synchronized Back-off

Dynatrace SaaS ortamı, API kaynaklarının adil kullanımını sağlamak için genel bir erişim limiti uygular. Bu limit genellikle **dakikada 50 istek** olarak belirlenmiştir. Bu limitin aşılması durumunda API, `HTTP 429 (Too Many Requests)` durum koduyla yanıt vererek geçici olarak erişimi engeller.

Bu durumu profesyonelce yönetmek için otomasyon betiğinin reaktif bir bekleme stratejisi uygulaması gerekir. Basit bir "Exponential Back-off" (üstel geri çekilme) stratejisi, ne kadar bekleneceğini *tahmin etmeye* dayandığı için verimsizdir ve çok uzun bekleme veya çok erken yeniden deneme riskleri taşır. Mimarî olarak üstün olan ve standart uygulama gereği olan yöntem, **"Synchronized Backoff"** (senkronize geri çekilme) stratejisidir:

- `X-RateLimit-Reset` **Başlığını Oku:** `HTTP 429` hatası alındığında betik, HTTP yanıt başlıkları arasında gelen `X-RateLimit-Reset` değerini okumalıdır.

- **Senkronize Bekleme:** Bu başlık, API limit kotasının sıfırlanacağı zamanı **Unix epoch zaman damgası (timestamp)** olarak içerir. Betiğin mantığı, bir sonraki isteği göndermeden önce sistem saati bu zaman damgasında belirtilen zamana ulaşana kadar yürütmeyi duraklatmalıdır. Bu deterministik yaklaşım, betiğin Dynatrace API kota döngüsüyle mükemmel bir şekilde senkronize çalışmasını sağlayarak, zaman veya istek israfı olmaksızın verimi en üst düzeye çıkarır.

Bu algoritmik kontroller, veri çekme işleminin tamamlanmasının ardından, elde edilen ham verinin anlamlı bir topoloji haritasına dönüştürüleceği veri işleme ve ayrıştırma adımlarına güvenli bir geçiş sağlar.

4. Veri İşleme ve Parsing

API'den başarılı bir şekilde çekilen ham JSON verisi, projenin nihai hedefine ulaşmak için işlenmeli ve yapılandırılmalıdır. Bu bölüm, iç içe geçmiş (nested) JSON yanıtının nasıl ayrıştırılacağını (parse edileceğini) ve otomasyon araçları tarafından kolayca tüketilebilecek, kullanılabilir bir topoloji haritasına nasıl dönüştürüleceğini detaylandırmaktadır. Bu adım, verinin mantıksal olarak yapılandırılarak projenin nihai çıktısının üretilmesini sağlar.

API yanıtının kök seviyesinde bulunan `entities` dizisi (array), sorgu kriterlerine uyan tüm servis nesnelerinin bir listesini içerir. Bağımlılıklar, her bir servis nesnesi içinde bulunan `fromRelationships` (giden çağrılar) ve `toRelationships` (gelen çağrılar) nesneleri altında yer alır.

Ancak, bağımlılık ID'leri doğrudan bu nesnelerin altında değildir. Bunun yerine, hedef varlık tipine göre gruplandırılmış daha derin bir seviyede bulunur. Uygulayıcılar için bu yapısal detayın gözden kaçırılması, ayrıştırma hatalarına yol açacaktır. Doğru yapı aşağıdaki gibidir:

```
"fromRelationships": {
  "calls": {
    "SERVICE": [
      { "id": "SERVICE-FGHIJ..." },
      { "id": "SERVICE-PQRST..." }
    ]
```

```
  }
}
```

Gelen ve giden bağımlılıkların bu yapıya göre ayrıştırılması için aşağıdaki adımlar izlenmelidir:

- **Giden Bağımlılıklar (Outgoing):** Döngüdeki her bir servis nesnesi için, bu nesnenin `entityId` değeri `Source Entity ID` olarak kabul edilir. `fromRelationships.calls.SERVICE` dizisi içindeki her bir `{ "id": "..." }` nesnesinden alınan ID'ler `Target Entity ID` olarak alınır. Bu işlem, `(Kaynak Servis ID, Hedef Servis ID)` şeklinde bir kenar (edge) oluşturur.
- **Gelen Bağımlılıklar (Incoming):** Aynı servis nesnesi için, bu nesnenin `entityId` değeri `Target Entity ID` olarak kabul edilir. `toRelationships.called_by.SERVICE` dizisi içindeki her bir `{ "id": "..." }` nesnesinden alınan ID'ler `Source Entity ID` olarak alınır. Bu işlem de `(Kaynak Servis ID, Hedef Servis ID)` şeklinde bir kenar oluşturur.

Bu ayrıştırma sürecinin nihai hedefi, tüm bu ilişkileri **"Düzleştirilmiş Edge Listesi"** formatında bir araya getirmektir. Bu yapı, her bir bağımlılığı tek bir satırda temsil eder ve CMDB sistemleri, graf veritabanları veya görselleştirme araçları için ideal bir girdi formatıdır.

Source Entity ID	Relationship Type	Target Entity ID
`SERVICE-ABCDE...`	`CALLS`	`SERVICE-FGHIJ...`
`SERVICE-KLMNO...`	`CALLS`	`SERVICE-ABCDE...`

Bu yapılandırılmış veri seti, ham JSON verisini eyleme geçirilebilir bir bilgiye dönüştürür. Bir sonraki bölümde, bu otomasyonu kurumsal bir ortamda güvenli bir şekilde çalıştırmak için gereken güvenlik yapılandırmaları ve operasyonel kısıtlamalar ele alınacaktır.

5. Güvenlik ve Operasyonel Kısıtlamalar

Otomasyon betiğini kurumsal ortamlarda, özellikle de finans gibi regülasyonların yoğun olduğu sektörlerde, güvenli ve sürdürülebilir bir şekilde çalıştırmak, projenin başarısı için en az algoritmanın kendisi kadar önemlidir. Bu bölüm, API token'ı için gerekli olan minimum yetkileri, token'ın güvenli bir şekilde saklanmasına yönelik en iyi uygulamaları ve sistemin genel performansını etkileyebilecek operasyonel kısıtlamaları kapsamaktadır.

Token Yetkileri (Scope) ve En Az Ayrıcalık İlkesi

Güvenli bir otomasyon, "En Az Ayrıcalık İlkesi"ne (Principle of Least Privilege - PoLP) sıkı sıkıya bağlı kalmalıdır. Bu ilke, bir bileşene veya kullanıcıya yalnızca görevini yerine getirmesi için gereken minimum yetkilerin verilmesini gerektirir. Bu proje için oluşturulacak API token'ının sahip olması gereken tek zorunlu yetki şudur:

- **Zorunlu Scope:** `entities.read` (`Read entities`)

Bu yetki, betiğin yalnızca izlenen varlıkların verilerini ve ilişkilerini okumasına izin verir. `entities.write` veya `settings.write` gibi yazma yetkileri, topoloji çıkarma görevi için gereksizdir ve potansiyel bir güvenlik riski oluşturur. Bu nedenle, bu tür yetkiler token'a **kesinlikle** verilmemelidir.

Token Güvenliği ve Saklama Yöntemleri

Dynatrace API token'ının gizli (secret) kısmı, bir parola gibi kabul edilmeli ve en yüksek güvenlik standartlarıyla korunmalıdır. Token'ların öngörülebilir bir formatı vardır (`dt0s01...` ön eki, halka açık bir bölüm ve gizli bir bölümden oluşur), bu da `git-secrets` gibi araçlarla kaynak koda yanlışlıkla dahil edilmelerini önlemek için proaktif güvenlik taramaları yapılmasına olanak tanır. Token'ın güvenli bir şekilde ele alınması için aşağıdaki kurallar kurumsal bir zorunluluktur:

- Token'lar **kesinlikle** kaynak koduna (source code) veya versiyon kontrol sistemlerine (örn. Git) dahil edilen düz metin yapılandırma dosyalarına yazılmamalıdır.
- Kurumsal standartlar gereği, API token'ının **HashiCorp Vault, Azure Key Vault** veya **AWS Secrets Manager** gibi bir sır yönetim sistemi (secrets manager) içinde saklanması bir zorunluluktur. Otomasyon betiği, bu sistemlere entegre edilerek token'ı çalışma zamanında (runtime) güvenli bir şekilde çekmelidir.

Kimlik Doğrulama Başlığı (Authorization Header)

API isteklerinde kimlik doğrulaması yapılırken token'ın URL içinde bir sorgu parametresi olarak gönderilmesi, token'ın loglarda, tarayıcı geçmişinde veya paylaşılan URL'lerde açığa çıkma riskini artırır. Güvenliği en üst düzeye çıkarmak için, token her zaman `Authorization` HTTP başlığı (header) içinde gönderilmelidir.

- **Doğru Format:** `Authorization: Api-Token {TOKEN_DEĞERİ}`

Performans ve Kaynak Planlaması

Dynatrace ortamındaki dakikada 50 istek API limiti, tek bir entegrasyona özel değil, tüm API istemcileri tarafından paylaşılan ortak bir kaynaktır. Yüksek hacimli bir topoloji çekme betiği, paylaşılan kotanın tamamını tüketerek diğer kritik entegrasyonları etkili bir şekilde aç bırakabilir ve **"gürültücü komşu" (noisy neighbor)** problemi yaratarak basamaklı arızalara neden olabilir. Bu, bir Kıdemli Mimarın azaltması gereken geniş bir mimari risktir.

Bu riski yönetmek için aşağıdaki operasyonel öneri dikkate alınmalıdır:

- Topoloji çekme betiği, diğer kritik otomasyonları engellememesi için, mümkünse API kullanımının daha az olduğu **yoğun olmayan saatlerde (off-peak hours)** çalıştırılacak şekilde zamanlanmalıdır. Bu strateji, kaynak çekişmesini en aza indirir ve tüm entegrasyonların sorunsuz çalışmasını sağlar.