Web Programlama I

Ders 04: EJB

Slaytlar Hakkında

BU SLAYTLAR OLDUKÇA YÜKSEK SEVİYEDE GENEL BİR BAKIŞ AÇISI SUNMAKTADIR

DETAYLAR GİT REPOSİTORY'SİNDEKİ YORUM SATIRLARINDA BULUNMAKTADIR

Enterprise Java Bean (EJB)

Bir EJB özel bir tag ile annotate edilmiş Java sınıfıdır

- *@Stateless*, *@Stateful* and *@Singleton*

JEE container'ında (*WildFly*, *GlassFish*, vs) bir EJB çalıştığında container özel fonksiyonlarla iyileştirmelerde bulunur

Örnek: varsayılan olarak, her bir EJB metodu transaction içerisinde yürütülür

- Böylelikle, EntityManager'da *begin()* ve *commit()* çağırmaya gerek kalmaz
- EJB basmakalıp yapıları azaltır

EJB İyileştirmeleri

- JEE EJB iyileştirmeleri 2 ana özelliği temel almaktadır
- Yalnızca JEE için değildir
- *Dependency Injection*: Container EJB'nin ihtiyaç duyduğu bağımlılıkları otomatik olarak ekler
- *Proxy Class*: Container EJB örneğini dönmez, iyileştirilmiş fonksiyonelliklerle birlikte bir subclass döner

Reflection ile Dependency Injection

- “*em*” için, constructor’da bir girdi veya bir setter bulunmaz
- JEE container mevcut «em»’yi otomatik olarak inject eder
- EJB yalnızca dependency alan olarak belirtilmelidir. Nasıl oluşturulduğu ve nasıl inject edildiği görevi ise container’a aittir

- **@Stateless**

```
public class UserBean {

    @PersistenceContext
    private EntityManager em;

    public UserBean(){}
```

Java Reflection

- Java'da (yalnızca JEE'de değil) her bir nesne örneği bildirilen sınıf hakkında bir bilgi tutar
- Sınıfın bilgileri çalışma zamanında sorgulanabilir:
  - metotlar, alanlar, annotationlar, vs.
- Alanlar reflection ile değiştirilebilir, hatta private olsa bile…
- … buna ihtiyaç duyan bir kütüphane yazmıyorsanız bu asla yapmamanız gereken bir şeydir (ör bir JEE container'ı, veya JSON/XML verisini (un)marshall yapma)

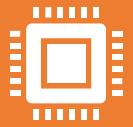
Proxy Class

- Proxy container tarafından otomatik olarak oluşturulacaktır

```
public class Foo {

    public String someMethod(){
        return "foo";
    }
}
```

```
public class FooProxy extends Foo{
    private final Foo original;

    public FooProxy(Foo original) {
        this.original = original;
    }

    @Override
    public String someMethod(){
        // öncesinde bir şeyler yap, ör transaction başlat
        String result = original.someMethod();
        //sonra bir şeyler yap, ör, transaction'ı commit et
        return result;
    }
}
```

Proxy Classes Oluşturma



Derleme zamanında bir proxy sınıf olmayacağından, aslında oldukça karmaşıktır.

Proxy sınıf çalışma zamanında bytecode manipülasyonu ile oluşturulur

Java SE (EE değil) API, proxy sınıfları oluşturmak için bazı temel işlevler sağlar, ancak bunlar yalnızca concrete sınıflara (abstract olmayan) değil, interface'lerin varlığına da ihtiyaç duyar.

Lazy Collection’lar

- @OneToMany ve @ManyToMany ile bildirilen collection’lar varsayılan olarak yüklenmez
- Bunlar yalnızca erişim sağlanırsa yüklenir (ör, *lazy* loading)
- Ancak bunlara bir transaction içerisinde erişmeniz gerekir
- Eğer dışarıdan erişim sağlamaya çalışırsanız bir hata alırsınız
- Yani, bu tür verilere ihtiyacınız varsa, bir işlem sırasında bunlara erişerek yüklemeyi zorlamanız gerekir
- Not: Transaction sınırlarının ayrıntılarına daha sonraki derslerde gireceğiz

Container Deployment

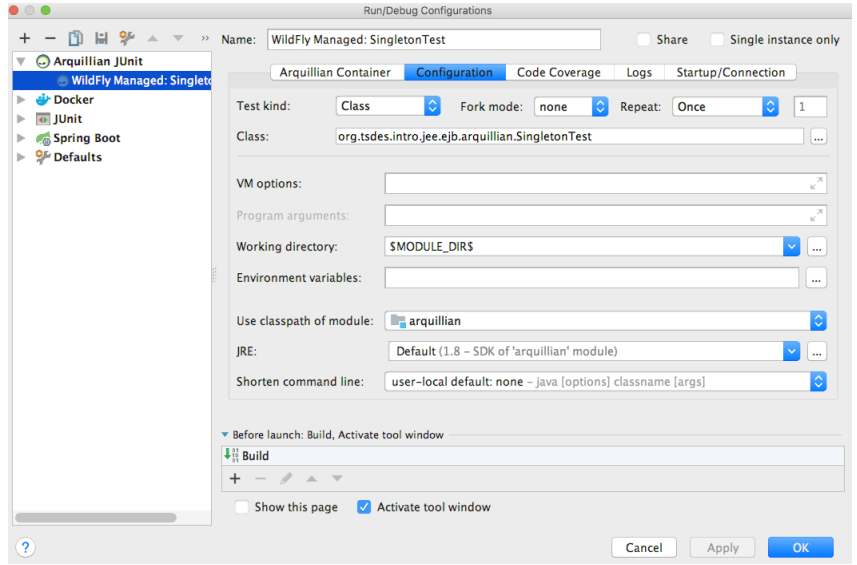
- EJBs kullanmak için, bir JEE Container içerisinde yürütmemiz gerekir
  - *WildFly, GlassFish, Payara*, etc.
- Kodumuzu JAR/WAR ile paketlememiz ve çalışan bir container'a kurmamız gerekir.
- Ancak bundan önce bir container'ı **indirmeli**, **kurmalı** ve **başlatmalıyız**.
- Ancak EJB'lerin metotlarını doğrudan JUnit ile nasıl test edebiliriz?

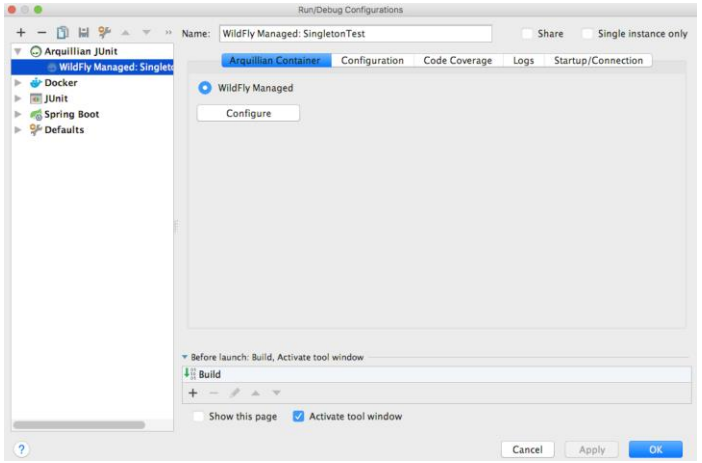
Arquillian

- JAR/WAR dosyalarını doğrudan testlerden paketlemenize ve bunları bir container'a deploy etmenize olanak tanıyan JUnit'i genişleten bir kütüphanedir
- Testler container'da çalıştırılır, bu nedenle dependency injection @EJB kullanabilir
- arquillian.xml adlı özel kaynak dosyasında yapılandırma
- Kısıtlama: yalnızca IDE'de sağ tık ile testler çalıştırılamaz, bazı manuel ayarlamalar yapmak gerekir…
- … ayrıca, hala bir JEE Container indirmeye ve kurmaya ihtiyacınız vardır
- Not: SpringBoot kullanmaya başladığımızda hayat kolaylaşacak…

Test Ayarlamaları

- Arquillian “*WildFly Managed*”
- “Working directory” -> “$MODULE_DIR$”
  - not: Intellij’in son sürümü belki farklı bir değişken adı kullanıyor olabilir, ör “$MODULE_WORKING_DIR$”. Drop-down list’ten doğru olanı seçin





Download/Install WildFly

- Build'in bir parçası olarak Maven plugin'i kullanarak yapacağız
  - Not: *Docker* docker kullanabiliriz… Ancak burada Maven eklentilerinin build sırasında birkaç farklı şey yapmak için nasıl kullanılabileceğini görmek istiyoruz.
- *WildFly* "*target*" klasörü altına kurulur
  - Yani bunu silmek istersen "*mvn clean*" çalıştırabiliriz
- Intellij'de testleri çalıştırmadan önce en azından bir sefer "*mvn test*" komutunu çalıştırmalı ve WildFly'ı download/install etmeliyiz.

Multi-Modüllü Projeler

- Genellikle, «mvn test» gibi bir komutu projenin kök dizininde yazarak çalıştırabilirsiniz
- webp büyük bir projedir: kök dizinde build yaparsanız, biraz zaman alabilir…
  - Buradaki «büyük» tanımı öğrenciler içindir, aslında büyük çaplı sistemler çok daha geniş çaplı projelerdir…
- Modülü doğrudan build ederseniz (ör "*mvn test*" i modül klasörü içerisinde çalıştırırsanız), diğer modülleri bağımlılık olarak kullanıyor ise hata oluşacaktır
- Projenin root klasöründe en az bir sefer "*mvn install -DskipTests*" komutunu çalıştırmalısınız
  - böylelikle, modüllerin jar dosyaları *~/.m2* klasörüne kurulacak ve modüller ayrı ayrı build edilse bile bu bağımlılıkları kullanacaktır

Git Repository Modülleri

- *NOT: açıklamaların büyük bir çoğunluğu kod içerisinde yorum satırı olarak bulunmaktadır, burada slaytlarda bulunmamaktadır*
- **intro/jee/ejb/stateless**
- **intro/jee/ejb/query**
- **intro/jee/ejb/lazy**
- **intro/jee/ejb/framework/injection**
- **intro/jee/ejb/framework/proxy**
- Ders 04 alıştırması (dokümantasyona bakınız)