JAVADAY ISTANBUL
2019

MODULARIZE YOUR APPLICATION WITH JAVA MODULE API

WORKSHOP

TANER DİLER

https://www.linkedin.com/in/tanerdiler/
https://github.com/tanerdiler/

JUG ISTANBUL

KAPSAM

Bu workshop süresince yazılmış olan MAVEN projesinin Java Modüllerine çevrilmesi hedeflenmektedir. Bunun için 5 tane görev tanımlanmıştır. Her görev izlenilmesi gereken yönergeleri içerir. Atölye çalışmasının etkili olabilmesi için verilen komutlar çalıştırılmalıdır.

Bu workshop kapsamında Spring Framework / Spring Boot ve diğer 3rd party api kullanımları olmayacaktır.

GEREKSINIMLER

1. Temel Java bilgisi
2. Minimum JDK 9 kurulumu (En son versiyonu kurulumu tercih edilmektedir)
3. Herhangi bir Java editörü. Vi/Vim ve herhangi bir text editörü tercih edilmektedir.
4. Git CLI
5. PDF Reader
6. Maven
7. Docker

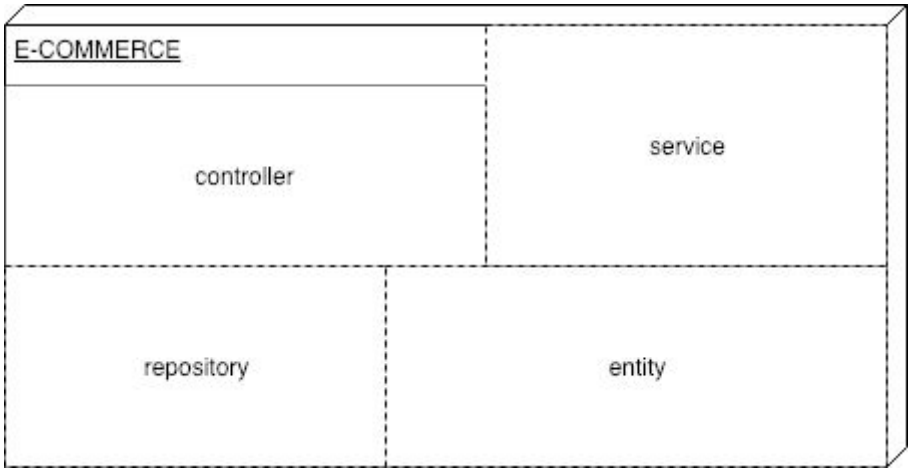
PROJE : E-COMMERCE PLATFORMU

Atölye süresince basit bir e-ticaret uygulaması üzerinde çalışacağız. Katmanlı bir mimariye sahip monolith bir uygulama iken alınan karar üzerine mikro-servise dönüştürülmesi hedeflenmektedir.

E-Ticaret uygulamasının aşağıdaki yapılardan oluşur :

1. Product : Satılan ürüne karşılık gelen sınıftır.
2. Catalog : Uygulama üzerindeki ürünlerin detay bilgilerini barındırır. Product sınıfına ait nesneleri barındırır.
3. Basket : Kullanıcının sepetidir. Seçile ürün ile birlikte kaç adet konduğu bilgisini barınıdırır.
4. Order : Sipariş bilgisini tutar. Hangi üründen kaç tane sipariş edildiğini bilgisini tutar.
5. Stock : Kullanıcıya sunulan ürünlerin miktarını tutar.

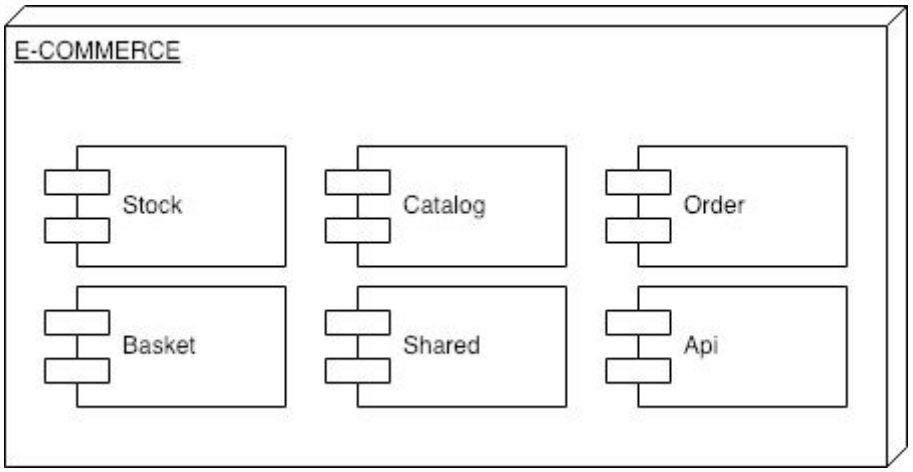
Uygulamanın en kritik senaryosu “Sipariş gerçekleştiğinde stoktan miktarın düşülmesi ve kullanıcının sepetinin sıfırlanmasıdır”.



Yapılan analizler ve toplantılar sonrasında sağlıklı bir mikro-service dönüşümü için önce Boundary Context’lerin çıkartılması ve bunların modüler hale getirilmesinin gerekliliği ortaya çıktı.

Yoğun çalışma sonrasında Stok Yönetimi, Katalog Yönetimi, Sipariş Yönetimi, Sepet Yönetimi ve bunların koordinasyonlarını sağlayan E-Commerce Platformu modülleri oluşturuldu.

Bu modüllerin versiyon ve bağımlılık yönetimi için MAVEN kullanımı tercih edildi.



Modüler bir mimari kurguladığımızda sadece Boundary Context'lerin ayrıştırılması dışında çok sık kullanılan yardımcı sınıfların tekrarlanmasını (Code Duplication) engellemek için bunların da bir modül içerisinde barındırılması ihtiyacı oluştu ve Shared modülü oluşturuldu.

Modüler bir mimaride ve özellikle Boundary Context'ler ayrıştırılmış ise en büyük tehlike Cyclic Dependency durumunun oluşmasıdır. Bunun nedeni bir modülün diğer modül ile iletişime geçmek istemesidir. Boundary Context'ler arasındaki mesajlaşmayı ve birbirine dönüşümünü yönetecek bir yapıya ihtiyaç duyulacaktır ve bu da Api modülür.

Başka bir Cyclic Dependency nedeni ise diğer modül içerisindeki bir modele ihtiyaç duyulmasıdır. Bu uygulamada Product ortak modeldir ve Cyclic Dependency oluşacak ise bu sınıf yüzünden oluşacaktır. Eğer bir modele ihtiyaç duyuluyorsa;

- o model Shared modülü içerisine taşınabilir
- sadece ihtiyaç duyulan bilgileri içeren bir model daha oluşturulabilir
- modüller arasında iletişimi sağlayacak json/xml gibi formatlar tercih edilip, ilgili modülün ihtiyaç duyduğu modele dönüştürmek için Converter'lar kullanılabilir
- event-driven bir mimari uygulanabilir

GÖREV - 1 : Maven Modülleri Java Modülüne Çevirelim

ÖN HAZIRLIK

1. Proje kodlarını lokalimize çekelim

```
> git clone git@github.com:tanerdiler/java-module-workshop.git
> cd java-module-workshop
```

2. Yukarıda aktarılan bilgiler ışığında kodları inceleyelim.
    a. model paketindeki sınıfları inceleyelim
    b. service paketindeki sınıfları inceleyelim ve özellikle OrderService sınıfındaki **order()** methodunun 24. ve 25. satırlarına dikkat edelim.
3. mission-1 branch'ine geçelim. Bunun için aşağıdaki komutları çalıştıralım

```
> git checkout mission-1
```

4. Maven modüllerini derleyelim, paketlerin oluşmasını sağlayalım ve uygulamamızı çalıştıralım

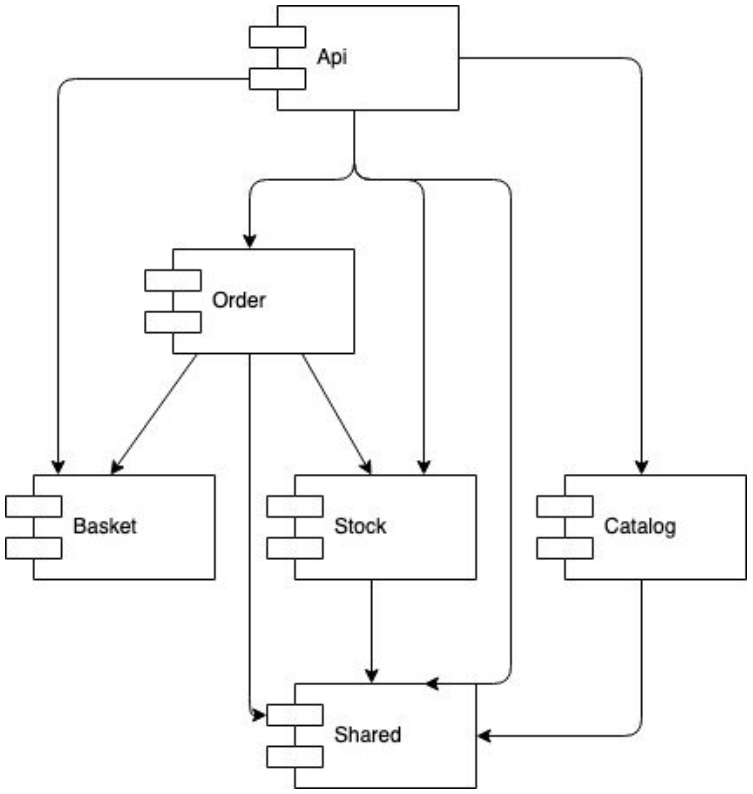
```
> ./build.sh
```

build.sh/build.bat dosyası MAVEN modüllerini derler ve oluşan paketleri classpath dizini altına kopyalar. Komut çalıştırıldığında aşağıdaki gibi bir çıktı alınmalı

```
CATALOG ADDED : {"product":Product{"id":0, "title":Boss Kulaklık, "unitPrice":500.00}}
CATALOG ADDED : {"product":Product{"id":1, "title":Apple Klavye, "unitPrice":200.00}}
CATALOG ADDED : {"product":Product{"id":2, "title":Dell Monitor, "unitPrice":200.00}}
BASKET ADD ITEM : {"buyerId":1001, "productId":0, "unitPrice":500.00, "count":5 }
BASKET ADD ITEM : {"buyerId":1001, "productId":1, "unitPrice":200.00, "count":3 }
BASKET ADD ITEM : {"buyerId":1001, "productId":2, "unitPrice":200.00, "count":1 }
********* Order Execution Started *********
BASKET CLEARED : {"buyerId":1001, "basketId":0, "totalPrice":3300.00}
PRODUCT CHECKOUTED from STOCK : {"productId":0, "countCheckouted":5, "countAfterCheckout":95}
PRODUCT CHECKOUTED from STOCK : {"productId":1, "countCheckouted":3, "countAfterCheckout":97}
PRODUCT CHECKOUTED from STOCK : {"productId":2, "countCheckouted":1, "countAfterCheckout":99}
ORDER EXECUTED : {"buyerId":1001, "basketId":0, "items":[{"productId":0, "unitPrice":500.00,
```

GÖREV TANIMI

Uygulamanın bağımlılık diyagramı aşağıdaki gibidir:



Java 9'a kadar ihtiyaç duyulan paket ve sınıf erişimlerini CLASSPATH üzerinden gerçekleştiriyorduk. CLASSPATH yanında birçok sorunu da getiriyordu:

1. Aynı jar dosyası farklı isimlerde ve hatta farklı versiyonlarda CLASSPATH içerisinde barındırılabiliyordu
2. ClassLoader ilk eriştiği jar dosyasındaki sınıfları yüklüyordu
3. ClassLoader bir kütüphane içerisindeki tüm sınıflara erişimi açık hale getiriyordu

Java Modül ile yukarıda belirtilen sorunlar ortadan kaldırılmıştır:

1. MODULEPATH içerisinde dosya ismi farklı da olsa aynı isimde sadece bir modül bulunabilir.
2. Bir modül içerisinde sadece izin verilen paketler altındaki public sınıflara erişim yapılabilmektedir.

Java Modül ekosisteminde üç farklı modül tipi vardır:

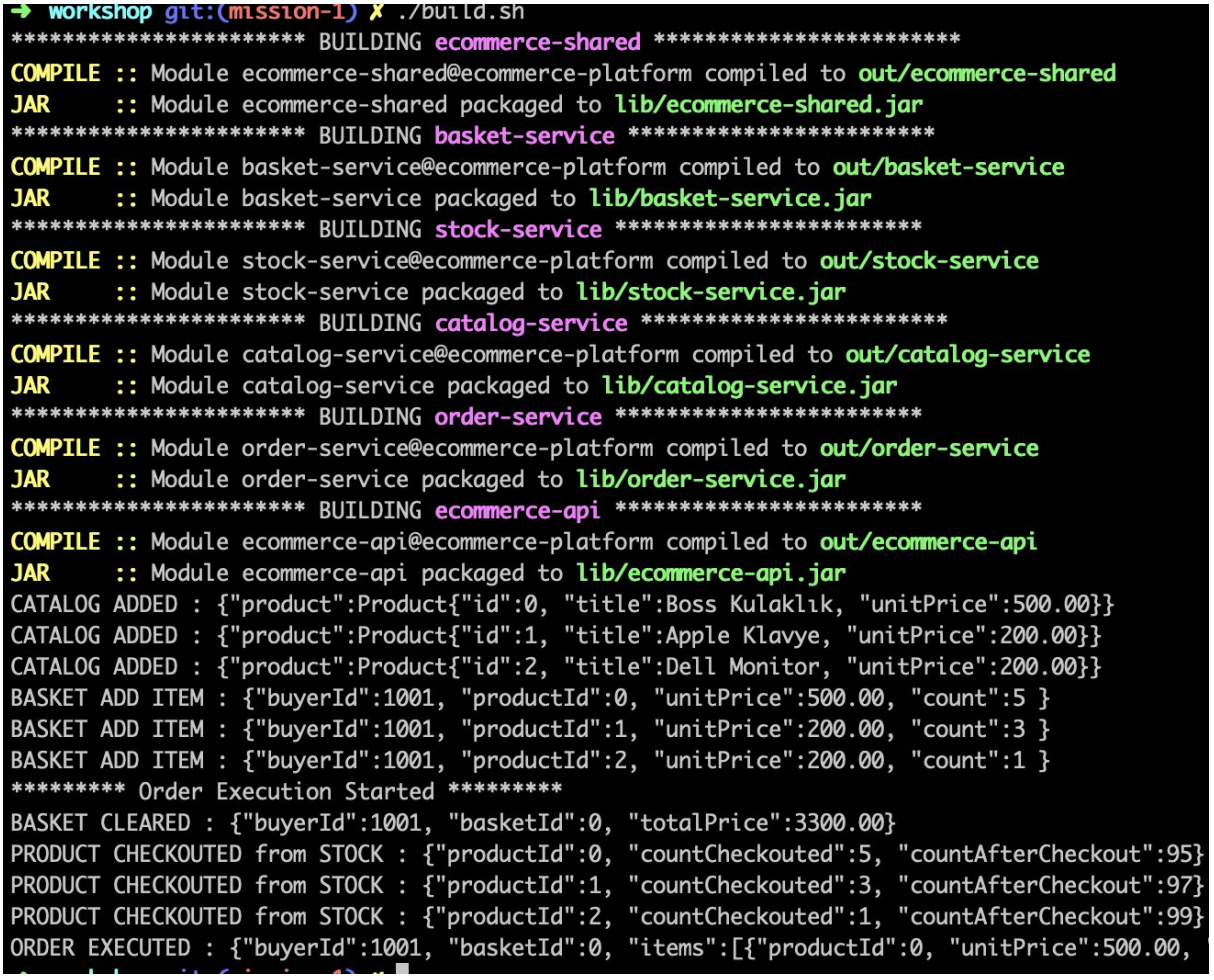
1. **NAMED MODULES :** İçerisinde module-info.java dosyası bulunan modüllerdir.
2. **AUTOMATED MODULES :** module-info.java dosyası bulunmayan JAR dosyaları MODULEPATH içerisinde barındırıldığında AUTOMATED MODULE olarak yüklenir. Modül ismi jar dosyasının isminden üretilmektedir.
3. **UNNAMED MODULES :** İsimsiz modüller Java Modül kullanılmayan yeni ve eski uygulamaların çalışmasını sağlar. CLASSPATH içerisinde bulunan JAR'lar unnamed module olarak yüklenir.

Modüller arasındaki bağımlılık ve erişim kontrolü **module-info.java** dosyasında **REQUIRES** ve **EXPORTS** yönlendirmeleri ile gerçekleşir.

- **EXPORTS :** belirtilen paketin altındaki public sınıfların erişimine izin verir. Burada belirtilecek olan paketin mutlaka en az bir java dosyası barındırması gerekmektedir.
- **REQUIRES** : Adı belirtilen modüle bağımlılığın oluşmasını sağlar.

➳ Bu bilgilerden sonra uygulamadaki modülleri Java Modüllerine dönüştürmeye başlayalım:

1. Ana dizindeki build.sh/build.bat dosyasını çalıştıralım. unnamed modüllerle çalıştığını görelim



2. Sistemde olan modüllerin neler olduğunu listeleyelim:

```
> java --list-modules
```

3. build.sh/build.bat dosyasını açalım ve ***unnamed*** ifadelerini ***named*** olarak değiştirelim ve tekrar çalıştıralım.
    a. ecommerce-shared ve basket-service modüllerinin başarılı bir şekilde oluştuğunu ama stock-service'den itibaren diğer modüllerde hata alındığını görelim
    b. modules dizini altında ecommerce-shared.jar ve basket-service.jar dosyalarının olduğunu kontrol edelim
    c. modules dizinini module-path olarak tanımlayarak sistemdeki modülleri tekrar listeleyelim

```
> java --module-path modules --list-modules
```

    d. listenin en sonunda ecommerce-shared ve basket service modüllerinin **Automatic Module** olarak listelendiğini kontrol edelim. Bunun nedeninin module-info.java dosyalarının daha tanımlanmamış olmasıdır.

4. Modüllerimizi **Named Module**'lere dönüştürmeye başlayalım.
5. Shared modülünü Named Module yapalım
    a. İlgili modülün src/main/java dizini altına **module-info.java** dosyasını oluşturalım
    b. Bu modül içerisindeki ecommerce.shared.model paketini erişime açacağız
    c. module-info.java dosyasına aşağıdaki içeriği girelim

```
module ecommerce.shared {
    exports ecommerce.shared.model;
}
```

    d. build.sh/build.bat dosyasını çalıştıralım
    e. modules dizini altında ecommerce-shared.jar ve basket-service.jar dosyalarının olduğunu görelim
    f. sistemdeki modülleri listeleyelim

```
> java --module-path modules --list-modules
```

        i. basket.service modülü **automatic** olarak listelenirken, ecommerce.shared modülü **named** olduğu için tip belirtimi yapılmamıştır..
6. Basket modülünü Named Module yapalım

a. İlgili modülün src/main/java dizini altına **module-info.java** dosyasını oluşturalım
b. Bu modül içerisindeki ecommerce.basket.service paketini erişime açacağız
c. module-info.java dosyasına aşağıdaki içeriği girelim

```
module basket.service {
    exports ecommerce.basket.service;
}
```

d. build.sh/build.bat dosyasını çalıştıralım
e. modules dizini altında ecommerce-shared.jar ve basket-service.jar dosyalarının olduğunu görelim
f. sistemdeki modülleri listeleyelim

```
> java --module-path modules  --list-modules
```

g. basket.service modülünün de named olarak listelendiğini kontrol edelim

7. Stock modülünü Named Module yapalım
    a. İlgili modülün src/main/java dizini altına **module-info.java** dosyasını oluşturalım
        i. Bu modül içerisindeki **ecommerce.stock.service** paketini erişime açacağız
        ii. ecommerce.shared modülü içerisindeki **ItemWithCount** sınıfı stock modülü içerisinde kullanıldığı için **ecommerce.shared** bağımlılığını tanımlayacağız
    b. module-info.java dosyasına aşağıdaki içeriği girelim.

```
module stock.service {
    exports ecommerce.stock.service;
    requires ecommerce.shared;
}
```

c. build.sh/build.bat dosyasını çalıştıralım
d. modules dizini altında stock-service.jar dosyasının da olduğunu görelim
e. sistemdeki modülleri listeleyelim

```
> java --module-path modules --list-modules
```

f. stock.service'in de named module olarak listelendiği kontrol edelim

8. Catalog modülünü Named Module yapalım
    a. İlgili modülün src/main/java dizini altına **module-info.java** dosyasını oluşturalım
        i. Bu modül içerisindeki **ecommerce.catalog.service** paketini erişime açacağız

ii. ecommerce.shared modülü içerisindeki **Product** sınıfı catalog modülü içerisinde kullanıldığı için **ecommerce.shared** bağımlılığını tanımlayacağız

b. module-info.java dosyasına aşağıdaki içeriği girelim.

```
module catalog.service {
    exports ecommerce.catalog.service;
    requires ecommerce.shared;
}
```

c. build.sh/build.bat dosyasını çalıştıralım
d. modules dizini altında catalog-service.jar dosyasının da olduğunu görelim
e. sistemdeki modülleri listeleyelim

```
> java --module-path --list-modules
```

f. catalog.service modülünün de named olarak listelendiğini kontrol edelim

9. Order modülünü Named Module yapalım
    a. İlgili modülün src/main/java dizini altına **module-info.java** dosyasını oluşturalım
        i. Bu modül içerisindeki **ecommerce.order.service** paketini erişime açacağız
        ii. order modülü tarafından **ecommerce.shared, basket.service ve stock.service** modüllerindeki sınıfların kullanıldığı için bu üç service olan bağımlılıkları tanımlayacağız
    b. module-info.java dosyasına aşağıdaki içeriği girelim.

```
module order.service {
    exports ecommerce.order.service;
    requires ecommerce.shared;
    requires basket.service;
    requires stock.service;
}
```

c. build.sh/build.bat dosyasını çalıştıralım
d. modules dizini altında order-service.jar dosyasının da olduğunu görelim
e. sistemdeki modülleri listeleyelim

```
> java --module-path --list-modules
```

f. order.service'in de named module olarak listelendiği kontrol edelim

10. Ana uygulama olan ecommerce-api modülünü Named Module yapalım

a. İlgili modülün src/main/java dizini altına **module-info.java** dosyasını oluşturalım
    i. Bu modül içerisindeki ecommerce.api paketini erişime açacağız
    ii. Tüm modüllere olan bağımlılıkları tanımlayacağız
b. module-info.java dosyasına aşağıdaki içeriği girelim.

```
module ecommerce.api {
    exports ecommerce.api;
    requires ecommerce.shared;
    requires basket.service;
    requires catalog.service;
    requires stock.service;
    requires order.service;
}
```

c. build.sh/build.bat dosyasını çalıştıralım ve alınan hatayı inceleyelim
    i. Bu hatanın nedeni ecommerce.api modülünün oluşturulamamasıdır. ecommerce.api modülünün neden oluşturulamadığını hata mesajından anlamaya çalışalım ve çözüme yönelik gerekli aksiyonu alalım.

Belli bir zaman harcadıktan sonra çözümü bulamadığınız durumda lütfen eğitmenden yardım isteyiniz.

d. build.sh/build.bat dosyasını çalıştıralım
e. ecommerce.api modülünün başarılı bir şekilde oluştuğunu gözlemleyelim
f. uygulamanın düzgün bir şekilde çalıştığından emin olalım.
g. sistemdeki modülleri listeleyelim

```
> java --module-path modules --list-modules
```

h. tüm modüllerin named module olarak listelendiğini kontrol edelim
i. Modül listesinde kullanmadığımız çok fazla modülün olduğunu görebilirsiniz. Bu listeyi sadece istediğimiz modülün bağımlılık ağacı bazında limitleyeb iliriz. Bunun için --limit-modules parametresini ekleyelim

```
> java --module-path modules  --limit-modules ecommerce.api --list-modules
```

Uygulama modülleri ile birlikte sadece java.base modülünün lislendiğini görebilirsiniz.

java.base modülü default modüldür ve özellikle module-info.java dosyalarında özellikle belirtilmesine gerek yoktur.

Peki java.base modülü içerisinde ne bulunduğuna ve hangi paketler erişime açılmış olduğuna bakalım. Bunun için **--describe-module** parametresini kullanalım.

```
> java --module-path modules --describe-module java.base
```

j. aynı komutu ecommerce.api modülüne uygulayalım ve bu modül tanımını jar dosyasını açmadan görelim

```
> java --module-path modules --describe-module ecommerce.api
```

SONUÇ :

Görev-1 kapsamında aşağıdakileri deneyimledik:

1. Named, Automatic Named ve Unnamed modülleri neler olduğunu
2. module-info.java dosyasının oluşturulması
3. bir paketin dışarıdan erişime açılması
4. başka bir pakete bağımlılığın tanımlanması
5. module-path içerisindeki modüllerin listelenmesi
6. bir modülün tanımının incelenmesi

GÖREV - 2 : Modül Bağımlılıklarını Ortadan Kaldıralım

ÖN HAZIRLIK

Çalışma ortamımız temizleyelim

```
> git reset --hard
> git clean -d -f
```

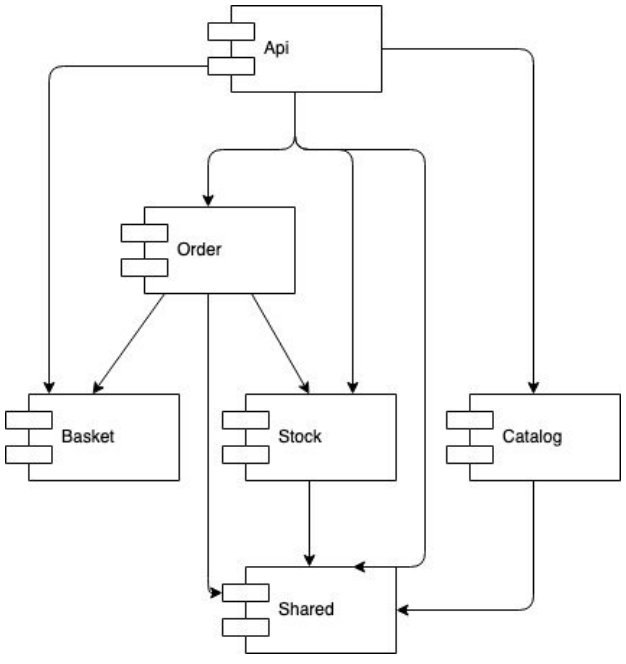
Mission-2 branch'ine geçelim

```
> git checkout mission-2
```

GÖREV TANIMI

Görev-1'de monolith uygulamayı Boundary Context modüllerine parçaladık.

Görev-1'deki modül diyagramını tekrar hatırlayalım:

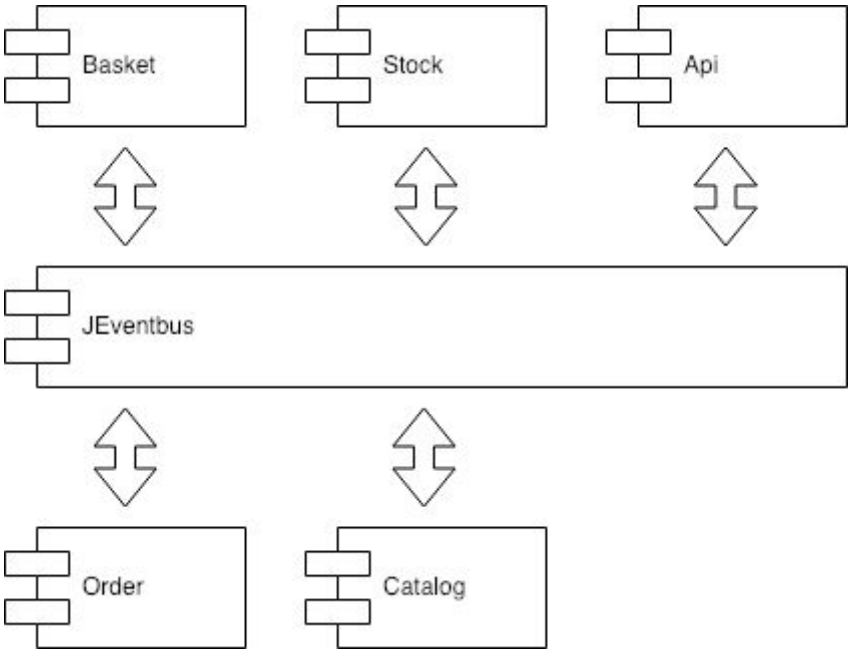


Bu diyagramı incelediğimizde Order modülünün Basket modülüne ve Stock modülüne bağımlılığının olduğunu görebiliriz. Microservice veya modül tabanlı bir mimaride bu seviyedeki bağımlılıkların olmaması gerekmektedir. Bunu önlemenin yöntemi modüller

arasındaki iletişimin bir şekilde sağlanması olacaktır. Bunun için Event Driven bir mimarı kurgulayacağız. Böyle bir mimari sonrasında istenilirse çok kolay bir şekilde :

- RabbitMQ, Kafka gibi Message Broker platformu entegre edilerek monolith uygulamadan mikro-servise geçiş yapılabilir
- Event Sourcing yapılabilir

Bu görevde sepete eklenen ürünlerin siparişi verildiğinde stoktan düşmesi ve sepetin boşaltılması işlemlerini event mekanizması üzerinden gerçekleşmesini sağlayacağız.

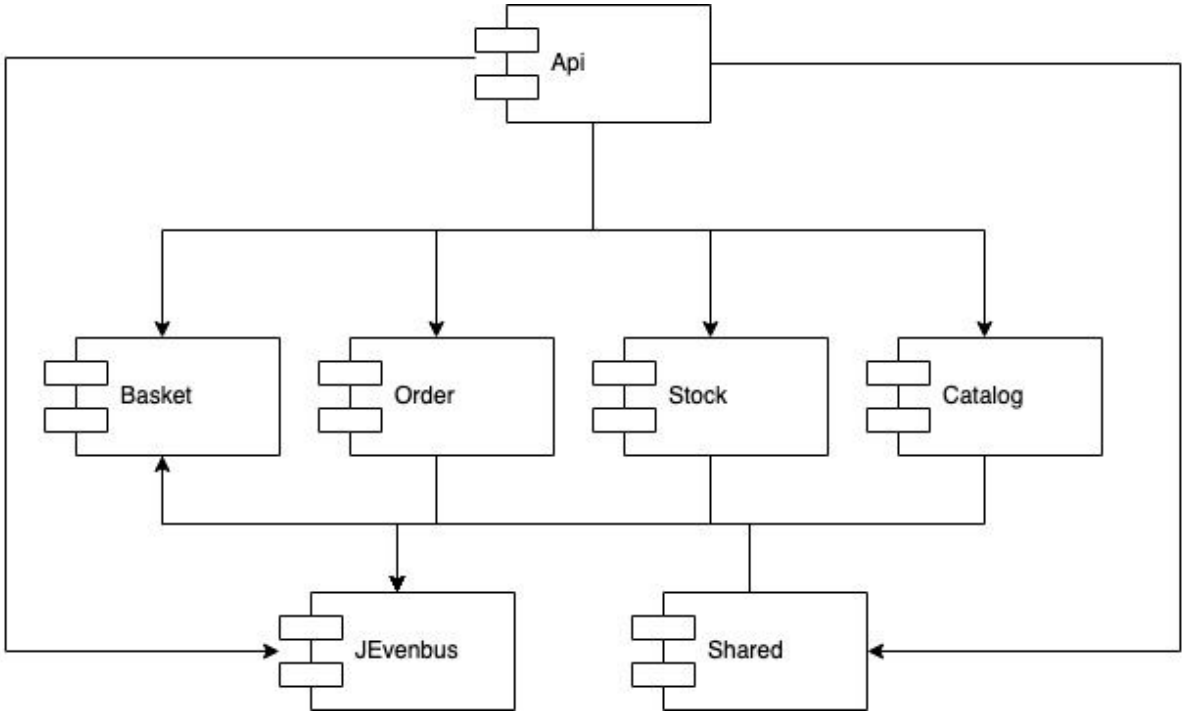


JEventbus isminde basit bir event mekanizmasını kullanacağız. JEventbus, bir olay fırlatıldığında listenerların tetiklenmesini Reflection kullanarak sağlamaktadır **BKZ: JEventbus Projesi / EventPathListenerNode.java.** Olayı Producer ile Subscriber arasında bilginin taşınması EventSource tarafından gerçekleşir. EventSource taşınacak değerleri Map<String,Object> nesnesinde tutar.

```
eventService
      .register(
            EventBuilder.aNew(ECommerceEventType.ORDER)
            .add(basketService)
            .add(stockService));
```

Bu örnekde, BasketService ve StockService nesneleri ORDER olayına Listener olarak ekleniyor. EcommerceEventType bir enum'dır ve EventType arayüzünün (interface) getMethodName() methodunu gerçekleştirir. Bu method ile döndürülen "onOrder" method ismi, Order olayı fırlatıldığında Listener'larda Reflection tarafından çalıştırılacak methodun bulunması için kullanılır.

Event mekanizması ile Order, Stock ve Basket modüllerindeki bağımlılığı kaldırdıktan sonra yeni modül diyagramı ve bağımlılıklar aşağıdaki gibi olacaktır.



Az önce event mekanizmasının Reflection kullanarak method çağrımı yaptığını öğrendik. Reflection'da private methodu çağrımı **setAccessible(true);** ile sağlanmaktadır. Java Modules ile private method ve field'lara erişim belli kurallar ile sınırlandırılmıştır. İşte bu tarz erişimlere **DEEP REFLECTION** deniliyor.

DEEP REFLECTION sınırlaması ve modüller arasındaki paket erişimlerinin kontrol edilebiliyor olması ile oluşturduğumuz JAR'lar artık STRONG ENCAPSULATION'ı sunar hale gelmiştir.

Bir modülü DEEP REFLECTION erişimine açmak için ya modül seviyesinde ya da paket seviyesinde "open" direktifi kullanılmalıdır.

```
open module stock.service {
    ....
}
```

```
module stock.service {
    open ecommerce.stock.service;
}
```

Aşağıdaki kullanımda da ilgili paket sadece jeventbus modülüne reflection erişimi için açılmıştır.

```
module stock.service {
    open ecommerce.stock.service to jeventbus;
}
```

Java Modules ile Reflection erişim kuralları:

1. Export edilmiş bir paket içerisinde sadece public alanlara reflection ile erişilebilir
2. Export edilmemiş bir paket içerisindeki alanlara DEEP REFLECTION yapılamaz
3. Modül reflection'a açıldığında içerisindeki tüm paketlere, exported/unexported, DEEP REFLECTION yapılabilir
4. Sadece belli bir paket reflection'a açıldıysa, exported/unexported olsa da DEEP REFLECTION yapılabilir

➳ Bu bilgiler ışığında gelin JEventbus modülünü projeye dahil edelim:

JEventbus git repository'si github'da bulunmaktadır. https://github.com/tanerdiler/jeventbus linkinden erişilebilir.

1. JEventbus modülünü oluşturmak için aşağıdaki komutları uygulayalım

```
> javac -d out/jeventbus $(find jeventbus/src/main/java -name "*.java")
> jar --create --file lib/jeventbus.jar -C out/jeventbus .
> mv lib/jeventbus.jar modules/.
```

2. JEventbus modülünün oluştuğunu --list-modules komutu ile kontrol edelim

```
> java --module-path modules --list-modules
```

3. jeventbus modülünün hangi paketi export ettiğini öğrenelim

```
> java --module-path modules --describe-module jeventbus
```

4. Yukarıdaki modül diyagramına bakarak uygulamamızdaki modüllere jeventbus bağımlılığını ekleyelim
5. build.sh/build.bat dosyasını çalıştıralım ve hatalar varsa gidermeye çalışalım
6. java.lang.reflect.InaccessibleObjectException hatasını aldığımız zaman artık ilgili paketlerin DEEP REFLECTION'a açılması sağlanmalı.

a. Her seferinde build.sh/build.bat çalıştırılarak reflection erişim problemleri giderilmeli
b. Yukarıda bahsedilen OPENS [MODULE], OPENS …, OPENS … TO .... yöntemlerinden biri tercih edilmeli
    i. Tavsiye edilen **opens … to …;** direktifidir.
c. Uygulamanın çalıştığından emin olana kadar hata düzeltmelerini yapalım

E-Commerce Platform'u; JEventbus'ın eklenmesi ve Order/Stock/Basket servislerinin birbirine olan bağımlılıklarının kaldırılmasıyla ideal bir yapıya ulaştı. Bizi rahatsız etmesi gereken ama farkında olmadığımız **bir sorun var**.

Bu sorun basket-service ve order-service modül tanımlamalarında core paketlerin dışarıya açılmış olmasıdır. Bu paketlere ecommerce-api modülü tarafından iki servis arasındaki mesajların birbirine dönüşümü için erişilmektedir. Mikro service veya Modül tabanlı mimarilerde mesajların birbirine dönüştürülmesinden kaçınılamaz. Bu dönüşümler kontrol altında olmalı yoksa ilerleyen zamanlarda modüllerin arasındaki bağımlılıkların tekrar oluşmasına neden olur.

Bu sorunu nasıl çözebiliriz? Tıpkı **opens … to …;** direktifi gibi **exports** direktifi ile bir paketin sadece belli modüllerin erişimine açabiliriz. Bunun için aşağıdaki direktif kullanılmalı.

exports [package] to [module];

➳ basket.service ve order.service modüllerinde core paketlerini yukarıdaki direktifi kullanarak sadece ecommerce.api modülünün erişimine açalım ve uygulamayı tekrar çalıştıralım

GÖREV - 3 : Loglamayı Servise Dönüştürelim

ÖN HAZIRLIK

➳ Çalışma ortamımızı temizleyelim

```
> git reset --hard
> git clean -d -f
```

➳ Mission-3 branch'ine geçelim

```
> git checkout mission-3
```

➳ Mission-3 dizininde bulunan logging projesini kullandığımız editöre ekleyelim

GÖREV TANIMI

E-Commerce platformunu ideal bir yapıya ulaştırdık gibi:

- Bütün boundary context'leri kendi modüllerinde barındırmaya başladık.
- Modüller arasındaki bağımlılık hiç yok denecek kadar en aza indi.
- JEventbus Event mekanizması ile modüllerin birbiri ile mesajlaşması sağlandı.
- Modülleri, deep reflection'a sadece JEventbus modülüne açık olacak şekilde oluşturduk.
- Modüllerin core business'ini barındıran core paketlerini sadece ecommerce-api modülüne açtık.

Kodları incelediğimizde business logic içerisinde çok fazla loglama satırı bulunmaktadır. Loglama her yazılımda olması gereken bir Software Concern'idir. Bu concern'i business logic içerisinden çıkarmak için farklı yöntemler bulunmaktadır ki AOP (Aspect Oriented Programming) bunlardan biridir. Bu proje kapsamında loglamayı diğer modüllerin dışına almalıyız ve bunu event mekanizması üzerinden gerçekleştireceğiz. Böylelikle modüllere SINGLE RESPONSBILITY prensibini uygulamış olacağız.

ecommerce-api modülü içerisindeki Logger sınıfı bir EventListener'dir. Önceki görevlerde her modül kendi loglamasını kendisi yapıyordu. Bu görev kapsamında ise loglama kodları Logger sınıfına kaydırıldı.

➳ Logger.java sınıfını bulup inceleyelim

Bu görev kapsamında oluşturacağımız yapı farklı loglama yöntemlerini desteklemeli. Ayrıca oluşturulacak yapı, istenildiğinde Runtime'da loglama yöntemini dinamik olarak değiştirmemizi sağlamalı.

Bunu Java Modules ile gerçekleştireceğiz. Java Modules, services özelliği ile birlikte gelmektedir. Az önce bahsettiğimiz dinamikliği özellik üzerine kurgulayacağız.

Services özelliği module-info.java dosyası içerisinde bir takım özel direktiflerin kullanılmasını gerektirir. Bu direktifler yardımıyla JVM, module path içerisinde bulunan modüllerin hangisinin Service Contract olup hangisinin Service Provider olduğunu tespit eder.

Bu direktifler:

```
provides [INTERFACE] with [IMPLEMENTER];
```

Provides direktifi ile, Service Contract'ı olan INTERFACE'i hangi sınıfın gerçekleştirdiği ve bu sınıf ile hizmet verileceği belirtilir. Service Contract'ını içeren modüle ayrıca requires direktifi ile bağımlılık oluşturulmalıdır.

```
uses [INTERFACE];
```

Uses direktifi ile hangi servisin modül tarafından kullanılacağı belirtilir. Bunun için hedef servisin Service Contract'ı olan INTERFACE belirtilmelidir. Service Contract'ını içeren modüle ayrıca requires direktifi ile bağımlılık oluşturulmalıdır.

Bu görevde mission-3 branch'ine geçiş ile birlikte logging isminde bir maven projesi gelecektir.

Bu projede ki maven modülleri:

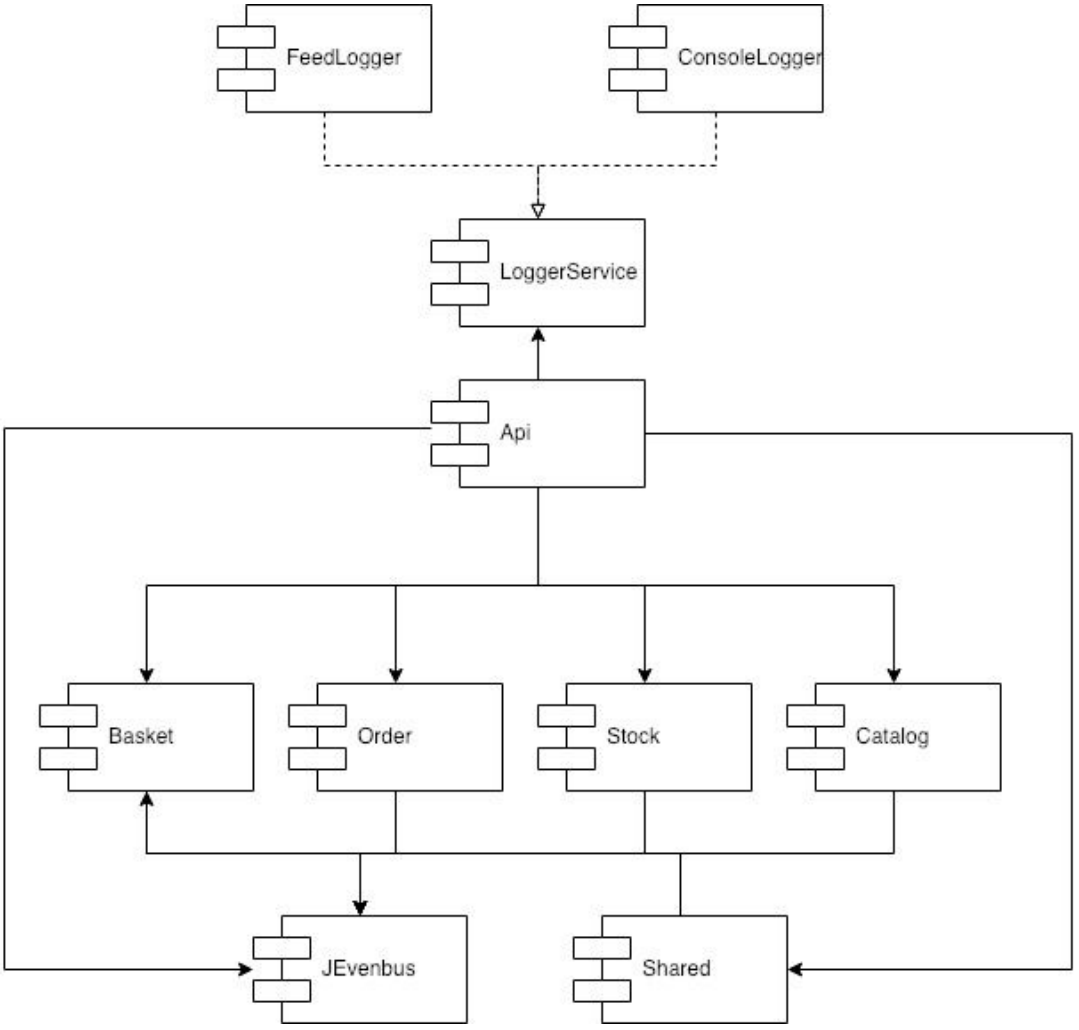
1. **logger-service modülü** : bu modül içerisinde model ve service contract'ı bulunmaktadır.
2. **console-logger modülü** : oluşan event'lerin konsola basılmasını sağlar.
3. **feed-logger modülü** : oluşan event'lerin feed kaydı olarak veritabanına kaydedilmesini sağlar. (Bu workshop için konsola basacak şekilde ayarlandı)

Java'da Runtime'da Dependency Injection olanağı sunan bir sınıf var, **java.util.ServiceLoader**. Bu sınıf Java 1.3'den beri var ama 1.3'de sadece dahili amaçlar için kullanılıyordu. Java 1.6'da bu sınıf harici kullanımlara açık hale getirilmiştir. Java 9 ile yeniden düzenlenmiştir.

ServiceLoader'in genel kullanım şekli aşağıdaki gibidir. Bu örnekte service provider/implementer sınıfları ismine göre Map içerisine atılmaktadır.

```
private LoggerFactory(){
        ServiceLoader loader = ServiceLoader.load(LoggerService.class);
        Iterator<LoggerService> iterator = loader.iterator();
        while(iterator.hasNext()){
            LoggerService serviceImpl = iterator.next();
            serviceProviders.put(serviceImpl.getServiceName(), serviceImpl);
        }
    }
```

Bu servis entegrasyonu ile modül diyagramı aşağıdaki gibi olacaktır:



➳ Bu bilgiler ışığında gelin Logging modülünü projeye dahil edelim:

1. logger-service modülünü Java Modülü'ne çevirelim
    a. module-info.java dosyasını oluşturalım
    b. modülün ismini "logger.service" koyalım
    c. paketleri dışarıdan erişime açalım
2. console-logger modülünü Java Modülü'ne çevirelim
    a. module-info.java dosyasını oluşturalım
    b. modülün ismini "console.logger" koyalım
    c. provides direktifini kullanarak LoggerService interface'nin hangi sınıf ile gerçekleştirildiğini belirtelim

```
provides logger.service.LoggerService with logger.console.ConsoleLogger;
```

    d. logger.service modülüne bağımlılık oluşturmayı unutmayalım
3. feed-logger modülünü Java Modülü'ne çevirelim
    a. module-info.java dosyasını oluşturalım
    b. modülün ismini "fee.logger" koyalım
    c. provides direktifini kullanarak LoggerService interface'nin hangi sınıf ile gerçekleştirildiğini belirtelim

```
provides logger.service.LoggerService with logger.feed.FeedLogger;
```

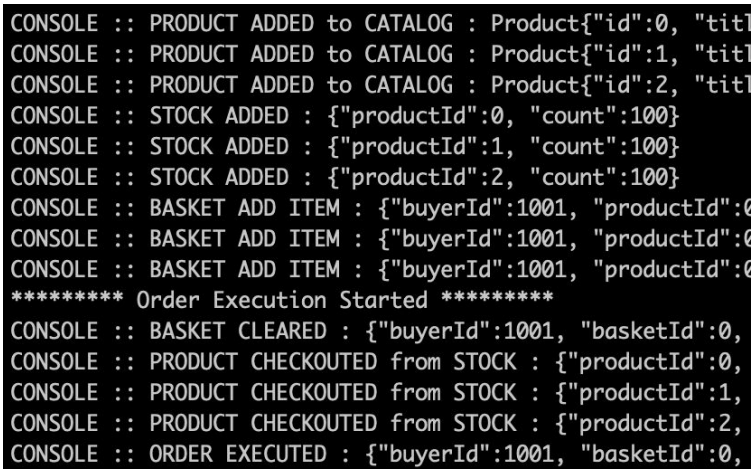
    d. logger.service modülüne bağımlılık oluşturmayı unutmayalım
4. ecommerce-api modülünün module-info.java dosyasına
    a. logger.service bağımlılığını oluşturalım
    b. logger.service içerisindeki LoggerService kullanımını tanımlayalım

```
uses logger.service.LoggerService;
```

5. build.sh/build.bat dosyasını çalıştıralım

```
FEED :: PRODUCT ADDED to CATALOG : Product{"id":0, "title
FEED :: PRODUCT ADDED to CATALOG : Product{"id":1, "title
FEED :: PRODUCT ADDED to CATALOG : Product{"id":2, "title
FEED :: STOCK ADDED : {"productId":0, "count":100}
FEED :: STOCK ADDED : {"productId":1, "count":100}
FEED :: STOCK ADDED : {"productId":2, "count":100}
FEED :: BASKET ADD ITEM : {"buyerId":1001, "productId":0,
FEED :: BASKET ADD ITEM : {"buyerId":1001, "productId":0,
FEED :: BASKET ADD ITEM : {"buyerId":1001, "productId":0,
********* Order Execution Started *********
FEED :: BASKET CLEARED : {"buyerId":1001, "basketId":0, "
FEED :: PRODUCT CHECKOUTED from STOCK : {"productId":0, "
FEED :: PRODUCT CHECKOUTED from STOCK : {"productId":1, "
FEED :: PRODUCT CHECKOUTED from STOCK : {"productId":2, "
FEED :: ORDER EXECUTED : {"buyerId":1001, "basketId":0, "
```

6. ecommerce-api modülünde Application.java içerisinde 24. satırda feed ifadesini console olarak değiştirelim ve build.sh/build.bat çalıştıralım



GÖREV - 4 : Deployment

ÖN HAZIRLIK

➳ Çalışma ortamımızı temizleyelim

```
> git reset --hard
> git clean -d -f
```

➳ Mission-4 branch'ine geçelim

```
> git checkout mission-4
```

GÖREV TANIMI

Gelin şimdiye kadar ne yaptığımızı kısaca hatırlayalım:

1. Monolith bir uygulamayı boundary context'lerine parçalamak için yola çıktık
2. Önce Monolith bir uygulamayı Maven modüllerine dönüştürdük
3. Maven Modüllerine module-info.java tanımlarını ekleyerek Named Module haline dönüştürdük
4. Exports, Exports To ve Requires direktiflerini kullanarak modül bağımlılıklarını kurduk
5. Event mekanizması ekleyerek modüller arasındaki direkt olan bağımlılıkları ortadan kaldırdık
6. Opens ve Opens To direktifleri ile event mekanizmasına private olan metodlar için deep reflection yapmasına izin verdik
7. Loglamayı, event mekanizmasını kullanarak business logic içerisinde çıkardık
8. Provides With ve Uses direktifleri ile loglama için farklı loglama servis sağlayıcılarının kullanabilmesini sağladık. Hatta çalışan servisin Runtime'da değiştirilebilir olmasını sağladık.

Artık çalışan uygulamamızı paketleyip deploy edebiliriz.

Java Module Api ile gelen bir başka özellik Runtime Image'lerinin oluşturulabilmesidir. Bildiğimiz gibi bir java uygulamasını çalıştırabilmek için o sunucuda JDK kurulmasına gerek yoktur, JRE'nin olması yeterlidir. JRE, java içerisindeki tüm temel kütüphaneleri içerir. Runtime Image'de uygulamamızın JRE gibi paketlenmesini sağlar.

Bu özelliği faydalarını şöyle sıralayabiliriz:

1. Gereksiz kodların bulunmasını engelleyerek uygulamanın güvenliğini artırır.
2. Modüllerin linklenmesi image oluşturulurken yapıldığı için uygulamanın ayağa kalkma süresi kısalır.
3. Gereksiz kütüphane/modüllerin barındırılmasını engelleyerek oluşturulan Runtime Image'in boyutu küçüktür.
4. Oluşturulan Runtime Image dışarıdan kodların değiştirilmesine, Java Versiyonunun değiştirilmesine izin vermez. Çalıştığı sistemdeki JRE kurulumundan etkilenmez. Ebediyen oluşturulan Java Versiyonunu kullanır.
5. Docker Container içerisinde barındırmaya uygundur.
6. Kurulum yapmadan Runtime Image çalıştırılabilir.

Yukarıda elde ettiğimiz faydalar ile Cloud Native Prensiplerini bir çoğunu karşılamış oluyoruz. Cloud Native Prensipleri için şu yazıyı inceleyebilirsiniz.

Runtime Image oluşturmak için Java 9 ve sonraki versiyonlar ile birlikte gelen **JLINK** komutu kullanılmaktadır.

Java 9 ile gelen bir diğer komut ise **JDEPS**'dir. JDeps ile bir modülün bağımlılıklarının analizi gerçekleştirilir.

➳ Önceki görevlerimizde modül seviyesinde analiz yapmak için hangi komutları kullandığımızı hatırlayalım:

1. **java --module-path [PATH] --list-modules** : belirtilen PATH içerisindeki tüm modülleri listeler.
2. **java --module-path [PATH] --limit-modules [MODULE-1] --list-modules** : Sadece MODULE-1'in modül ağacındaki modülleri listeler.
3. **java --module-path … --describe-module …** : belirtilen modülün module-info.java içeriğini gösterir.

Java Runtime Image oluşturmak için;

1. build.sh/build.bat dosyasını çalıştıralım
2. ecommerce.api'nin modül ağacını analiz edelim

```
> jdeps --module-path modules -m ecommerce.api
> jdeps --module-path modules -verbose:package -m ecommerce.api
```

komutu çalıştırdığımızda aşağıdaki gibi bir çıktı verecektir.

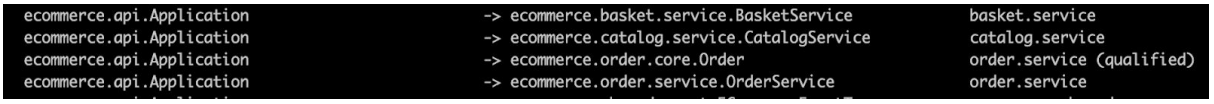
Yukarıda görüldüğü gibi üç bölümden oluşan bir döküm almaktadır:

1. Kısım : module-info.java dökümüdür.
2. Kısım : bağımlı olunan modül isimleri listelenir.
3. Kısım : paket seviyesinde bağımlılık dökümüdür. Şu şekilde yorumlanabilir: ***ecommerce.api*** *modülündeki* ***ecommerce.basket.core*** *paketi* ***basket.service modülüne*** *bağımlıdır.*

➳ Paketlerdeki hangi sınıflar bu modülleri kullanıyor?

```
> jdeps --module-path modules  -verbose:class -m ecommerce.api
```

Bu komut çalıştırdığımızda 3. kısmın ikinci sütununda paketler yerine sınıflar listelenmektedir.



➳ Peki özel bir paket ismi veya sınıf ismini aratmamız gerekirse **--regex [EXP]** parametresini kullanabiliriz.

```
> jdeps --module-path modules --regex ".*shared.*" -verbose:package --module
ecommerce.api
```

➳ Peki bu paketlerdeki sınıfların bağımlılıklarını tespit edelim?

```
> jdeps --module-path modules --regex ".*shared.*" -verbose:class --module
```

ecommerce.api

--regex parametresi ile arama yaptığımızda sadece ecommerce-api değil onun bağlı olduğu modüllerin de analizinin yapıldığını görebilirsiniz.

➳ “Bu kadar bilgi fazla ben sadece modül isimlerinin listelenmesini istiyorum” diyorsanız o zaman **-summary** parametresini kullanabilirsiniz.

> jdeps --module-path modules **-summary** --module ecommerce.api

```
ecommerce.api -> basket.service
ecommerce.api -> catalog.service
ecommerce.api -> ecommerce.shared
ecommerce.api -> java.base
ecommerce.api -> jeventbus
ecommerce.api -> logger.service
ecommerce.api -> order.service
ecommerce.api -> stock.service
```

3. Modül bağımlılıklarının grafik olarak gösterimini sağlayalım

> jdeps --module-path modules **--dot-output out** --module ecommerce.api

bu komut out dizini altına dot uzantılı dosya oluşturur. Bu dosyaları iki şekilde görüntüleyebilirsiniz:

a. Dot dosya içeriğini https://dreampuf.github.io/GraphvizOnline editöründe görselleştirebilirsiniz.
b. Graphviz’in CLI kurup “**dot -Tpng -O out/ecommerce.api.dot**” komutu çalıştırılarak png dosyasını oluşturabilirsiniz.

4. Runtime Image’i oluşturmaya başlayalım

a. Aşağıdaki komutu çalıştıralım

> jlink --module-path modules --add-modules ecommerce.api --output out/ecomm-api --launcher ecom=ecommerce.api/ecommerce.api.Application

Bu komut ile out/ecomm-api dizini altına ecommerce.api modülü ve onun modül bağımlılıklarını kullanarak ecom komutu ile çalıştırılabilir bir imaj oluşturuyoruz.

b. out/ecomm-api dizininin oluştuğunu kontrol edelim
c. Oluşan imajı iki şekilde çalıştırabiliriz
    i. --launcher parametresi ile belirtilen ecom’u çağırarak

```
> ./out/ecomm-api/bin/ecom
```

ii. java komutu ile

```
> ./out/ecomm-api/bin/java -m ecommerce.api/ecommerce.api.Application
```

iii. her iki komutu çalıştırdığımızda aşağıdaki hatayı almalıyız.

iv. Bu hatanın nedenini tespit etmek için logger service provider modüllerinin eklenip eklenmediğini kontrol edelim

Öncelikle yeni öğrendiğimiz jdeps komutunu kullanalım

```
> jdeps --module-path modules -summary --module ecommerce.api
```

Modül listesinde console.logger ve feed.logger modüllerinin olmadığını göreceğiz.

Diğer bir yöntem ise önceki görevlerde öğrendiğimiz --list-modules komutunu kullanmaktır. Oluşan imajın küçük bir JRE olduğunu söylemiştik. Demek ki bin dizini altında java komutu olacaktır. Aşağıdaki komutu çalıştıralım.

```
> ./out/ecomm-api/bin/java --list-modules
```

Listelenen modüllerin içerisinde feed.logger ve console.logger modüllerinin olmadığını göreceğiz.

jlink komutu belirtilen modül ve ya modüllerin bağımlılık ağacını kullanır. Hatırlarsanız ecommerce.api'nin module-info.java tanımı içerisinde özellikle feed.logger ve

console.logger belirtimi yoktur. Sadece logger.service vardır.

5. JLink komutunun service provider modüllerini otomatik eklemesi için **--bind-services** parametresini kullanabiliriz

> jlink --module-path modules --add-modules ecommerce.api **--bind-services** --output out/ecomm-api --launcher ecom=ecommerce.api/ecommerce.api.Application

Oluşan imajı tekrar çalıştıralım ve uygulamanın çalıştığını görelim.

> ./out/ecomm-api/bin/ecom

--bind-services parametresini kullandığımızda module path içerisinde ilgili servise ait tüm service provider modülleri eklenir. Bu kontrolümüz dışında istemediğimiz modüllerin eklenmesine ve bu da imaj boyutunun büyümesine, gereksiz olan kodlardan dolayı güvenlik seviyesinin düşmesine neden olur. Tavsiye edilen yöntem service provider modülleri teker teker eklemektir.

6. Logger Servisi için hangi modülleri ekleyeceğimizi tespit edelim. Bunun için Jlink'i **--suggest-providers** parametresi ile çalıştıralım:

> jlink --module-path modules **--suggest-providers** logger.service.LoggerService

komut çalıştırıldığında aşağıdaki gibi bir sonuç alınacaktır.

```
Suggested providers:
  console.logger provides logger.service.LoggerService used by ecommerce.api
  feed.logger provides logger.service.LoggerService used by ecommerce.api
```

Artık imaj oluştururken hangi service provider'ları ekleyeceğimizi biliyouruz. Aşağıdaki komutu istediğimiz service provider'ları ekleyerek tekrar çalıştıralım:

> jlink --module-path modules **--add-modules ecommerce.api,feed.logger,console.logger** --output out/ecomm-api --launcher ecom=ecommerce.api/ecommerce.api.Application

İmajı tekrar çalıştıralım ve uygulamanın çalıştığını görelim.

```
> ./out/ecomm-api/bin/ecom
```

7. Oluşan Runtime Image'in boyutu nedir ve daha da küçültebilir miyiz?
    a. Oluşan imaj dizininin boyutunu öğrenelim

```
> du -sh out/ecomm-api
```

    b. Elde ettiğimiz değeri daha küçültebilir miyiz? Aşağıdaki komutu çalıştıralım:

```
> jlink --module-path modules  --add-modules ecommerce.api,feed.logger,console.logger
--verbose  --strip-debug --compress 2 --no-header-files --no-man-pages
--output out/ecomm-api  --launcher ecom=ecommerce.api/ecommerce.api.Application
```

boyut kontrolü yapalım

```
> du -sh out/ecomm-api
```

yeni boyutun bir önceki boyuttan küçük olduğunu görebiliriz.

    c. Compress değerini değiştirerek etkinin ne olduğunu aşağıdaki tabloya yazalım

--compress=0	???
--compress=1	???
--compress=2	???

Runtime Image'ı oluşturduk ve kabul edilebilir bir boyuta kadar küçülttük. Runtime Image klasik yaklaşımla SSH ile hedef sunucuya atılabilir. Bundan sonraki adımlarımızda Runtime Image'ı Docker Container içerisine yerleştireceğiz. Bunun için sisteminizde Docker Engine ve Docler CLI kurulu olmalıdır.

➳ Mission-4 Branch'i ile çalışma ortamımızda Dockerfile'in geldiğini görebiliriz. Dockerfile'i açalım ve inceleyelim.

➳ Dockerfile'in iki stage'den oluştuğunu görebilirsiniz. Build isimli stage'de JLink komutu çalıştırılarak imaj oluşturulmaktadır. Build stage'ine ihtiyaç duymamızın nedeni olulan imajın cross-platform olmamasıdır. Bu yüzden tavsiye edilen yöntem imajın, çalıştırılacağı ortamda oluşturulmasıdır.

Docker imajı oluşturalım ve container oluşturarak çalıştıralım:

```
> docker build . -t ecommerce-api
> docker run --name ecommerce ecommerce-api
```

➳ Oluşan docker imajının boyutunu kontrol edelim ve boyutun 50Mb'a yakın olduğunu görelim

```
> docker images -a  | grep ecommerce
```

Bu imajı Alpine Linux üzerinden oluşturduk. Alpine Linux imajı yaklaşık 4Mb boyutundadır. Dockerfile içerisinde Alpine Imajına glibc eklediğimizi görebilirsiniz. Alpine Linux, diğer linux dağıtımlarından farklı olarak **musl libc** kullanmaktadır. OpenJDK musl libc ile uyumlu değildir. Devam etmekte olan Portola projesi ile OpenJDK, musl libc uyumlu hale getirilmeye çalışılmaktadır.

GÖREV - 5 : Unnamed Module ve Automatic Module İle Çalışalım

ÖN HAZIRLIK

➳ Çalışma ortamımızı temizleyelim

```
> git reset --hard
> git clean -d -f
```

➳ Mission-5 branch'ine geçelim

```
> git checkout mission-5
```

GÖREV TANIMI

Şimdiye kadar tamamladığımız görevlerde hep Named Module'ler üzerinde çalıştık. Şimdi gelin Unnamed Module ve Automatic Module ile e-commerce platformunu ayağa kaldırmaya çalışalım.

Otomatik Modüller modül dizini içerisinde bulunur ve içerisinde module-info.java dosyası bulunmaz. İsimlendirme dosya isminden yapılır. Bu modüllerin içerisindeki tüm paketler erişilebilir durumdadır. module-info.java dosyası olmadığı içini modül diyagramı yoktur. Bu yüzden bağımlılık çözümlemesi yapılamaz. Bunun için bağımlı olunan modüller **--add-modules** parametresi ile belirtilmelidir.

Aşağıda e-commerce platformundaki modüllerin bağımlılıklarını inceleyebilirsiniz:

PROJE	MODÜL TİPİ	BAĞIMLILIKLAR
jeventbus	Named Module	
logging/logger-service	Automatic Module	
logging/console-logger	Automatic Module	logger.service
logging/feed-logger	Automatic Module	logger.service
ecommerce-shared	Automatic Module	jeventbus
basket-service	Automatic Module	jeventbus,ecommerce.shared
catalog-service	Automatic Module	jeventbus,ecommerce.shared

stock-service	Automatic Module	jeventbus,ecommerce.shared
order-service	Automatic Module	jeventbus,ecommerce.shared
ecommerce-api	Automatic Module	jeventbus,logger.service,feed.logger,console.logger,ecommerce.shared,basket.service,catalog.service,stock.service,order.service

Bu görev kapsamında build.sh dosyasında bazı değişiklikler yapıldı. Modül tanım array'ine 4. parametre olarak bağımlı olunan modüller eklenmiştir. Automatic olarak işaretlenmiş modüllerin oluşturulmasında kullanılmaktadır.

Otomatik modüller üzerinden e-commerce uygulamasını çalıştıralım:

1. build.sh/build.bat dosyasını çalıştıralım
    a. NullPointerException hatasının alındığı görülebilir. Bunun nedeni Loglama servisi için service provider'ların JVM tarafından tespit edilememiş olmasıdır.
    b. ServiceLoader sınıfının Java 9'dan önce de var olduğunu daha önce söylemiştik. Bu versiyonlarda hangi sınıfın hangi servisi gerçekleştirdiği farklı bir yöntem ile belirtilmektedir.
    c. feed-logger maven modülünde;
        i. src/main/**resource/META-INF/services** path'ini oluşturalım
        ii. Bu dizin altında **logger.service.LoggerService** isminde bir dosya oluşturalım
        iii. Bu dosya içerisinde interface'i gerçekleştiren sınıfı, paketi ile birlikte belirtelim: içerisine **logger.feed.FeedLogger** yazalım
    d. console-logger maven modülünde;
        i. src/main/**resource/META-INF/services** path'ini oluşturalım
        ii. Bu dizin altında **logger.service.LoggerService** isminde bir dosya oluşturalım
        iii. Bu dosya içerisinde interface'i gerçekleştiren sınıfı paketi ile birlikte belirtelim: içerisine **logger.console.ConsoleLogger** yazalım
    e. build.sh/build.bat dosyasını çalıştıralım ve uygulamanın düzgün çalıştığından emin olalım

Otomatik modüllerin isimlendirmesinin dosya isminden yapılmamasını ve kalıcılığını sağlayabiliriz. Bunun için META-INF dizini altında MANIFEST.FS dosyası oluşturup içerisine aşağıdaki tanımlamayı yapmak yeterli olacaktır.

Automatic-Module-Name: <module-name>