

## Entegrasyon Testleri

### 1. Entegrasyon Testinin Amacı ve Kapsamı

Entegrasyon testi, sistemin birbiriyle etkileşen parçalarının birlikte doğru çalışıp çalışmadığını sınar. Birim test, tek bir sınıf ya da fonksiyonun iç mantığını izole biçimde doğrularken; entegrasyon testi, bu bileşenlerin sınırlarında ortaya çıkabilecek veri uyumsuzluğu, sözleşme ihlali ve yan etki problemlerini ortaya çıkarır. HTTP katmanından veri katmanına kadar uzanan çağrı zincirinin beklenen sonucu üretip üretmediği, hataların doğru ele alınıp alınmadığı ve transaction sınırlarının doğru işletildiği bu seviyede görünür olur.

### 2. Entegrasyon Testi Stratejileri (Şekillendirilmiş Özet)

Aşağıdaki çerçeve, farklı stratejilerin hangi koşullarda tercih edilmesi gerektiğini somutlaştırır.

- Big Bang: Tüm modüller aynı anda çalıştırılır; hızlı başlamak için uygundur. Hata yeri tespiti zordur. Mimari henüz oturmamışsa önerilmez.
- Top-Down: Üst katmanlar (ör. controller/servis) gerçek, alt katmanlar (ör. veri/harici servis) başlangıçta stub/contract testleri ile temsil edilir; sonra gerçek bileşenler eklenir. Karmaşık iş kurallarını erken doğrulamak için uygundur.
- Bottom-Up: Önce veri ve altyapı bileşenleri gerçekçi şekilde doğrulanır, ardından üst katmanlar bağlanır. Kalıcı katmanların davranışı kritik olduğunda tercih edilir.
- Sandwich (Karma): Üstten ve alttan paralel ilerlenir, orta katmanda buluşulur. Büyük ekiplerde, bağımlı akışlar ayrıştırılmak istendiğinde etkilidir.

Seçim Kriterleri: Modüller arası bağımlılığın erişilebilirliği, harici sistemlere erişim maliyeti, veri şemalarının uygunluk düzeyi, test süresine ilişkin kısıtlar ve CI politikası.

**Big Bang** modelinde temel fikir, modüllerin tek tek değil tamamının (veya çok büyük bir alt kümesinin) aynı anda entegre edilmesidir. Yani her modülün ayrı ayrı doğrulaması yapılmadan, sistem bir bütün olarak ayağa kaldırılır. Bu durumda:

#### 1. Tüm modüller derlenir ve deploy edilir.

Genellikle CI/CD boru hattında (örneğin Jenkins, GitHub Actions ya da GitLab CI) “integration” aşaması tek bir adımda tüm microservice’leri veya modülleri ayağa kaldırır.

Örnek: docker-compose up ile bütün servisleri, veritabanlarını ve API’leri aynı anda başlatmak.

2. *Test runner veya framework bütün testleri tek seferde yürütür.*

Örneğin:

- Java/Spring projelerinde mvn verify veya gradle test komutu, Surefire/Failsafe gibi eklentilerle tüm test sınıflarını aynı anda koşturur.
- Python'da pytest kullanılıyorsa pytest tests/ --maxfail=1 -v komutu tüm testleri (unit + integration) topluca yürütür.
- Frontend tarafında Cypress, Playwright, Jest gibi araçlar aynı prensiple “full regression suite” modunda tüm testleri paralel veya ardışık biçimde çalıştırabilir.

3. *Çoğunlukla container tabanlı orkestrasyon kullanılır.*

Big Bang modelinde servislerin hepsi birden kalktığı için:

- Docker Compose veya Kubernetes manifestleriyle tüm bileşenler aynı anda çalıştırılır.
- Testler bu canlı sisteme karşı yürütülür (örneğin HTTP, gRPC veya MQ üzerinden).
- Jenkins pipeline'ında bu genellikle şöyle görünür:

```
stage('BigBang Integration') {  
  
    steps {  
  
        sh 'docker compose up -d'  
  
        sh './mvnw verify -Pintegration'  
  
    }  
  
    post {  
  
        always { sh 'docker compose down' }  
  
    }  
  
}
```

4. *Avantajı ve riski:*

- Avantaj: Gerçek sisteme çok yakın, uçtan uca bir doğrulama sağlar.
- Dezavantaj: Hatanın hangi modülden kaynaklandığını izole etmek zordur; çünkü hepsi birden çalışmaktadır.

**Top-Down** entegrasyon testi modeli, Big Bang'ın tam tersine daha planlı ve kademeli ilerleyen bir yaklaşımdır.

Bu yöntemde amaç, sistemi bir anda birleştirip kaotik bir test yapmak yerine en üst seviyeden başlayıp aşağıya doğru adım adım test etmektir.

### Temel Mantık

1. **Üst seviye modüller (örneğin Controller veya UI)** önce test edilir.

Ancak bu aşamada alt seviye bileşenler (Service, Repository, Database vb.) henüz hazır olmayabilir.

2. Alt modüller henüz geliştirilmemişse, onların yerine **“stub”** adı verilen geçici taklit bileşenler kullanılır.

Stub'lar, alt modülün beklenen davranışını simüle eder; örneğin sabit bir JSON yanıtı döner veya belirli bir değeri hesaplar.

3. Alt modüller geliştikçe, test süreci adım adım derinleştirilir.

Her eklenen gerçek modül, ilgili stub'un yerini alır ve entegrasyon zinciri aşağıya doğru genişler.

4. Tüm katmanlar (Controller → Service → Repository → Database) tamamlandığında sistem artık uçtan uca test edilebilir hâle gelir.

Bir “sipariş oluşturma” akışını düşünelim:

- Controller → OrderService → PaymentService → Database

Top-Down yaklaşımda:

- İlk aşamada sadece Controller test edilir. OrderService bir **stub** olarak davranır:

```
class OrderServiceStub implements OrderService {  
  
    public OrderResponse createOrder(OrderRequest req) {  
  
        return new OrderResponse("STUB_ONAYLANDI");  
  
    }  
  
}
```

- İkinci aşamada OrderService gerçek hale gelir ama PaymentService hâlâ stub'tur.
- Son aşamada tüm servisler gerçektir; test artık veritabanına kadar ulaşır.

### *Avantajlar*

- Hatalar erken aşamalarda yakalanır (özellikle kontrol akışı hataları).
- Test süreci yukarıdan aşağıya sistematik olarak ilerler, mimari hiyerarşiyi yansıtır.
- UI veya API seviyesinde kullanıcı senaryoları erkenden doğrulanabilir.

**Bottom-Up** modelinde test süreci altyapıdan başlar:

veritabanı, veri erişim katmanı (repository/DAO), servisler ve en son kontrol katmanı (API/UI) entegrasyon testine dahil edilir. Yani önce temel taşların sağlamlığından emin olunur, ardından üst seviyeler bu yapı üzerine entegre edilir.

### **Adım Adım Süreç**

1. *Alt modüller (örneğin repository veya veri erişim sınıfları) test edilir.*

Bu aşamada gerçek bir veritabanı ya da Testcontainers ile izole edilmiş bir veritabanı kullanılır.

Örneğin UserRepository veya OrderRepository testleri burada yapılır.

2. *Servis katmanı eklenir.*

Bu servisler veri katmanına erişir; testlerde veri erişimi artık gerçek repository üzerinden yapılır, ama üst katmanlar (Controller) henüz dahil edilmemiştir.

3. *Üst katman (Controller / API) entegrasyonu yapılır.*

Tüm alt katmanlar artık test edilmiş ve güvenilirdir, bu nedenle API seviyesinde testler daha kararlı hale gelir.

4. *Sistemin tamamı en son birleştirilir (bazen bu noktada Big Bang'in mini versiyonu gibi düşünülür) ve uçtan uca test yapılır.*

### **Basit Bir Örnek**

Bir öğrenci yönetim sistemini düşünelim:

- Repository katmanı: StudentRepository, CourseRepository
- Servis katmanı: StudentService
- Controller: StudentController

Bottom-Up modelinde:

- İlk olarak StudentRepository test edilir:

```

@DataJpaTest

class StudentRepositoryTest {

    @Autowired StudentRepository repo;

    @Test

    void kayitBasariliMi() {

        Student s = new Student("Ali", "Okumus");

        repo.save(s);

        assertThat(repo.findAll()).hasSize(1);

    }

}

```

- Daha sonra StudentService, gerçek repository kullanılarak test edilir:

```

@SpringBootTest

class StudentServiceIntegrationTest {

    @Autowired StudentService service;

    @Test

    void ogrenciKaydiBasariliMi() {

        service.addStudent("Fatih", "Okumus");

        assertThat(service.listAll()).isNotEmpty();

    }

}

```

- Son olarak StudentController, MockMvc ile uçtan uca test edilir:

```

@SpringBootTest

@AutoConfigureMockMvc

class StudentControllerIT {

    @Autowired MockMvc mockMvc;

    @Test

    void postRequestKayitYapar() throws Exception {

        mockMvc.perform(post("/students")

            .contentType(MediaType.APPLICATION_JSON)

            .content("{\"firstName\":\"Yunus\", \"lastName\":\"Cengiz\"}"))

        .andExpect(status().isCreated());

    }

}

```

### Avantajlar

- Temel güven: Alt seviye bileşenler önce test edildiği için sistemin temeli sağlam olur.
- Hata izolasyonu kolaydır: Üst katman testlerine geçmeden önce veri erişim sorunları elenir.
- Doğal CI uyumu: Veri katmanından başlayan testler otomatik pipeline’larda kolayca paralelleştirilebilir.

### Dezavantajlar

- Üst katman testleri geç başlar, bu yüzden sistemin davranışını bütünsel olarak geç görürüz.
- Kullanıcı senaryolarını test etmek, tüm alt katmanlar tamamlanana kadar mümkün değildir.
- Eğer servisler arasında güçlü bağımlılıklar varsa, ilerleme yavaş olabilir.

**Sandwich** modelinde test süreci hem üst katmandan (örneğin Controller veya UI) hem de alt katmandan (örneğin Repository veya Database) eşzamanlı başlatılır. İki yönlü test akışı sistemin orta katmanında (Service / Business Logic) buluşur. Böylece hem kullanıcı seviyesinde (yukarıdan aşağıya) hem de veri seviyesinde (aşağıdan yukarıya) test yapılmış olur. Amaç, geliştirme süresini kısaltmak ve hataları iki yönden de erken yakalamaktır.

### *Nasıl Çalışır?*

#### 1. Top-Down hattı:

Üst modüller (ör. Controller) alt modüller (ör. Service) için geçici stub bileşenler kullanır. UI veya API akışları, henüz tamamlanmamış servislerin taklitleriyle test edilir.

#### 2. Bottom-Up hattı:

Alt modüller (ör. Repository, Database) kendi testlerini gerçekleştirir; üst modüller (ör. Service, Controller) henüz hazır değilse, onların yerinde driver bileşenleri bulunur. Driver, üst katmanın çağrılarını simüle eder.

#### 3. Orta katmanda birleşme:

Hem stub'lar hem driver'lar, orta katmanda (örneğin Service) yerini gerçek bileşenlere bırakır. Böylece sistem hem yukarıdan hem aşağıdan entegre olmuş olur.

### *Örnek Senaryo*

Bir e-ticaret sistemi düşünelim:

UI/Controller  $\longleftrightarrow$  OrderService  $\longleftrightarrow$  Repository  $\longleftrightarrow$  Database

Sandwich modelinde süreç şöyle ilerler:

- Top-Down kısmı:

Controller testleri başlar. OrderService henüz hazır değilse, onun yerine OrderServiceStub devrededir. Kullanıcı isteklerinin (örneğin /api/orders) doğru işlendiği test edilir.

- Bottom-Up kısmı:

Repository testleri gerçek veritabanı (örneğin H2 veya PostgreSQL Testcontainers) ile yapılır. OrderService çağrıları, henüz gerçek servis yazılmadığı için OrderServiceDriver aracılığıyla tetiklenir.

- Orta aşamada:

OrderService geliştirildiğinde hem stub hem driver kaldırılır, iki yönlü test zinciri birleşir. Artık uçtan uca test yapılabilir hale gelir.

### *Avantajlar*

- Paralel geliştirme mümkündür: Üst ve alt ekipler bağımsız çalışabilir.
- Erken hata tespiti: Hem kullanıcı akışları hem de veri katmanı erken doğrulanır.
- Zaman tasarrufu: Tam sistem hazır olmadan, iki uçtan da entegrasyon testleri yapılabilir.

## Dezavantajlar

- Koordinasyon gerektirir: Stub ve driver'ların sözleşmeleri uyuşmazsa orta katman birleşmesinde çatışmalar yaşanabilir.
- Test karmaşıklığı: Hem yukarıdan hem aşağıdan ilerlediği için test senaryolarının yönetimi daha zordur.
- Kaynak kullanımı: İki yönlü test ortamını aynı anda çalıştırmak daha fazla kaynak gerektirir.

## Uygulama Örneği

Spring Boot + Jenkins tabanlı bir karma test süreci aşağıdaki gibi yapılandırılabilir:

```
pipeline {
  agent any
  stages {
    stage('Top-Down Stubs') {
      steps {
        sh './mvnw test -Ptopdown'
      }
    }
    stage('Bottom-Up Drivers') {
      steps {
        sh './mvnw test -Pbottomup'
      }
    }
    stage('Merge Integration') {
      steps {
        sh './mvnw verify -Pmerged'
      }
    }
  }
}
```

- topdown profili: MockMvc ve Stub'lar ile controller testleri.
- bottomup profili: Testcontainers + Repository testleri.
- merged profili: Gerçek servisler ile uçtan uca entegrasyon.

## 3. Test Ortamı ve Araçlar

- Spring Boot Test çerçevesi: @SpringBootTest ile gerçek Bean grafiğini başlatır; HTTP yüzeyi için MockMvc veya reaktif uygulamalarda WebTestClient kullanılabilir.

- Veritabanı: Hızlı geri bildirim için H2; üretime yakın doğrulama için Testcontainers + PostgreSQL/MySQL. JDBC URL'sinde jdbc:tc: kısıyolu ile konteyner yaşam döngüsü testlere bağlanır.

- Harici Servis Simülasyonu: WireMock ile HTTP sözleşmelerini taklit ederek başarılı/başarısız yanıtlar üretmek; üçüncü parti sistemleri izole etmek.

- Build/Koşturma: Maven/Gradle; Surefire (birim test) ve Failsafe (entegrasyon test) ayrımı; CI'da profile veya adlandırma ile (\*IT.java) ayırıştırma.

- Gözlemlenebilirlik: Test sırasında log seviyelerini artırma (Spring Web/SQL), request/response loglama, SQL açıklama (hibernate.show\_sql=false, ancak org.hibernate.SQL=DEBUG gibi selektif loglama) ve test raporlama (JaCoCo, Surefire/Failsafe raporları).

#### 4. Kod Düzeyinde Uçtan Uca Örnek

Aşağıdaki örnek, HTTP katmanından başlayıp servis ve veri katmanına inen bir akışı sınar. Başarılı ödeme senaryosunda sipariş durumunun "ONAYLANDI" olmasını ve veritabanına yazılmasını bekler.

```
@SpringBootTest(webEnvironment = SpringBootTest.WebEnvironment.RANDOM_PORT)
@AutoConfigureMockMvc
class OrderIntegrationTest {
    @Autowired MockMvc mockMvc;
    @Autowired OrderRepository orderRepository;

    @Test
    void siparisOlustur_ÖdemeBasarili_VeritabaninaYazilir() throws Exception {
        String body = """
            { \"cartId\": \"C123\", \"amount\": 199.90, \"payMethod\": \"CREDIT_CARD\" }
            """;

        mockMvc.perform(post(\"/api/orders\")
            .contentType(MediaType.APPLICATION_JSON)
            .content(body))
            .andExpect(status().isCreated())
            .andExpect(jsonPath(\"$.status\").value(\"ONAYLANDI\"));

        var kayitlar = orderRepository.findAll();
        assertThat(kayitlar).hasSize(1);
        assertThat(kayitlar.get(0).getStatus()).isEqualTo(OrderStatus.ONAYLANDI);
    }
}
```

#### 5. Transaction Yönetimi: @Transactional Örnekleri

Transaction davranışını testlerde iki farklı biçimde göstermek yararlıdır. Varsayılan rollback ile izolasyon sağlanır; commit gerektiren durumlar ayrıca ele alınır.

```
// 5.1 Rollback ile izole test
@SpringBootTest
@Transactional // test sonunda otomatik rollback
class UserServiceTxIsolationTest {
    @Autowired UserService userService;
    @Autowired UserRepository userRepository;

    @Test
    void createUser_RollbackSonrasiVeritabaniTemizKalir() {
        userService.createUser(\"ali@example.com\");
        assertThat(userRepository.count()).isEqualTo(1);
        // Test bittiğinde rollback olacağı için kalıcı değişiklik yok
    }
}
```

```
// 5.2 Commit gereken senaryo
@SpringBootTest
class UserServiceTxCommitTest {
    @Autowired UserService userService;
    @Autowired UserRepository userRepository;

    @Test
    @Commit // veya @Transactional(propagation = Propagation.REQUIRES_NEW)
    @Transactional
    void createUser_CommitSonrasiVeriKaliciOlur() {
        userService.createUser("veli@example.com");
    }

    @Test
    void commitSonrasiVeriGorunur() {
        // Önceki test commit ettiği için sayı artmış olmalı
        assertThat(userRepository.existsByEmail("veli@example.com")).isTrue();
    }
}
```

**Not:** Commit eden testlerin birbirine bağımlı hâle gelmemesi için veri temizliği stratejisi belirleyin (ör. @Sql ile öncesi/sonrası scriptleri, veya her testte benzersiz test verisi).

## 6. Jenkins ile Sürekli Entegrasyonda Entegrasyon Testleri

Aşağıdaki Declarative Pipeline örneği, birim ve entegrasyon testlerini ayrı aşamalarda koşturur. Testcontainers kullanacaksanız Jenkins ajanında Docker erişimi bulunmalıdır.

```
pipeline {
    agent any
    tools { jdk 'temurin-17' } // Jenkins'te tanımlı JDK ismi

    options {
        timestamps()
        ansiColor('xterm')
    }

    stages {
        stage('Checkout') {
            steps { checkout scm }
        }

        stage('Build & Unit Tests') {
            steps { sh './mvnw -B -DskipITs=true clean test' }
            post {
                always {
                    junit 'target/surefire-reports/*.xml'
                    publishHTML(target: [allowMissing: true, alwaysLinkToLastBuild: true,
                        reportDir: 'target/site', reportFiles: 'jacoco/index.html', reportName: 'JaCoCo'])
                }
            }
        }

        stage('Integration Tests') {
            environment {
                TESTCONTAINERS_RYUK_DISABLED = 'true' // Kurumsal ağlarda gerekli olabilir
            }
            steps {
                sh './mvnw -B -DskipITs=false verify'
            }
            post {
                always {
                    junit 'target/failsafe-reports/*.xml'
                }
            }
        }
    }
}
```

```

    }
  }
}

post {
  always { archiveArtifacts artifacts: 'target/*.jar', fingerprint: true }
}
}

```

**Maven Yapılandırma İpucu:** Entegrasyon testlerini Failsafe ile ayırmak için maven-failsafe-plugin kullanıp test sınıflarını \*IT.java olarak adlandırabilirsiniz. Birim testler Surefire, entegrasyon testleri Failsafe tarafından koşturulur.

```

<build>
  <plugins>
    <plugin>
      <groupId>org.apache.maven.plugins</groupId>
      <artifactId>maven-surefire-plugin</artifactId>
      <version>3.2.5</version>
      <configuration>
        <includes>
          <include>**/*Test.java</include>
        </includes>
      </configuration>
    </plugin>
    <plugin>
      <groupId>org.apache.maven.plugins</groupId>
      <artifactId>maven-failsafe-plugin</artifactId>
      <version>3.2.5</version>
      <executions>
        <execution>
          <goals>
            <goal>integration-test</goal>
            <goal>verify</goal>
          </goals>
          <configuration>
            <includes>
              <include>**/*IT.java</include>
            </includes>
          </configuration>
        </execution>
      </executions>
    </plugin>
  </plugins>
</build>

```