Taner DİLER

MODULARIZE YOUR APPLICATION WITH JAVA MODULE API

WORKSHOP

TANER DİLER

taner.diler[at]gmail.com

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

CONTENT / SCOPE

A monolith project that has been already written on Java by using Maven builder will be converted to Java Modules during this workshop. The workshop has 5 missions. Each missions includes directives to achieve mission.

This workshop doesn’t force you to write code. Code has been already written and refactored for each mission. Refactored code that you are working on is committed to branch named as mission-nth.

High level frameworks such as Spring Boot, Microprofile, Quarkus is not being targeted on this workshop.

REQUIREMENTS

1. Java Knowledge/Experience
2. Minimum JDK 9 installation (last version preferred)
3. A Java Editor that you have experience - it will be needed only for code review.
4. Basic text editor
5. Git CLI
6. PDF Reader
7. Maven
8. Docker

PROJECT : E-COMMERCE PLATFORM

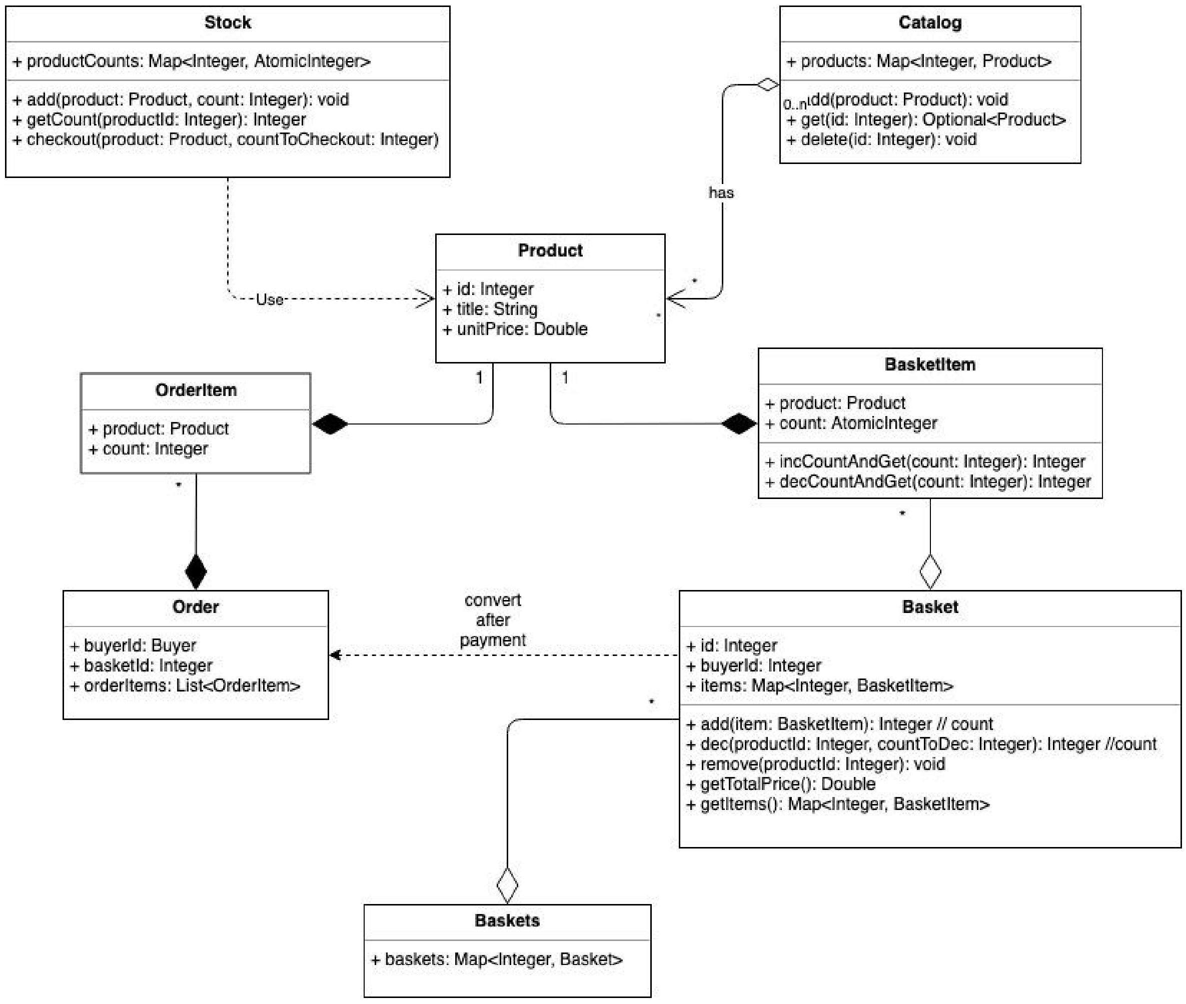
We will work on a basic e-commerce app during the workshop. This application is a standalone application that doesn't have any database and web service integration. The company that founder of this project has decided to migrate from monolith to microservice architecture.

The main reasons of this migration are that :

- cycle time is high to be acceptable
- it forces all teams to work same code base
- a small change needs other components to be changed
- it forces whole application to be restarted
- it takes huge time to start

Let's review the current architecture:

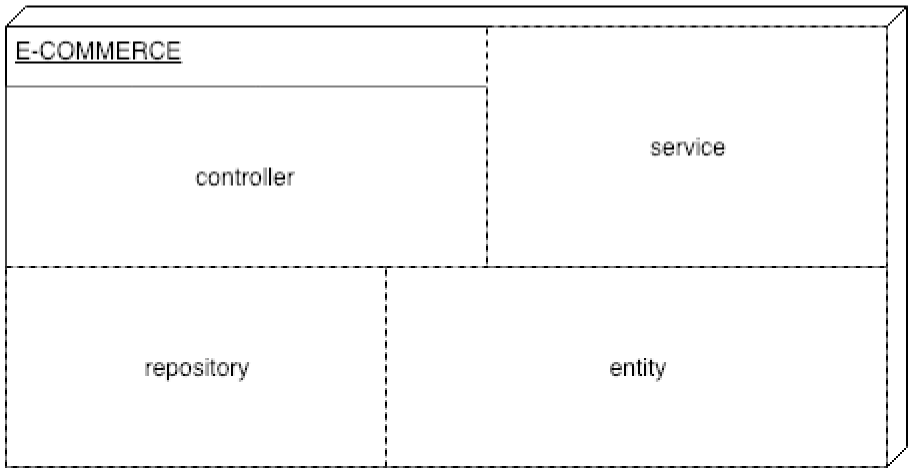
Application has 5 main models **- product, catalog, stock, basket, order**



Stock
+ productCounts: Map<Integer, AtomicInteger>
+ add(product: Product, count: Integer): void
+ getCount(productId: Integer): Integer
+ checkout(product: Product, countToCheckout: Integer)
Catalog
+ products: Map<Integer, Product>
+ add(product: Product): void
+ get(id: Integer): Optional<Product>
+ delete(id: Integer): void
0..n
has
*
Use
Product
+ id: Integer
+ title: String
+ unitPrice: Double
1
1
OrderItem
+ product: Product
+ count: Integer
BasketItem
+ product: Product
+ count: AtomicInteger
+ incCountAndGet(count: Integer): Integer
+ decCountAndGet(count: Integer): Integer
Order
+ buyerId: Buyer
+ basketId: Integer
+ orderItems: List<OrderItem>
convert after payment
Basket
+ id: Integer
+ buyerId: Integer
+ items: Map<Integer, BasketItem>
+ add(item: BasketItem): Integer // count
+ dec(productId: Integer, countToDec: Integer): Integer //count
+ remove(productId: Integer): void
+ getTotalPrice(): Double
+ getItems(): Map<Integer, BasketItem>
Baskets
+ baskets: Map<Integer, Basket>

As you see in uml diagram above, all classes are depended to Product class.

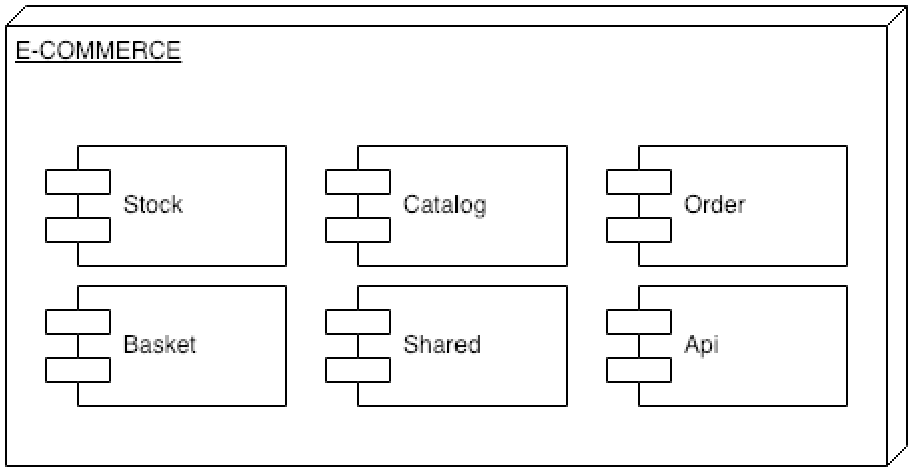
Anemic Domain Model is root cause of many problems.



After a series of meetings, it was decided to use Domain Driven Design practices and defining boundary contexts would be the first step:

1. Stock Management
2. Catalog Management
3. Order Management
4. Basket Management

After defining boundary contexts, source code would be splitted to 6 maven modules.



Additional to 4 module of boundary contexts defined above, Api and Shared modules were created. We need to Shared module to locate widely used models in order to decrease duplication. We need to create Api module as a facade to response calls from mobile ui or web ui.

We have to be aware of creating cyclic dependencies between modules. While working with modules, it is easy to fall into this problem. We could say that communications between modules would be mandatory while growing project.

In order to escape from this situation:

- common classes could be put into a shared module
- communication between modules is being done by using json/xml. Converters could be written to convert json/xml to model vice versa.
- event-driven architecture could be used

After giving a short information why the company needs to migrate to microservices and obstacles on the way, let’s start workshop.

PREPARATION : Review Monolith App

1. Download project source codes into local

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

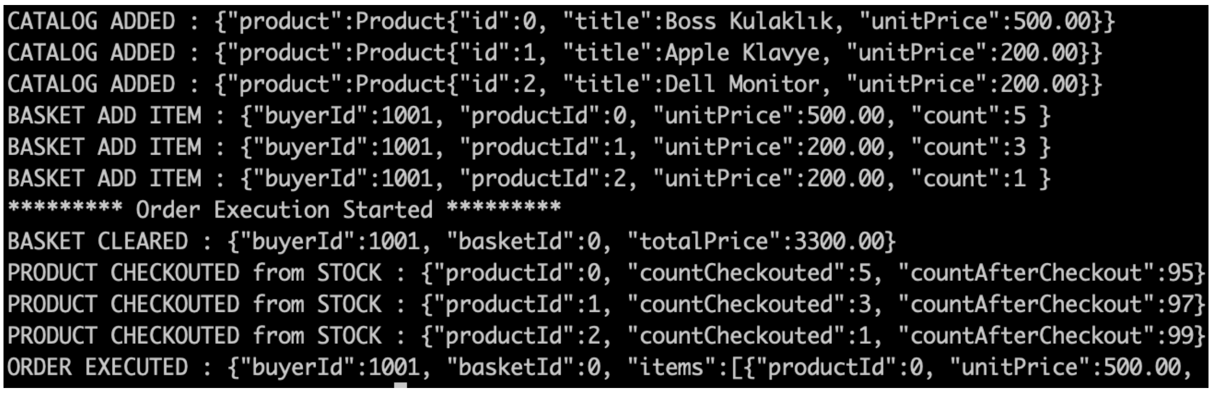
2. Let's review the codes based on the information above
   a. review classes in model package
   b. review classes in service package.

😟 Pay attention to OrderService class. OrderService has composition to BasketService and StockService. We can figure out that communication between services is mandatory. That could be a big problem in the future.

3. Let's run monolith application

```
cd workshop
cd ecommerce-platform
mvn clean package
java -cp target/ecommerce-api.jar ecommerce.api.Application
```

4. Be sure that taking result below



```
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,
```

MISSION - 1 : Let’s Convert Maven Modules to Java Modules

PREPARATION

Let’s switch to mission-1 branch

```
git checkout mission-1
```

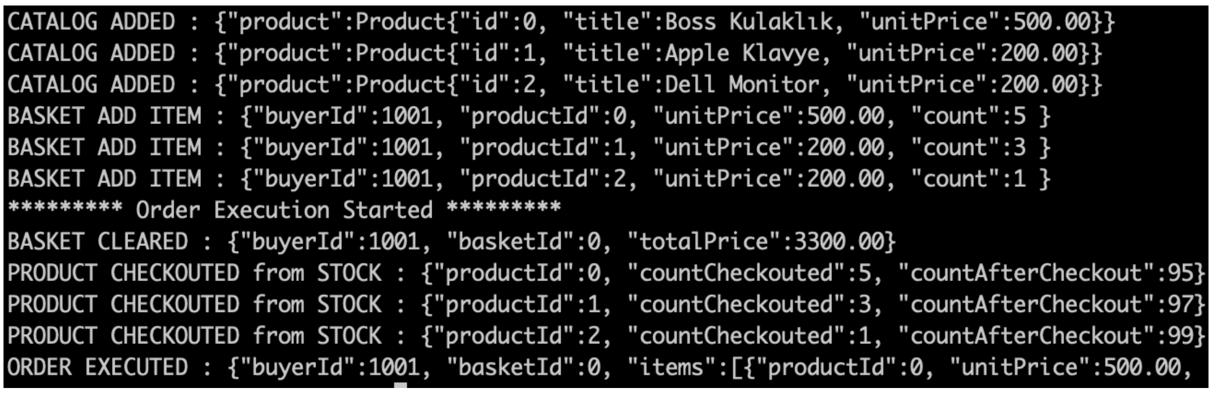
Let’s execute build.sh file

```
./build.sh
```

build.sh/**build.bat** executable file :

- creates/clear folders
- compiles java files module by module
- packages them
- finally executes application.

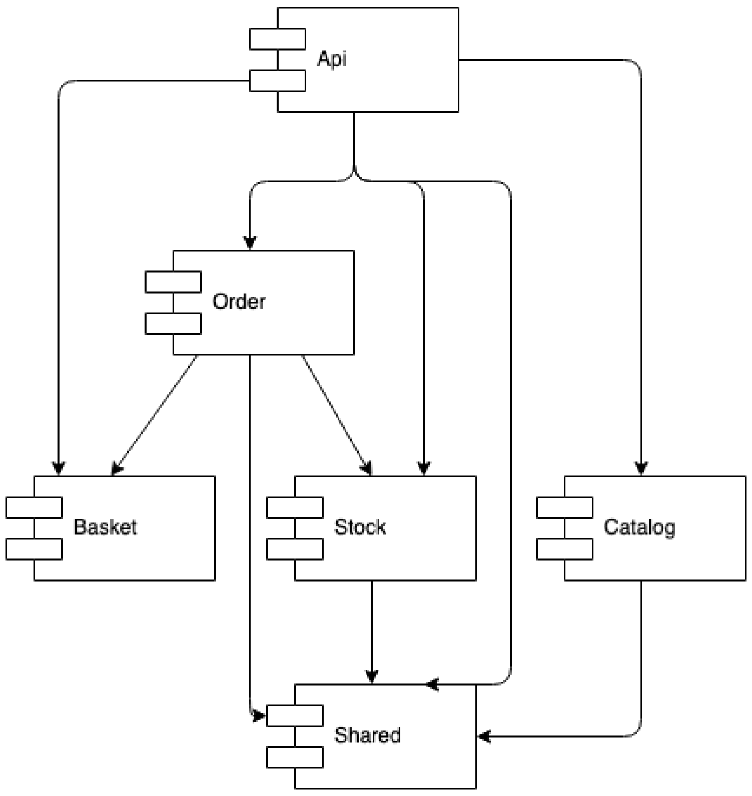
3. Be sure that taking result below



```
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,
```

MISSION DEFINITION

Monolith application have already separated to modules for you. You don't have to make any change on the codes. Let's review the dependency diagram of modules.



As you see in diagram above, Api module is depended to all modules. Almost all modules are depended to shared module. Order module is also depended to both Basket module and Stock module. It's a little bit chaotic, right? The dependency between Order, Basket and Stock is undesirable. We will remove the dependency in the future missions.

In this mission, we will define maven modules as Java Modules.

Let's meet Java Module Api coming with Java 9. As you know, Java uses ClassLoader to load classes in classpath. But it brings some issues such as:

1. It allow us to put same jar with different names into classpath.
2. It allow us to put same project jar with different version into classpath. Classloader doesn't give any guarantee to load correct version of jar. It could load an old version of jar. That causes problems on runtime.
3. ClassLoader allows to access all classes in jar even private field/method via reflection.
4. It doesn't provide an environment to have true encapsulation.

Java Module Api fixes the issues above:

1. It prevents locating a module with different name of jar files in MODULEPATH.
2. Only public fields and methods could be accessed by other modules.
3. It looks at module-info.java file to solve exported packages and module dependencies.

Java Module Api has three types of module:

1. **NAMED MODULES :** Module that has module-info.java file and located in MODULEPATH
2. **AUTOMATED MODULES :** Module that hasn't module-info.java file but that is located in MODULEPATH. Java gives a name that generated from name of jar file to this kind of module.
3. **UNNAMED MODULES :** Modules that isn't located in MODULEPATH but in CLASSPATH.

Accessibility between those module types takes place under current certain rules. We will see the rules in future missions.

As we say above, Java needs module-info.java file to be located in module in order to define it as named module. module-info.java file is different than java file we already know. It has special keywords. Basic definition of module-info.java :

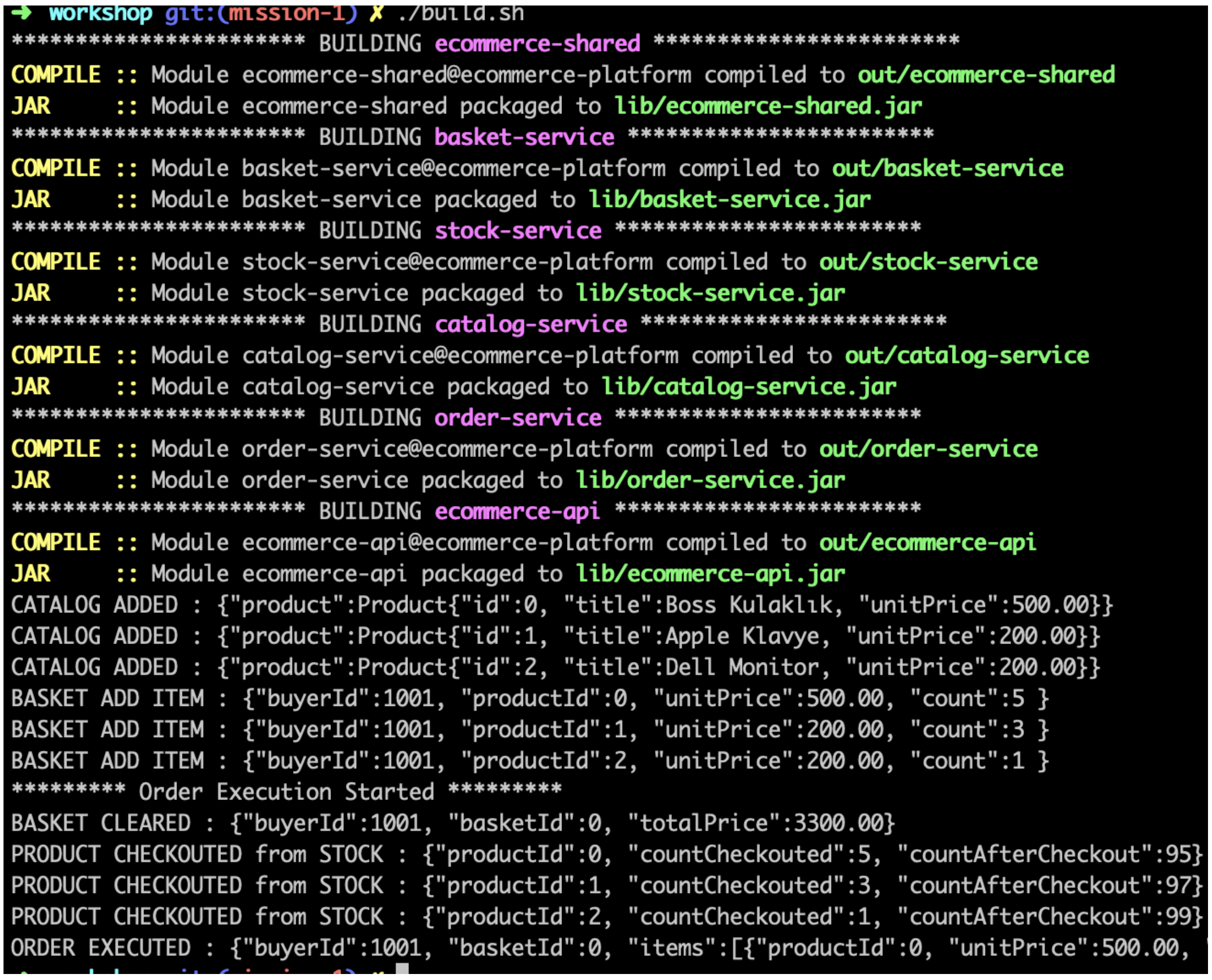
```
module [MODULE_NAME] {
    exports [PAKCKAGE_TO_EXPORT];
    ....
    exports [PAKCKAGE_TO_EXPORT];
    requires [MODULE_NAME];
}
```

- **EXPORTS :** This directive allows other modules to access public classes in specified package. The specified package must have a class at least
- **REQUIRES** : This directive defines dependency to other module.

MISSION TASKS

After giving informations above, let's start to define maven modules as java modules:

1. Let's check our application is working by executing build.sh/build.bat file. Now we can call our classic jar file as Unnamed Module. This execution will be occurred by using unnamed modules.



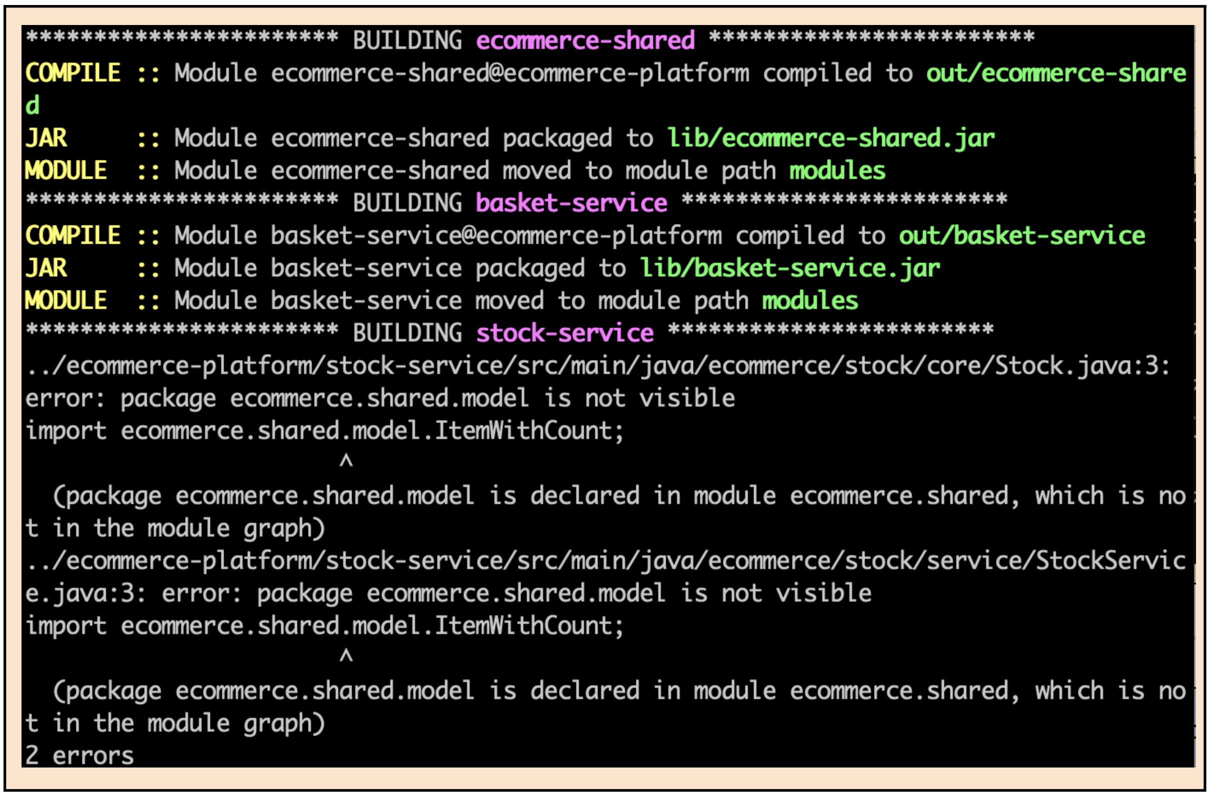
```
➜ workshop git:(mission-1) ✗ ./build.sh
*********************** BUILDING ecommerce-shared ***********************
COMPILE :: Module ecommerce-shared@ecommerce-platform compiled to out/ecommerce-shared
JAR     :: Module ecommerce-shared packaged to lib/ecommerce-shared.jar
*********************** BUILDING basket-service ***********************
COMPILE :: Module basket-service@ecommerce-platform compiled to out/basket-service
JAR     :: Module basket-service packaged to lib/basket-service.jar
*********************** BUILDING stock-service ***********************
COMPILE :: Module stock-service@ecommerce-platform compiled to out/stock-service
JAR     :: Module stock-service packaged to lib/stock-service.jar
*********************** BUILDING catalog-service ***********************
COMPILE :: Module catalog-service@ecommerce-platform compiled to out/catalog-service
JAR     :: Module catalog-service packaged to lib/catalog-service.jar
*********************** BUILDING order-service ***********************
COMPILE :: Module order-service@ecommerce-platform compiled to out/order-service
JAR     :: Module order-service packaged to lib/order-service.jar
*********************** BUILDING ecommerce-api ***********************
COMPILE :: Module ecommerce-api@ecommerce-platform compiled to out/ecommerce-api
JAR     :: Module ecommerce-api packaged to lib/ecommerce-api.jar
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,
```

2. Let’s list the modules that coming with Java Installation:

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

3. Let’s execute application with named modules
    a. open build.sh/build.bat file and change unnamed keyword to named in module definition section
    b. execute build.sh/build.bat file

😟 Do not panic. This execution will be failed. We will fix through the mission.



```
*********************** BUILDING ecommerce-shared ************************
COMPILE :: Module ecommerce-shared@ecommerce-platform compiled to out/ecommerce-share
d
JAR     :: Module ecommerce-shared packaged to lib/ecommerce-shared.jar
MODULE  :: Module ecommerce-shared moved to module path modules
*********************** BUILDING basket-service ***********************
COMPILE :: Module basket-service@ecommerce-platform compiled to out/basket-service
JAR     :: Module basket-service packaged to lib/basket-service.jar
MODULE  :: Module basket-service moved to module path modules
*********************** BUILDING stock-service ***********************
../ecommerce-platform/stock-service/src/main/java/ecommerce/stock/core/Stock.java:3:
error: package ecommerce.shared.model is not visible
import ecommerce.shared.model.ItemWithCount;
                       ^
  (package ecommerce.shared.model is declared in module ecommerce.shared, which is no
t in the module graph)
../ecommerce-platform/stock-service/src/main/java/ecommerce/stock/service/StockServic
e.java:3: error: package ecommerce.shared.model is not visible
import ecommerce.shared.model.ItemWithCount;
                       ^
  (package ecommerce.shared.model is declared in module ecommerce.shared, which is no
t in the module graph)
2 errors
```

c. examine the output of execution

✓ ecommerce-shared and basket-service has been created but ❌an error has been occured while packaging stock-service.

The key message to solve this error is that *“package ecommerce.shared.model is not visible”*.

Next to it, there is a description that says *“package ecommerce.shared.model is declared in module ecommerce.shared, which is not in the module graph”*

Visibility and module-graph are new terms, right? So, what should we do now?

build.sh/build.bat packages project modules under *modules* path when we sign modules as *named.* Java Module Api seeks under the specified module path for depended modules on both compile-time and runtime.

d. To check that ecommerce-shared.jar and basket-service.jar are available, Let’s look at under *modules* path.
e. Let’s list modules again by adding *modules* path

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

--module-path option is being used to specify module paths.

You can see that ecommerce-shared and basket service are being listed as **Automatic Module** bottom of the list.

Automatic Module is one of three module types. It is a module that doesn't have a module-info.java and also must be located in specified module paths.

Second module type is **Named Module** that has module-info.java and located in specified module paths.

4. Let's convert shared module to **Named Module** by adding module-info.java file.
    a. Create a module-info.java file in src/main/java
    b. Add the content below to module-info.java file we created.

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

exports directive makes ecommerce.shared.model package visible to world. It's the solution of *"is not visible"*.

    c. Let's run build.sh/build.bat
    d. Let's list modules in specified module-path

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

We can see that ecommerce.shared is being listed as **named module** but basket.service is still **automatic module**.

5. Let's convert basket service to Named Module by adding module-info.java file.
    a. Create a module-info.java file in src/main/java
    b. Add the content below to module-info.java file we created.

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

    c. Let's run build.sh/build.bat
    d. Let's list modules in specified module-path

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

We can see that ecommerce.shared is being listed as **named module**, too**.**

6. Let’s convert stock service to Named Module by adding module-info.java file.
    a. Create a module-info.java file in src/main/java
    b. Add the content below to module-info.java file we created.

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

Stock service needs to access ItemWithCount class in shared module. Cause of that we must define a dependency to ecommerce.shared module. **requires** directive defines the dependency.

    c. Let’s run build.sh/build.bat
    d. Let’s list modules in specified module-path

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

We can see that stock.service is being listed as **named module**, too**.**

7. Let’s convert catalog service to Named Module by adding module-info.java file.
    a. Create a module-info.java file in src/main/java
    b. Add the content below to module-info.java file we created.

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

    c. Let’s run build.sh/build.bat
    d. Let’s list modules in specified module-path

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

We can see that catalog.service ared is being listed as **named module**, too**.**

8. Let’s convert order service to named module by yourself without directive.

   a. **Please use dependency diagram at the start of mission.**
   b. **Make ecommerce.order.service package visible to the world.**

9. Let’s convert main application ecommerce-api to named module
   a. Create a module-info.java file in src/main/java
   b. Add the content below to module-info.java file we created.

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

   c. Let’s run build.sh/build.bat

😟 Do not panic. This execution will be failed. But we believe you can fix it. The only thing you can do pay attention to error message.

If you still try to fix it, please ask for help.

   d. After fixing, let’s run build.sh/build.bat
   e. Be sure that the application is running
   f. Let’s list the modules

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

   g. Be sure that our modules are being listed as named module

As you see, there are too many unnecessary modules listed that are coming from default JVM Module Path. We can limit the list by using **--limit-modules [module-name,...]** option.

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

You can see java.base module is being listed in spite of limiting. Because, java.base is default module. You don’t have to specify this dependency on module-path.java file.

h. Let’s inspect java.base module to get which packages are visible. We will use **--describe-module** parameter.

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

i. Let’s execute same command for inspect ecommerce.api

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

WHAT WE LEARN :

1. How to convert a Maven Module to Java Module
2. How to specify a module path with using --module-path param
3. How to create Named Module
4. How to create Automatic Module
5. How to create module-info.java file
    a. Make a package visible
    b. Depend to a module
6. How to list modules in module-path and limit them
7. How to inspect a module without unjar

MISSION - 2 : Make modules decouple

PREPARATION

Clean our workspace:

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

Switch to mission-2 branch:

```
git checkout mission-2
```

Clone JEventbus project from Github:

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

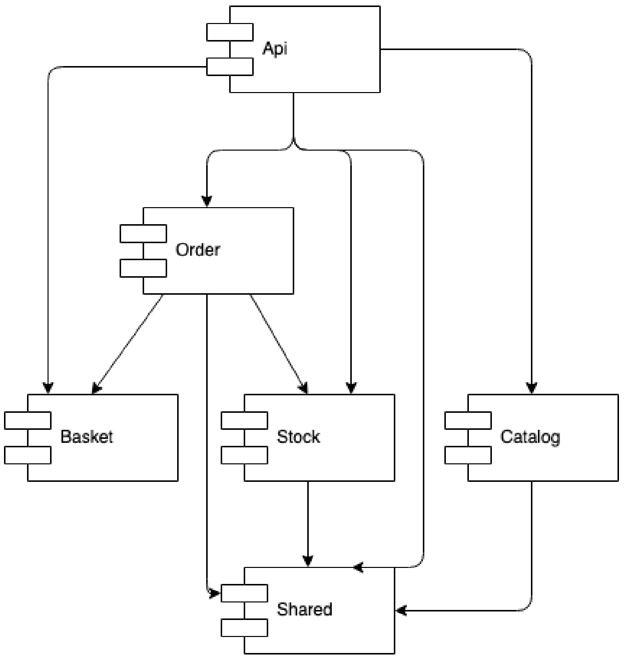
```
???????????????????????
```

MISSION DEFINITION

Let's remember what we have done in Mission-1:

- We have segregated Boundary Contexts to modules
- We have defined modules as Java Module

Let's remember the module diagram in Mission-1:

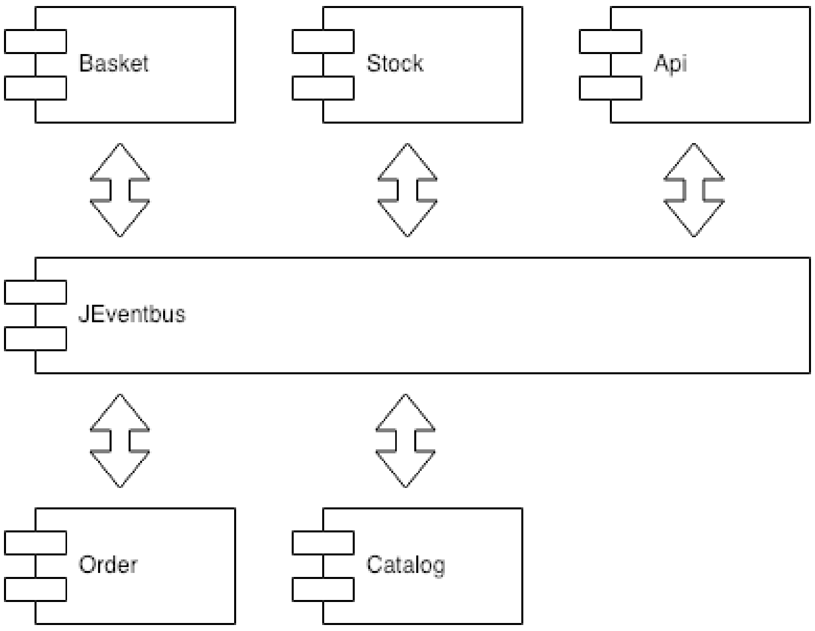


We can see that Order Module has depency to both Basket Module and Stock Module. These kind of dependencies are caused by a business rule “When an order is occured, basket must be cleaned and stock of items must be decreased as much as count of item ordered ”. If your goal is either having Microservice Architecture or segregate to boundary-contexts, kind of these dependencies are unacceptable. An approach to avoid kind of these dependencies is that implementing event-driven architecture.

In this mission, we are going to apply event-driven approach. It will allow us to :

- pushing events to an message broker system such as RabbitMQ, Kafka
- have a chance to apply event-sourcing

We will use a simple event mechanism that named as JEvenbus.



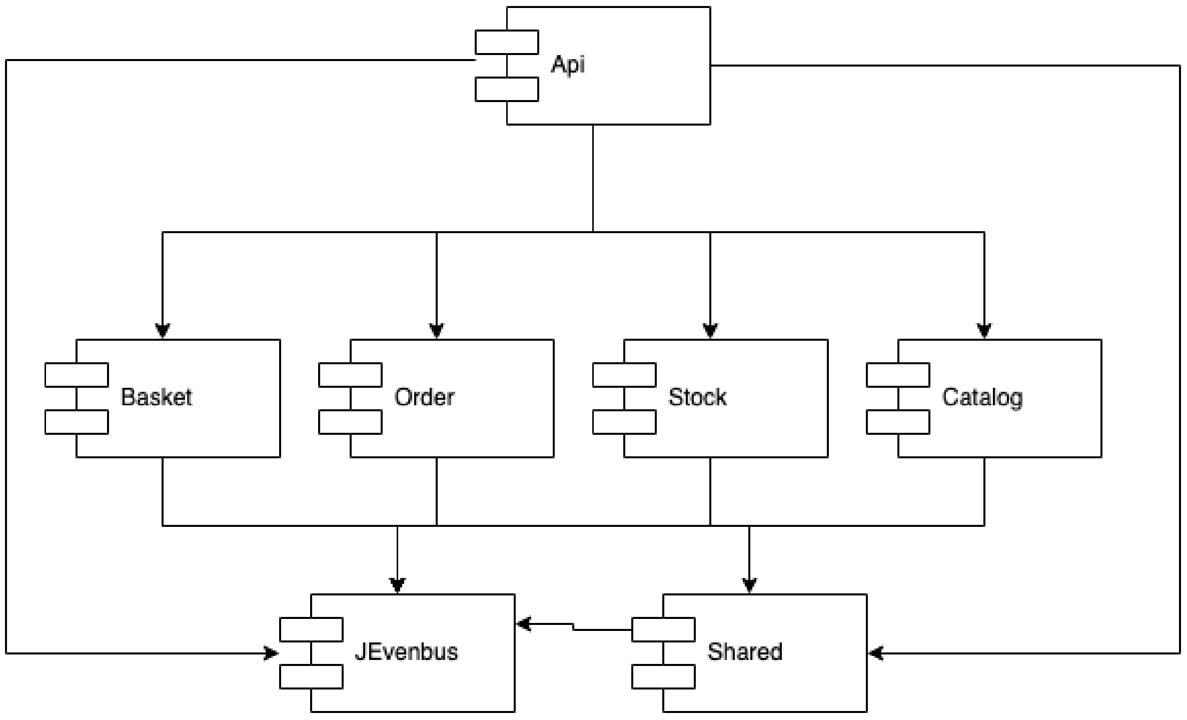
JEventbus uses reflection to trigger listeners when an event is fired. You can review **BKZ: EventPathListenerNode.java** file in JEventbus project**.** EventSource class has reference to object of Map<String,Object> to carry datas specific to event.

Creating event and Attaching listeners:

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

On the code above, BasketService and Stock Service were added to ORDER event as listener. ECommerceEventType is an enum and has method names to be going to call when event is fired. Reflection is being used to call the method named specified in enum.

The updated diagram fter removing dependencies to both Order and Stock by using JEventbus.



We have just learned that the event mechanism is using Reflection to call methods of Listeners. As you know, in order to call a private method by using Reflection, we have to make private method accessible by calling **setAccessible(true).**

This kind of usage of Recflection is named as **DEEP REFLECTION**. Accessibility between modules by using Reflection is restricted by rules.

With restrictions on DEEP REFLECTION and package visibility between modules, Java Module Api allows us to create modules that providing **STRONG ENCAPSULATION**.

In order to open a module to DEEP REFLECTION, we have to use OPEN directive. There are three usages of OPEN directive.

1. If you want to open whole module,

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

2. If you want to open a specific package,

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

1. If you want to open a specific package to specific module,

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

MISSION TASKS

Let’s integrate JEventbus module to project:

1. Create JEventbus module

```
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. Let’s check JEventbus module existence

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

3. Let’s learn that which package is being exported in JEventbus module

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

4. Let’s add JEventbus dependency to required modules - you can look at the module diagram above.
5. Let’s execute build.sh/build.bat file and fix the problem if exists.

😟 Do not panic. This execution will be failed. But we believe you can fix it. The only thing you can do pay attention to error message.

You might take **java.lang.reflect.InaccessibleObjectException.** When you take this exception, you know what you are going to do. You must open packages to DEEP REFLECTION.

ℹ️ You can use one of these directives OPENS [MODULE], OPENS, OPENS TO. Buy you should use OPENS TO in order to have secure and reliable project.

😁 CONGRATULATIONS!!! E-Commerce platform has an ideal architecture by let Order/Stock/Basket services communicate each other.

But we should make an refactoring because there is something that we don’t be aware. We have to fix it.

😟 ecommerce-api needs to access core package of both basket-service and order-service. These two services has opened the core packages of them to all of the world. This conditions brakes strong encapsulation.

ecommerce-api has a duty while accessing the core packages of basket-service and order-service. It converts messages to acceptable format between those services. This is inevitable.

How can we fix this situation? As **opens … to …** directive, we can export package to specified packages. We must use directive below:

```
exports [package] to  [module];
```

6. Let’s limit accessing to core package by applying the directive above to package-info.java file of both service.
7. Let’s execute build.sh/build.bat file

MISSION - 3 : Let’s Integrate LOGGING as Service

PREPARATION

Clean our workspace:

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

Switch to mission-23 branch:

```
git checkout mission-3
```

Add logging project in Mission-3 path into editor we used.

MISSION DEFINITION

When we look at the source code, we would realize that codes of logging are everywhere. Logging is one of the main concerns of software programming. There are different ways to extract logging codes from inside of business codes. One of them is AOP, you know. But, we are going to extract logging to own module by positioning it as event consumer.

When you look at source of Logger class, you would see it’s defined as EventListener.

➳ Let’s review Logger.java class

While extracting logging into a module, we must provide :

1. Solution should support different logging methods
2. Solution should support that the logging method should be able to be changed at runtime.

We are going to do these requirements by using Service Provider feature of Java Module. Service Provider feature provides these requirements by adding special directives into module-info.java file. These directives tell jvm that which modules are service implementer and which modules are service contracts/interfaces.

These directives:

provides [INTERFACE] **with** [IMPLEMENTER];

ℹ This directive must be defined in module that has implementer class of service. It specifies that which interface is being implemented by which class. The service provider module must have dependency to module that has service interface.

uses [INTERFACE];

ℹ This directive must be defined in client module. It specifies that which service interface is being used by client module. The client module must have dependency to module that has service interface.

The mission-3 branch has three modules extra:

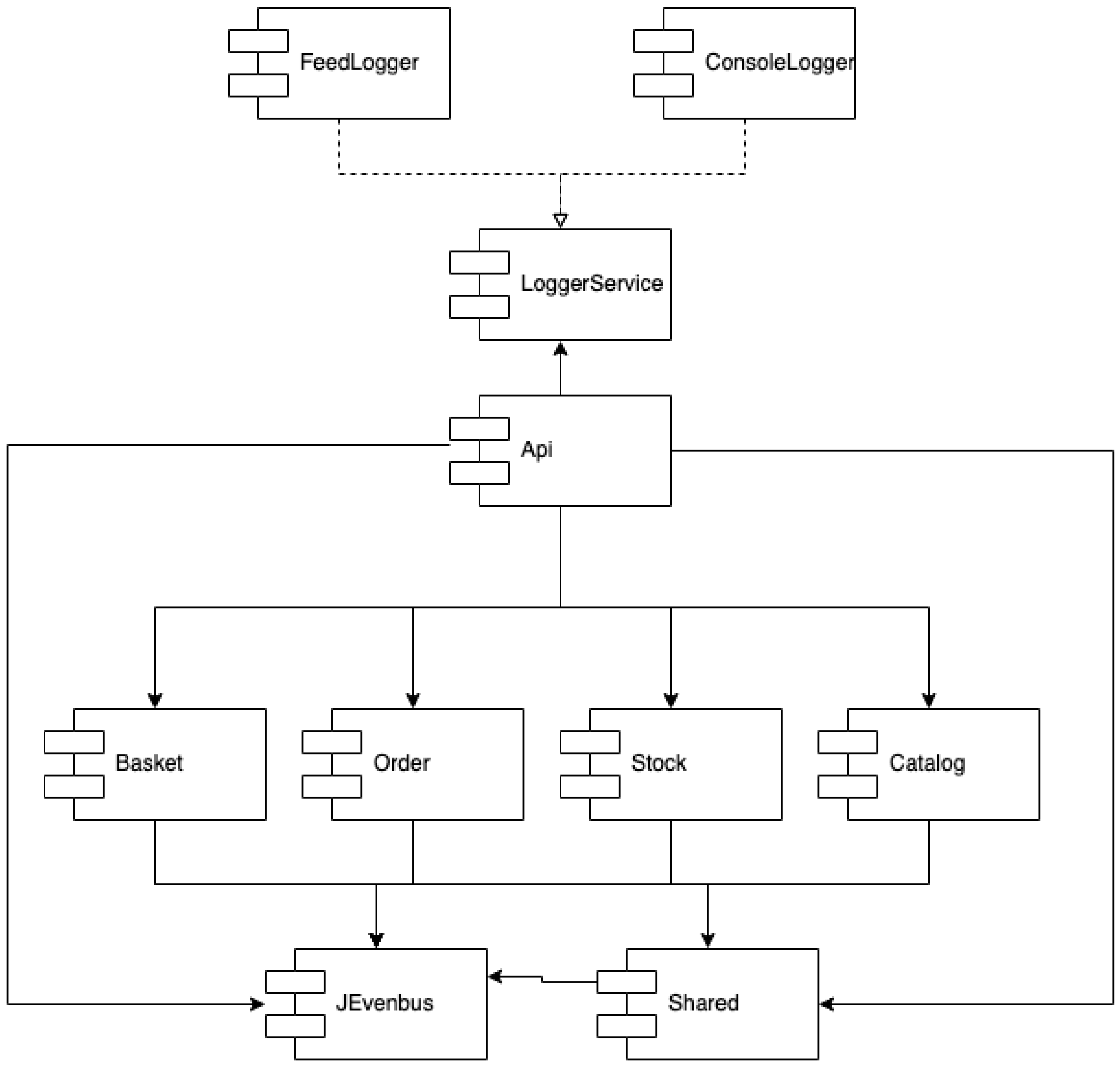
1. **logger-service** : This module has service interfaces/contracts.
2. **console-logger** : This module is a service provider. It prints logs to console.
3. **feed-logger** : This module is a service provider. It saves logs to database, but for this workshop it prints to console with special format.

Do you know that Java has a class, **java.util.ServiceLoader**, that allows to inject service provider in runtime. This class exists since Java 1.3 but it was being used for internal purposes. After Java 1.6, this class was opened to external usages.

Usage of ServiceLoader is as below. In this usage, service providers are being put into map with name key. getServiceName() is a method defined in LoggerService.

```
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);
        }
    }
```

Module diagram will become as below after integration of logging service.



MISSION TASKS

Let’s integrate logging service and it’s providers to our project;

1. Let’s convert logger-service to Java Module
    a. Create module-info.java file
    b. Give “logger.service” name to module
    c. Export packages
2. Let’s convert console-logger to Java Module
    a. Create module-info.java file
    b. Give “console.logger” name to module
    c. Define logger.console.ConsoleLogger class as implementer of logger.service.LoggerService interface

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

    d. Don’t forget to define dependency to logger.service module
3. Let’s convert feed-logger to Java Modüule
    a. Create module-info.java file
    b. Give “fee.logger” name to module
    c. Define logger.console.FeedLogger class as implementer of logger.service.LoggerService interface

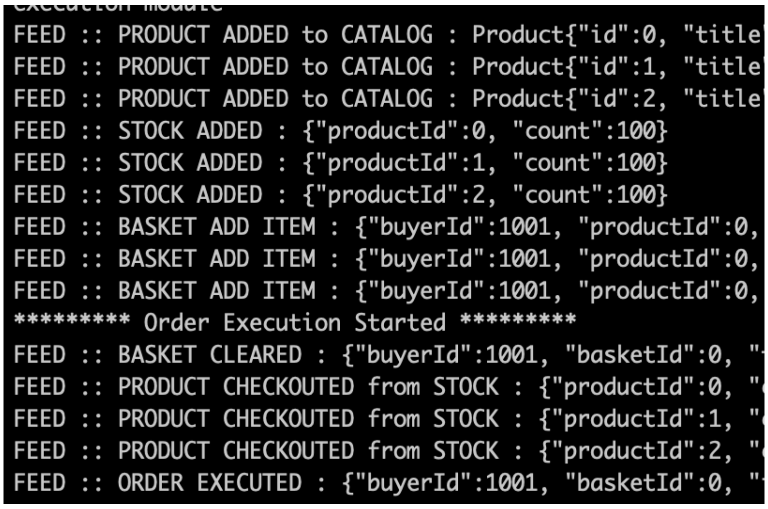
```
provides logger.service.LoggerService with logger.feed.FeedLogger;
```

    d. Don’t forget to define dependency to logger.service module
4. Let’s define usage of service

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

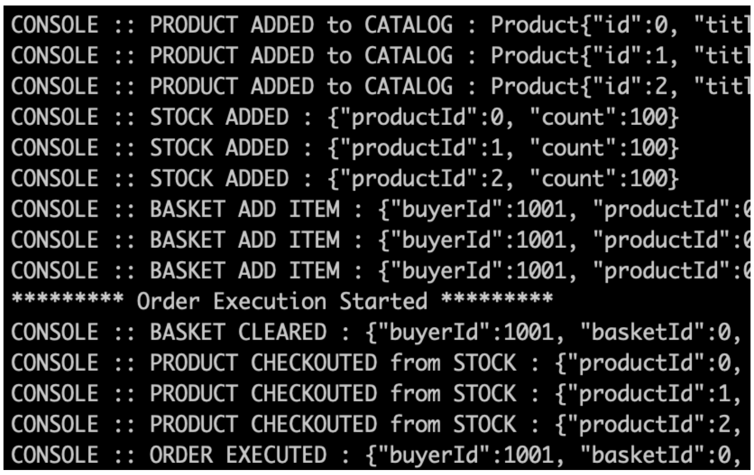
Don’t forget to define dependency to logger.service module

5. Let’s run build.sh/build.bat



```
execution module
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. Let’s change feed to console on 24th line of Application.java
7. Let’s run build.sh/build.bat



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

MISSION - 4 : DEPLOYMENT

PREPARATION

Clean our workspace

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

Switch to mission-4 branch

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

MISSION DEFINITION

We now can pack the application then deploy it.

Another feature coming with Java 9 is that creating RUNTIME IMAGE. We don’t have to install JRE in order to execute our application. Runtime Image feature allows us to create our own custom JRE that contains application and depended modules.

We can list the benefits of this feature:

1. Increase security by putting known modules to image.
2. Decrease start time of application because JVM links modules whiles creating runtime image. While starting application by classic way, JVM links modules and it causes performance lost.
3. Size of runtime image is less.
4. Java Version of runtime image couldn’t not be changed.
5. It’s suitable to locate in container.
6. You can start Runtime Image without any installation.

JLINK command creates Runtime Image but before that we would learn what **JDEPS** command do. JDEPS command analyzes dependency graph of a created module.

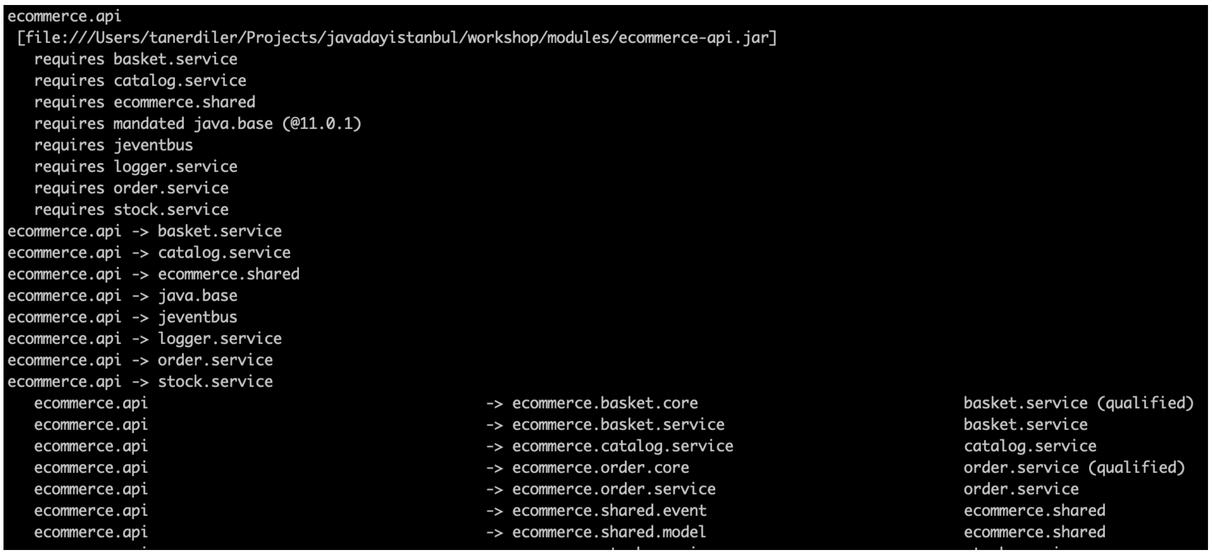
Let’s remember which commands we have used to make an analyzes module;

1. **java --module-path [PATH] --list-modules** : list all modules at specified path
2. **java --module-path [PATH] --limit-modules [MODULE-1] --list-modules** : lists only modules at module graph of MODULE-1.
3. **java --module-path … --describe-module …** : display content of module-info.java of specified module

Let’s start to use JDEPS command

1. Let’s run build.sh/build.bat
2. Let’s anaylze dependency graph of ecommerce.api module

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



```
ecommerce.api
 [file:///Users/tanerdiler/Projects/javadayistanbul/workshop/modules/ecommerce-api.jar]
   requires basket.service
   requires catalog.service
   requires ecommerce.shared
   requires mandated java.base (@11.0.1)
   requires jeventbus
   requires logger.service
   requires order.service
   requires stock.service
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
   ecommerce.api                                      -> ecommerce.basket.core                              basket.service (qualified)
   ecommerce.api                                      -> ecommerce.basket.service                           basket.service
   ecommerce.api                                      -> ecommerce.catalog.service                          catalog.service
   ecommerce.api                                      -> ecommerce.order.core                               order.service (qualified)
   ecommerce.api                                      -> ecommerce.order.service                            order.service
   ecommerce.api                                      -> ecommerce.shared.event                             ecommerce.shared
   ecommerce.api                                      -> ecommerce.shared.model                             ecommerce.shared
```

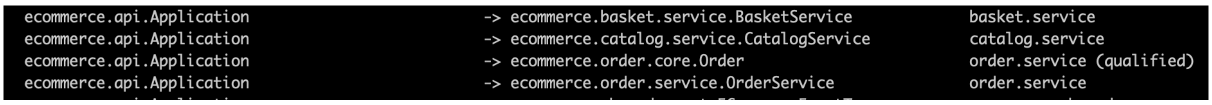
We get the result with three sections seen as above:

- First Section : content of module-info.java file.
- Second Section : list of depended modules
- Third Section : list of dependencies at package level. We can interpret a line as “***basket.service*** *module is being used by any class in* ***ecommerce.basket.core*** *package of* ***ecommerce.api*** *“*

3. Let’s re-execute anaylzes on class level

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

You can see that classes is being listed instead of package on third column of third section.



```
ecommerce.api.Application                          -> ecommerce.basket.service.BasketService           basket.service
ecommerce.api.Application                          -> ecommerce.catalog.service.CatalogService         catalog.service
ecommerce.api.Application                          -> ecommerce.order.core.Order                       order.service (qualified)
ecommerce.api.Application                          -> ecommerce.order.service.OrderService             order.service
```

4. Let's filter for a specific package. We will add **-regex [EXP]** parameter to the command.

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

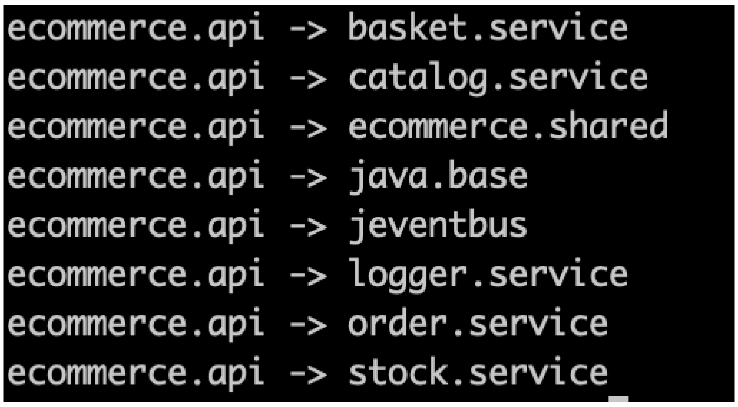
5. Let's filter for a specific class.

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

You will see that jdeps executed the analyzes deeply on whole module graph of ecommerce.api when you add --regex option.

6. If you say "these information is more than I need. I just want to learn depended module names ", you should use **-summary** option.

```
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
```

7. Let's do analyzes on diagram of dependencies

```
jdeps --module-path modules --dot-output out --module ecommerce.api
```

There are two ways to display created file as image:

a. You can visualize dot file on https://dreampuf.github.io/GraphvizOnline
b. Execution of “**dot -Tpng -O out/ecommerce.api.dot**” command. It needs installation of Graphviz’in CLI

We will use JLINK command while creating Runtime Image. Let’s create our Runtime Image:

1. Let’s execute the command below

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

   This command creates executable image base on module graph of ecommerce.api

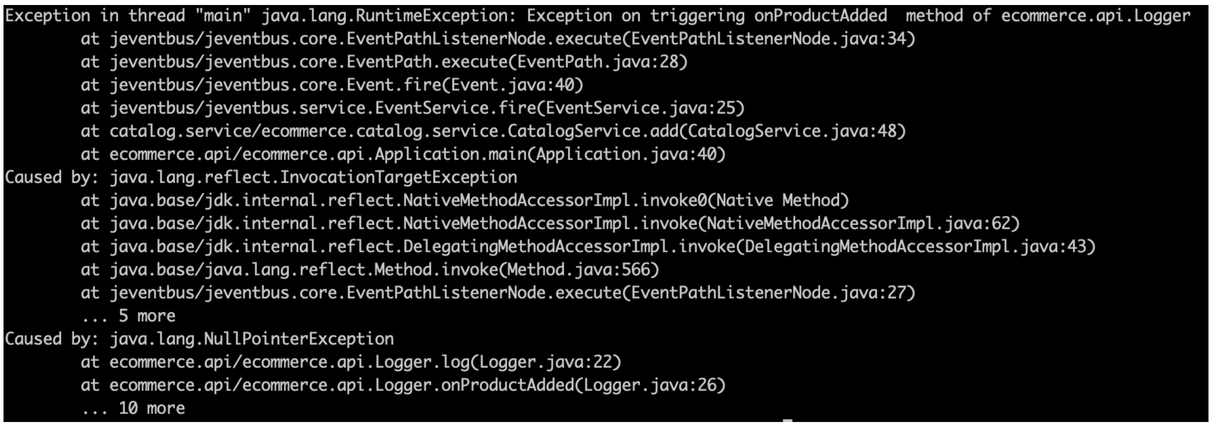
2. Let’s check out/ecomm-api folder either created
3. Let’s run image
   a. One way is that calling ecom specified with --launcher option while executing previous command

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

   b. Other way is that calling java by specifying module with class has main method

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

   c. When we execute both command, we would take exception below



```
Exception in thread "main" java.lang.RuntimeException: Exception on triggering onProductAdded  method of ecommerce.api.Logger
	at jeventbus/jeventbus.core.EventPathListenerNode.execute(EventPathListenerNode.java:34)
	at jeventbus/jeventbus.core.EventPath.execute(EventPath.java:28)
	at jeventbus/jeventbus.core.Event.fire(Event.java:40)
	at jeventbus/jeventbus.service.EventService.fire(EventService.java:25)
	at catalog.service/ecommerce.catalog.service.CatalogService.add(CatalogService.java:48)
	at ecommerce.api/ecommerce.api.Application.main(Application.java:40)
Caused by: java.lang.reflect.InvocationTargetException
	at java.base/jdk.internal.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
	at java.base/jdk.internal.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccessorImpl.java:62)
	at java.base/jdk.internal.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessorImpl.java:43)
	at java.base/java.lang.reflect.Method.invoke(Method.java:566)
	at jeventbus/jeventbus.core.EventPathListenerNode.execute(EventPathListenerNode.java:27)
	... 5 more
Caused by: java.lang.NullPointerException
	at ecommerce.api/ecommerce.api.Logger.log(Logger.java:22)
	at ecommerce.api/ecommerce.api.Logger.onProductAdded(Logger.java:26)
	... 10 more
```

   d. Let’s examine that the create image has any provider of logger service
      i. Let’s check by using JDEPS command

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

We will see that image doesn't contain neither console.logger nor feed.logger.

ii. Another way to exam is that calling java command with --list-modules options. You know that create image is a kind of JRE so that it would include java command in /bin folder.

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

We will see that image doesn't contain neither console.logger nor feed.logger.

jlink command uses module graph of specified modules in order to create image. As you remember, module-info.java file of ecommerce.api doesn't include any directive for dependency to either feed.logger or console.logger. only dependency to logger.service. --bind-services option jlink adds service providers automatically to image.

4. Let's re-create image by adding --bind-services option

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

You will see that image will run successfully

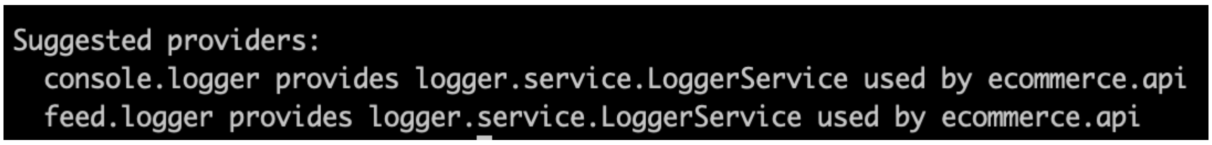
```
./out/ecomm-api/bin/ecom
```

😟 --bind-services option allows jlink to add all service providers into create image. That will cause adding unwanted modules and increasing size of image. The suggested way is adding service provider modules one by one.

5. Let's find out that we will add which service provider modules. In order to do that we will execute jlink with **--suggest-providers** option:

```
jlink --module-path modules --suggest-providers logger.service.LoggerService
```

Output would be:



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

6. Let's re-create image by adding modules of service providers we want.:

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

Execute image then be sure it will run successfully.

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

7. What is size of created image and do we have a chance to decrease size?

    a. Let's check size of image

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

    b. Let's re-create image by adding new options that excludes some datas

```
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
```

Let's check size of image

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

😁 We can see the size is less than previous

    c. Let's check image size for different compression levels and fill the table below

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

--compress=2	???

We made our Runtime Image smaller enough. It’s ready to deploy by using classic way of SSH. But we will deploy it with modern way, dockerizing. You must be sure that you have Docker Engine and Docker CLI installed.

You will see there is a Dockerfile coming with Mission-4 branch.

1. Let’s review Dockerfile.

You will see there are two stages defined in Dockerfile. In stage named as Build, Runtime Image is being created by using JLink. The main reason of needing Build stage is that Runtime Image created is not cross-platform. So suggested way is that creating image on server has same specifications with server that image will run in.

2. Let’s create Docker image and run it as container

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

3. Let’s check the size of created image

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

😟 We used Alpine Linux as base image. Size of Alpine Linux is approximately 4Mb. You will see that glibc is being added on Build stage. Alpine Linux is using **musl libc** differently to other linux distributions. OpenJDK is not compatible with musl libc. But it will be compatible with Portola project.

MISSION - 5 : AUTOMATIC MODULE & UNNAMED MODULE

PREPARATION

Clean workspace

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

Switch to Mission-5 branch

```
git checkout mission-5
```

MISSION DEFINITION

We used Named Modules until now. Let’s work with Automatic Module and Unnamed Module.

Automatic Modules are the modules that are located in module path without having module-info.java. All of packages in automatic module are open to access. JVM doesn’t generate a module graph cause of there is no module-info.java. Dependency resolution is not be doing and we have to add dependencies one by one with --add-modules option.

Here is the list of dependencies of modules:

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

1. Let's execute build.sh/build.bat

😟 Do not panic. This execution will be failed. We will take NullPointerException cause of Logging service provider is not found. Because, JVM will not find service providers. We need to specify them with another way. As we say before, service provider features is supported since 1.3.

2. In feed-logger maven module;
    a. Let's create src/main/**resource/META-INF/services** path
    b. Let's create a file named as **logger.service.LoggerService**
    c. Let's write the implementer class with package ,**logger.feed.FeedLogger**, into file we've created
3. In console-logger maven module;
    a. Let's create src/main/**resource/META-INF/services** path
    b. Let's create a file named as **logger.service.LoggerService**
    c. Let's write the implementer class with package ,**logger.console.ConsoleLogger,** into file we've created
4. Let's execute build.sh/build.bat

ℹ️ JVM gives us a option to make naming of Automatic Modules permanent. You put the definition below into META-INF/MANIFEST.MF file.

Automatic-Module-Name: <module-name>

ℹ️ Unnamed Modules are the classic jar files that are specified Classpath. Unnamed modules can access all packages in a module but named modules doesn't access an unnamed module cause of security concerns.

5. You can try interaction between unnamed, automatic and named modules by changing type of modules specified in build.sh/build.bat