T.C. SANAYİ VE TEKNOLOJİ BAKANLIĞI

#MİLLİ TEKNOLOJİ HAMLESİ

TÜBİTAK

TESTCONTAINERS İLE ENTEGRASYON TEST YAZIM TEKNİKLERİ

ARAŞTIRMA SERİSİ - SAYI 9

BİLGEM

YAZILIM TEKNOLOJİLERİ ARAŞTIRMA ENSTİTÜSÜ

Simge ve Kısaltmalar

Kısaltmalar	Açıklama
TÜBİTAK	Türkiye Bilimsel ve Teknolojik Araştırma Kurumu
BİLGEM	Bilişim ve Bilgi Güvenliği İleri Teknolojiler Araştırma Merkezi
YTE	Yazılım Teknolojileri Araştırma Enstitüsü
AWS	Amazon Web Services
URL	Uniform Resource Locator

Yazar
Muhammed Fatih DOĞMUŞ

Yayın Koordinatörü
Elif ŞENYİĞİT

Editörler
Veli Can AYDIN
Sevinç KARAKAŞ
Tuğçe YILMAZ

Tasarım
Şeyma KOÇER

İletişim: 0(312) 289 92 22 - yte.bilgi@tubitak.gov.tr

yte.bilgem@tubitak.gov.tr

İçindekiler

Önsöz

TÜBİTAK BİLGEM Yazılım Teknolojileri Araştırma Enstitüsü (YTE), 2012 yılından bu yana yazılım teknolojilerinde Ar-Ge faaliyetleri yürüten bir araştırma kuruluşudur. Araştırma faaliyetlerinde elde ettiği birikimini stratejik, hassas ve kritik projeler yürüterek kamu adına hayata geçirmekte; kurumlarımıza dijital dönüşüm, yazılım geliştirme teknolojileri ve kalite süreçleri konusunda danışmanlık vermektedir.

TÜBİTAK BİLGEM YTE tarafından hazırlanan Araştırma Serisi ile kurum içi içerik üretme çalışmalarının yaygınlaştırılması ve hazırlanan içeriklerin sektörün erişimine açılması amaçlanmaktadır. Araştırma Serisi'nde yayınlanan çalışmalar TÜBİTAK BİLGEM YTE çalışanlarının projelerde elde ettiği bilgi birikimini paylaşmak adına derlenmiştir. Bu çalışmalar ile ülkemizin yazılım sektörüne katkı sağlanılması hedeflenmektedir.

Giriş

Entegrasyon testleri, bir projenin olmazsa olmaz test türlerindendir. Sistemin bağımsız geliştirilen parçalarını, birbirleriyle entegre bir şekilde test ederek aralarındaki konuşma protokollerinin doğru olduğundan emin olurlar. Entegrasyon testi olmadan, sadece birim testler ile bir projenin hatasız bir şekilde müşteriye sunulması düşünülemez. Birim testler sadece ufak parçaların kendi içlerinde doğru çalıştığından emin olurlar. Bu sebeple entegrasyon testlerinin güvenilir ve tutarlı bir şekilde çalışıyor olması çok önemlidir.

Entegrasyon testleri, doğaları gereği kodun kendisinden olmayan dış faktörlerden çok daha kolay bir şekilde etkilenerek yanlış negatif sonuçlar verebilmektedir. Bu durum zaman içerisinde entegrasyon testlerine olan güveni azaltmakta, bu testlerde çıkan bir hatanın yine kodun kendisinden değil, dış etmenlerden kaynaklandığı varsayılarak testlerin ortaya çıkardığı gerçek hatalar atlanabilmektedir. Kodda hata olmadığına dair güven vermeyen bir test grubu, hiçbir anlam ifade etmemektedir. Yıllar içerisinde bu problemleri çözmek için özel oluşturulmuş yöntemler kullanılsa da hiçbiri Testcontainers kadar kolay kullanım, tutarlılık ve güven sunmamıştır.

Bu çalışmada amaç, entegrasyon testlerini zorluk olmaktan çıkaran Testcontainers kütüphanesinin genel çalışma mantığını, kapsamını, ve java ile yazılan entegrasyon testlerimizde nasıl kullanılacağını göstermek olacaktır.

Geleneksel Yöntem ve Problemleri

Bir sürekli entegrasyon ortamında Testcontainers kullanmadan testler çalıştırılmak istendiğinde yapılması gereken adımlar şunlardır:

1. **Kullanılacak dış bağımlılıkların testlerin çalıştırılacağı sürekli entegrasyon sunucularına elle kurulması:** Bu adımda, ihtiyaç olunan tüm bağımlılıklar, tüm makinelere tek tek kurulmak zorundadır. Tahmin edileceği üzere bağımlılık sayısı arttıkça bu adım zor bir hale gelmektedir.
2. **Testlerin doğru bağımlılıkları kullanmasının sağlaması:** Testler test ortamında çalıştığında, doğru bağlantı parametreleri ile gerekli bağımlılıkların kullanması sağlanmalıdır.
3. **Testlerin çalıştırılması:** Sürekli entegrasyon ortamına atılan testler düzenli aralıklarla çalıştırılmalıdır.
4. **Testler çalıştıktan sonra artık verinin temizlenmesi:** Bu adımda çalıştırılan testlerin oluşturduğu artık veriler temizlenmelidir. Bu sayede sonraki test çalıştırılmasında testler temiz veri ile iş yapabilmektedir.

1. Kullanılacak Dış Bağımlılıkların Kurulma Zorluğu

Entegrasyon testleri çalıştırmadan önce, çalışacak ortamda uygulamanın ayağa kalkabilmesi için gerekli dış bağımlılıkların kurulması gerekmektedir. Bu bağımlılıklar; veri tabanı, mesaj kuyrukları, diğer dış sistemler gibi farklı unsurlar olabilir. Bunların tamamının ayakta olduğundan emin olmak ve sürekli bakımlarını yapmak (örnek: versiyon güncellemesi) geliştiricilerin üzerinde büyük bir yük oluşturmaktadır. Dış bağımlılık sayısı arttıkça bu sorumluluk ve hata yapma oranı da artmaktadır. Her gelen yeni versiyonlar ile bu bağımlılıkların güncellenmesi de ciddi bir bakım maliyeti getirmektedir.

Sürekli entegrasyon aracı birden fazla sunucuya yüklü ise, bunların her birine ayrı ayrı kurulum ve bakım da ekstra yük getirmektedir. Burada yapılacak bir hata, farklı sunucularda farklı versiyonların kullanılmasına, bundan dolayı da versiyon uyumsuzluğu sebebiyle hatalar ortaya çıkmaktadır.

2. Test Verilerinin Temizlenmesi

Her test çalıştırılmasından sonra, mevcut test verilerinin silinmesi gerekmektedir. Bu veriler; veri tabanına yazılan kayıtlar, eklenen dosyalar, mesaj kuyruklarına eklenen mesajlar gibi farklı formatlarda olabilir. Verilerin herhangi bir sebepten dolayı düzgün bir şekilde temizlenememesi, sonraki çalıştırılacak testlerin yanlış veri ile çalışması, hatta çoğu durumda başarısız olmasına sebep olabilmektedir. Geleneksel yöntemlerde veri tutarlılığının sağlanması zor ve fazla emek isteyen bir süreçtir.

3. Testlerin Paralel Çalıştırılamaması

Proje büyüdükçe test sayısı artmaktadır. Entegrasyon testlerin doğası gereği bu testlerin çalıştırılması uzun zaman almaktadır. Bu sebepten dolayı testlerin çalıştırılmasının paralelleştirilmesi kaynakların daha verimli kullanılması ve test sürelerinin kısaltılması için önemlidir. Fakat geleneksel yöntemlerde, bir makinede tek bir veri tabanı, mesaj kuyruğu vb. bulunduğu için; paralelde çalışan testler, birbirlerinin verilerini ezip, diğer testin çalışmasını bozabildiği gibi aynı veriye yazmaya çalışan testlerde çakışma yaşanabilmektedir.

Uzun soluklu bir projede bağımlılıkların yönetilmesi ve veri bütünlüğünün sağlanması, yüksek bir efor harcanmasına ve kolayca yapılabilecek hataların testlerin güvenilirliğini sarsmasına sebep olmaktadır. Özellikle 1. ve 3. adımda bahsedilen zorluklar, alternatif yöntemlerin denenmesini gerektirmektedir.

Testcontainers

Testcontainers, uygulama içerisinde tek kullanımlık Docker konteynerler ayağa kaldırılmasını sağlayan bir kütüphanedir. İlk başta sadece Java dili için kullanıma sunulmuş olsa da sonrasında Go, Node.js, .NET gibi farklı dil ve geliştirme çatılarını da desteklemeye başlamıştır.

Testcontainers'ın asıl çıkış amacı, entegrasyon testlerinin yazımını kolaylaştırmak olsa da bu ana amacının çok ötesine geçmiştir. Testcontainers, verilen konteynerleri ayağa kaldırmak için Docker kullandığından dolayı, Testcontainers kullanan uygulamaların çalıştırıldığı makinelerde Docker yüklü olması bir ön koşuldur.

1. Ana Çalışma Prensibi

Testcontainers, arka tarafta Docker kullanmaktadır. Docker, sanallaştırılmış uygulama konteynerlerini oluşturan, dağıtan ve yöneten bir açık kaynak kodlu yazılım platformudur. Konteyner çalıştırma platformlarından en popüleri olan Docker, Testcontainers'ın ayağa kaldıracağı konteynerleri çalıştırdığı platformdur.

Testcontainers, temelde basit bir prensip üzerine çalışmaktadır. Kurulu olan Docker ile doğrudan iletişim kurarak, yazılan koda göre bazı Docker konteynerler ayağa kaldırılmasını sağlamaktadır. Örnek olarak testler için bir PostgreSQL veri tabanına ihtiyaç varsa, bunu doğrudan uygulama kodu üzerinden Testcontainers'a söylenmesi durumunda, verilen konteyner versiyonu ile bir PostgreSQL konteyneri ayağa kaldırıp, bu oluşturulan konteynere bağlanılabilmesi için gerekli URL ve port bilgilerini vermektedir. Verilen bu bağlantı bilgileri ile de kod içerisinde konteynere rahatlıkla bağlanılabilmektedir.

Bilinmesi gereken diğer bir önemli nokta ise Testcontainers, oluşturduğu konteynerleri rastgele portlarda ayağa kaldırmaktadır. Bunun sebebi ise eğer aynı konteynerden ayağa kaldırmaya çalışan birden fazla uygulama varsa, bu uygulamaların çakışmadan rahatlıkla konteynerlerini ayağa kaldırabilmesini sağlamaktır. Bu yüzden Testcontainers, verilen konteyneri ayağa kaldırdıktan sonra, bağlanılabilmesi için bağlantı bilgilerini vermektedir.

Son önemli nokta ise, Testcontainers tarafından oluşturulan tüm konteynerler, aktif bağlantı bulunmadığı fark edildiği anda otomatik olarak yok edilirler. Bunun için Testcontainers, Testcontainers ile herhangi bir konteyner ayağa kaldırıldığında, kendisine ait Ryuk denen ayrı bir konteyner ayağa kaldırmaktadır. Bu konteyner, Testcontainers'ın yönettiği konteynerleri izlemekte ve eğer bir konteynerin aktif bağlantısı bulunmuyorsa, konteyneri otomatik olarak silmektedir. Testcontainers'ın yönettiği hiçbir konteyner kalmazsa, Ryuk konteyneri da kendini sonlandırmaktadır. Bu sayede herhangi bir şekilde artık veri veya portların düşünülmesine gerek kalmamakta, tüm işi Testcontainers'ın kendisi gerçekleştirmektedir.

2. Desteklediği Diller

İlk olarak sadece Java dilini destekleyerek çıkmış olsa da daha sonrasında Go, .NET, Node.js ve Python gibi pek çok farklı dili destekler hale gelmiştir. Bu sayede kullanılan dilden bağımsız olarak, Testcontainer'ın getirdiği kolaylıklardan faydalanılabilmektedir.

3. Mevcut Entegrasyonlar

Testcontainers, hali hazırda kullanılabilecek bazı modüller ile gelir. Herhangi bir Docker konteynerini çalıştırabilse de, kendisi de bazı sık kullanılan konteynerlere kolay erişim sağlayarak, o konteynere özgü olan belli başlı ayarların yapılmasını da sağlayabilmektedir. Örnek olarak bir Kafka konteyneri ayağa kaldırıldığında, bağlanmak için kullanılması gereken kullanıcı adı ve şifre doğrudan kod üzerinden bu entegrasyon sayesinde verilebilmektedir. Genel çerçevede sağladığı modül entegrasyonları şu şekilde sayılabilir.

- **Veri tabanları:** PostgreSQL, MySQL, MongoDB, MariaDB, OracleDB, Cassandra gibi hem SQL hem NoSQL pek çok veri tabanına birinci elden destek sunmaktadır.
- **Mesaj kuyrukları:** Kafka, RabbitMQ, Redpanda, HiveMQ, Apache Pulsar gibi en sık kullanılan mesaj kuyruklarını doğrudan ayağa kaldırabilmekte ve bu mesaj kuyruklarına özgü ayarlar doğrudan kod üzerinden yapılabilmektedir. Örnek olarak Kafka kullanırken Zookeeper ayarları yapılması gerekirse doğrudan kod üzerinden bu ayarlar yapılabilmektedir.
- **Bulut servisleri:** Bulut servisleri, sunmuş oldukları servisleri yerel ortamda test edebilmek ve geliştirme yapabilmek için, kendi servislerini Docker konteyneri halinde geliştiricilerin hizmetine sunmaktadır. Testlerde de bu servislerin kullanılması gerektiği durumda, Testcontainers'ın sunduğu Microsoft Azure, Google Cloud veya AWS servisleri yerel ortamda çalıştırılabilmektedir. Henüz tam olgunluğa erişmemiş olsa da ürün ortamına en yakın şekilde test edebilmek için önemli bir sistem olarak görülmektedir.

4. Test Geliştirme Çatısı Desteği

Testcontainers, ana olarak Junit 4 ile çalışması için tasarlanmıştır. Fakat Junit 4 üzerine Junit 5 ve Spock çatılarına da destek getirilmiştir. Bu durum Java camiasındaki en yaygın kullanılan test çatıları için destek getirildiği anlamına gelmektedir.

Bu çalışmada, verilecek örnekler Junit 5 ile yapılacaktır.

5. Uygulama Yöntemi

Bu bölümde Testcontainers'ın özelliklerinden ve pratikte nasıl uygulanabileceğinden bahsedilmiştir.

5.1. Konteynerin Ayağa Kaldırılması ve Bağlanılması

```
@Testcontainers [1]
public class RedisBaseContainerTest {

    private final String imageName = “redis:7-alpine”;

    @Container [2]
    private GenericContainer redisContainer = new
GenericContainer(DockerImageName.parse(imageName)) [3]
            .withExposedPorts(6379); [4]

    @Test
    void redis_konteynerine_basariyla_baglanilir() {
        Integer port = redisContainer.getMappedPort(6379);
        String host = redisContainer.getHost();

        RedisURI redisURI = RedisURI
                .builder()
                .withHost(host)
                .withPort(port)
                .withDatabase(0)
                .build(); [5]
        RedisClient redisClient = RedisClient.create(redisURI);

        var connection = redisClient.connect();
        connection.sync().set(“key”,”Hello World”); [6]

        String helloWorld = connection.sync().get(“key”); [7]

        assertThat(helloWorld).isEqualTo(“Hello World”);
    }
}
```

Kod parçasının daha kolay anlaşılabilmesi için aşağıda adım adım üzerinden geçilmiştir.

1. @Testcontainers anotasyonu, üzerine konulan test sınıfının Testcontainers tarafından denetlendiğini ve yönetildiğini belirtmektedir. Bu anotasyon, testin içerisinde yer alan @Container ile işaretlenmiş olan tüm konteynerlerin başlatılması ve durdurulmasını üstlenir.

2. @Container anotasyonu, işaretlenmiş değişkenin bir konteyner tanımı tuttuğunu ve Testcontainers'ın bunu yönetmesi gerektiğini belirtir. Not: Bu değişkenin tuttuğu değer, Testcontainers'ın başlatabileceği bir sınıf tipinde olmalıdır.

3. Burada, konteynerinin tanımı yapılmaktadır. Testcontainers'ın kendisinin yönetmediği modüllerden bir docker imajı kullanılması gerekiyorsa, GenericContainer sınıfı, herhangi bir docker imaj ismi alabilmektedir. DockerImage.parse metoduna kullanılmak istenen docker imajının ismi verilip, bunu da GenericContainer sınıfının constructor'ına verildiğinde, Testcontainers bu imajı otomatik olarak çekip, ayağa kaldırmaktadır.

4. Testcontainers, kendi yönettiği modüllerdeki bağımlılıkların, hangi portlarının açık olduğunu bilir. Örnek olarak bir PostgreSQL konteyneri ayağa kaldırıldığında, ana dinlenen 5432 portunun açık olması gerektiğini bildiği için bunu otomatik olarak dış dünyaya açar. Fakat burada Testcontainers'ın bilmediği bir imaj kullanıldığı için, konteyner içerisindeki hangi portların dışarıya açılması gerektiğini Testcontainers'a bildirmek gerekmektedir. Redis varsayılan olarak 6379 portunda ayağa kalktığı için, bu satır içerideki bu portun dışarıya açılmasın belirtir.

5. Burada Lettuce denen bir kütüphane ile Redis bağlantısı yapılmaktadır. Burada verilen host ve port değerleri, doğrudan konteynerin kendisi üzerinden alınmalıdır. Çünkü daha önce de bahsedildiği gibi, diğer ayağa kaldırılacak konteyner portları ile çakışma olmaması için dışarıya açılan portlar, host makinesindeki rastgele bir port ile eşleştirilir.. Host makinesinde hangi portun açıldığı bilinmediği için, konteyner üzerindeki getMappedPort fonksiyonu ile konteyner içerisinde açılmış olan port'un, host makinesinde gerçekte hangi port'a denk geldiği alınmaktadır.

6. Lettuce'dan alınan bağlantı ile redis konteynerine bir anahtar ve değer atanır.

7. Aynı bağlantı üzerinde konulan anahtarın değeri okunur ve konulan değerle aynı olduğu doğrulanır.

Görüldüğü üzere kullanılması gereken konteynerleri ayağa kaldırmak çok kolaydır. Uygulamaların isterlerine göre bu ayağa kalkan konteynerler rahatlıkla kullanılabilir. GenericContainer ile herhangi bir docker imajı ayağa kaldırılıp yönetilebilmektedir. Fakat Testcontainers, bazı sık kullanılan imajları kendisi yönetir. Bunlarla beraber de o konteyner sınıfı üzerinde konteynere ait sık yapılan ayarları, doğrudan yapabilmeyi sağlamaktadır.

5.2. Testcontainer'ın Yönettiği İmajlar ile Konteyner Yönetimi

Eğer kullanılmak istenilen docker imajı, Testcontainers'ın yönettiği bir imaj ise, Testcontainers gerekli bağımlılığı uygulamaya ekleyip, doğrudan gerekli konteyner sınıfını kullanılabilmektedir. Bu, normal genel konteyner kullanımına ek, sınıf üzerinde kullanılabilecek faydalı fonksiyonlar vermektedir. Bir örnek aşağıda sunulmuştur.

```
@Testcontainers
public class PostgreSQLTestcontainersTest {

    DockerImageName imageName = DockerImageName.parse("postgres:14.5");

    @Container
    PostgreSQLContainer postgreSQLContainer = new PostgreSQLContainer(imageName)
                        .withDatabaseName("test-db")
                        .withUsername("username") [1]
                        .withPassword("password");

    @Test
    void baglanti_basariyla_alinir() throws SQLException {
        Connection connection = DriverManager.getConnection(
                postgreSQLContainer.getJdbcUrl(), [2]
                "username",
                "password"
        );
    }
}
```

Verilen örnek diğerine çok benzemektedir. Fakat diğer örnekten farklı olarak, genel konteyner sınıfını kullanmak yerine, doğrudan PostgreSQL konteyner sınıfı kullanılmaktadır. Bu durum bazı avantajlar kazandırmaktadır.

1. Sık kullanılabilecek veri tabanı ismi, bağlanmak için kullanılacak kullanıcı adı ve şifre gibi ortam değişkenleri, doğrudan konteyner sınıfı üzerinden ayarlanabilmektedir. Bunun avantajı, kullanılan her konteynerin ortam değişkenlerinin ne olduğuna tek tek bakmak yerine, doğrudan sınıfın üzerindeki fonksiyonlarla değiştirilebilmesidir. Bu yöntem kullanılmadığında, docker imajının aldığı ortam değişkenleri araştırılıp tek tek verilmesi gerekmektedir.
2. Burada da yine PostgreSQLContainer sınıfına özgü getJdbcUrl fonksiyonunu kullanarak, bağlantı adresini elle oluşturmak yerine bu işin Testcontainers'a devredilmesi sağlanmıştır.

5.3. Konteynerlerin Yeniden Kullanımı

Testcontainers varsayılan ayarlarında, oluşturduğu konteynerleri her test için yeniden oluşturmaktadır. Fakat bu, testlerin çok yavaş olmasına sebep olabilir. Genellikle, bir test

grubunun (test suite) beraber çalışacağı varsayılırsa, bir test grubunda kullanılacak konteynerin aynı olmasında bir sakınca olmayacaktır. Yazılan verileri geri alarak, veri yönetimi testin içerisinde yapılabilir ve böylelikle testlerin daha hızlı çalışması sağlanabilir. Bunun için tek yapılması gereken, konteynerin tanımlandığı değişkeni static yapmaktır.

Kullanılan konteyner değişkeninin static yapıldığı durumda, aynı test grubu içerisinde çalışan tüm testler için, Testcontainers aynı konteyneri kullanacaktır. Fakat bu yöntem daha da iyileştirilebilir. Genellikle, belli bir test grubunda, aynı konteynerler kullanmak gerekir. Çünkü test mantığı aynı olduğunda, farklı farklı konteynerler kullanmak anlamsız olmaktadır. Bu aşamada, singleton konteynerler devreye girmektedir.

5.4. Singleton Konteynerler

Singleton konteynerler, tanımlandıkları sınıfa kalıtım yapılan (inheritance) tüm alt sınıflardaki test grupları tarafından ortak kullanılmaktadır. Örneğin veri tabanına erişen tüm testleri, aynı konteynerin kullanması sağlanabilir. Böylelikle her test grubu için ayrı ayrı ayar yapılmasına ve daha da önemlisi her test grubu için ayrı ayrı konteyner ayağa kaldırılmasına ihtiyaç kalmaz. Bunu yapmak için diğer test gruplarının kalıtım yapabileceği bir ana sınıfa ihtiyaç bulunmaktadır.

```
@Testcontainers
public abstract class BaseDatabaseTest { [1]

    static DockerImageName imageName = DockerImageName.parse(“postgres:14.5”);


    static PostgreSQLContainer pgSQLContainer = new
PostgreSQLContainer(imageName) [2]
                        .withDatabaseName(“test-db”)
                        .withUsername(“username”)
                        .withPassword(“password”);

    static { [3]
        pgSQLContainer.start();
    }
}
```

1. Sınıfın bir objesinin oluşturulamaması için soyut yapılmıştır, böylelikle herhangi bir objesinin oluşturulması engellenmiş olur. Ayrıca konteynerin yaşam döngüsü elle kontrol edileceği için, @Testcontainers annotation’ına da artık ihtiyaç kalmamıştır.
2. Konteynerin tanımı aynıdır, fakat konteyner elle başlatılacağı için başına @Container koyulmamalıdır.
3. Burada konteyner başlatılmaktadır. Singleton konteynerlerin yaşam döngüsünü Testcontainers’ın kendisi yönetmemektedir. Bu yüzden de static bloğu içerisinde konteynerin

elle başlatılması gerekir. Burada önemli nokta ise konteynerin işi bittiğinde elle durdurulmasına ihtiyacı yoktur. Daha önce de bahsedildiği gibi yardımcı bir konteyner olan Ryuk konteyneri, kullanılan konteynerin işi bittiğinde kendiliğinden temizlik yapmaktadır. Bu sayede elle temizlik yapılmasına gerek kalmaz.

Bu sınıfımızı kalıtan tüm sınıflar, aynı konteyneri kullanabilirler. Eğer oluşan verileri temizlemeye özen gösterilirse, test çalıştırma hızında çok ciddi artışlar olacaktır.

5.5. Testlerde Spring Boot ile Entegrasyon

Eğer Spring Boot kullanılıyorsa, entegrasyon testlerinde ayağa kaldırılan bu konteynerlere bir şekilde bağlanılması gerekmektedir. Uygulamalarda bu ayar, application.properties dosyasından verilir, fakat Testcontainers ayağa kaldırdığı bağımlılığın portunu rastgele bir şekilde atadığı için, sabit bir şekilde bu bilgiye ulaşılması mümkün değildir. Bu sebeple, bağımlılığa olan bağlantı bilgilerinin, dinamik bir şekilde konteyner üzerinden alınıp Spring'e verilmesi gerekmektedir. Bunun için de Spring Boot'un vermiş olduğu @DynamicPropertySource anotasyonu kullanılabilir. Bu anotasyon, dinamik bir şekilde Spring'e ait özellikleri (properties) değiştirmeyi ve eklemeyi sağlamaktadır.

Örnek olarak bir PostgreSQL konteyneri ayağa kaldırıp bunu Spring Boot tabanlı testlerimizde kullanılması gerekirse, aşağıdaki gibi bir kullanım yapılabilir.

```
public class PostgreSqlContainerTest {

    static PostgreSQLContainer postgres = new PostgreSQLContainer();

    static {
        postgres.start();
    }

    @DynamicPropertySource
    static void properties(DynamicPropertyRegistry r) {
        r.add("spring.datasource.url", postgres::getJdbcUrl);
        r.add("spring.datasource.username", postgres::getUsername);
        r.add("spring.datasource.password", postgres::getPassword);
    }
}
```

Bu şekilde yapıldığında, Spring Boot'un, veri tabanına bağlanması için gerekli bilgileri ayarlanmış olmaktadır. Aslında burada olan, dinamik bir şekilde Spring'e ait özellikleri (properties) ayarlamaktır. Tıpkı bu bilgiler application.properties'e yazılmış gibi, Spring Boot tarafında okunarak otomatik olarak veri tabanına olan bağlantı yapılır. Bu sayede testlerde, ayağa kalkan konteynere otomatik olarak bağlanılabilmektedir.

Sonuç ve Öneriler

Geleneksel yöntemleri iyi bilen kişiler, Testcontainers'ın getirdiği kolaylıkları rahat bir şekilde görecektir. Tüm makinelere ayrı ayrı tüm bağımlılıkları kurmaktan kurtulmak, bu bağımlılıkların versiyonlarını koddan rahatlıkla güncelleyebilmek, artık veriler ve çakışan portlar ile uğraşmamak, entegrasyon testlerinin çalıştırılmasını çok ciddi anlamda kolaylaştırmaktadır. Eskiden entegrasyon testlerinin yazımı zor olduğu için birim testlerine ağırlık verilmesi tavsiye edilirdi. Fakat Testcontainers ile birlikte, pek çok kişinin, daha fazla entegrasyon testi yazılmasını tavsiye ettiği görülmektedir. Bu kesin bir yargı olmamak ile beraber, geleneksel test piramidinin yeniden gözden geçirilmesi gerektiği açıktır. Her ne kadar entegrasyon testlerinde hız bir problem olarak devam etse de yazım ve çalıştırılması yönündeki problemler çok azaldığı için daha az mock kullanarak daha fazla entegrasyon testinin yazılması değerlendirilmesi gereken bir yön olduğu görülmektedir.

Son zamanlarda yaptığı atılımlarla, sadece Java ekosistemini değil, diğer popüler C#, Go, Node.js, Python gibi arka yüz programlama ekosistemlerini de destekleyerek, tüm geliştiricilerin entegrasyon testlerini yazmayı kolaylaştırmayı hedefleyen Testcontainers, kesinlikle teknolojik alet çantanızda olmazsa olmaz araçlardan birisidir.

Kaynakça

1. https://testcontainers.com
2. https://docs.docker.com

T.C. SANAYİ VE
TEKNOLOJİ BAKANLIĞI

#MİLLİ TEKNOLOJİ HAMLESİ

TÜBİTAK

İşçi Blokları Mahallesi Muhsin Yazıcıoğlu Caddesi No:51/C 06530 Çankaya/ANKARA

+90 (312) 289 92 22 - yte.bilgi@tubitak.gov.tr

TÜBİTAK - BİLGEM Yazılım Teknolojileri Araştırma Enstitüsü (YTE)