Dagger 2

https://google.github.io/dagger/

1. 주입이 필요한 Interface와 그 Interface를 구현한 Class 작성. 이 때, 생성자 주입이 필요한 경우 생성자에 @Inject.
    1. Module(객체 공급자)을 작성한다. 각 함수는 “provide”라는 접두사를 붙인다.
2. 1에서 작성한 각 Bean을 이용하는 Context역할. 즉, 각 Bean을 주입받을 Class를 작성한다.
    1. Component(객체를 사용할 수 있도록 다리 역할을 하는 연결자)을 작성한다. 이 때 2에서 작성한 Class Type의 Setter 혹은 Getter를 작성한다.

Framework, Dependency Injection

Dagger 공식 홈에 있는 CoffeeMaker 예제 단순화.

pom.xml

```
...

    <dependencies>
        <dependency>
            <groupId>com.google.dagger</groupId>
            <artifactId>dagger</artifactId>
            <version>${dagger.version}</version>
        </dependency>
    </dependencies>

    <build>
        <plugins>
            <plugin>
                <groupId>org.apache.maven.plugins</groupId>
                <artifactId>maven-compiler-plugin</artifactId>
                <version>3.6.1</version>
                <configuration>
                    <annotationProcessorPaths>
                        <path>
                            <groupId>com.google.dagger</groupId>
                            <artifactId>dagger-compiler</artifactId>
                            <version>${dagger.version}</version>
                        </path>
                    </annotationProcessorPaths>
                </configuration>
            </plugin>
        </plugins>
    </build>

...
```

주입 할 대상

커피 머신이 작동하려면 커피 물을 끓일 Heater, 달여진 커피를 받을 Pump가 필요.

- Heater

```
public interface Heater {
    void on();
    void off();
    boolean isHot();
}
```

```
public class MyHeater implements Heater {
    private boolean heating;

    public void on() {
        System.out.println("~ ~ ~ heating ~ ~ ~");
        this.heating = true;
    }

    public void off() {
        this.heating = false;
    }

    public boolean isHot() {
        return heating;
    }
}
```

- Pump

```
public interface Pump {
    void pump();
}
```

```
public class MyPump implements Pump {
    private final Heater heater;

    @Inject
    public MyPump(Heater heater) {
```

```
        this.heater = heater;
    }

    public void pump() {
        if (heater.isHot()) {
            System.out.println("=> => pumping => =>");
        }
    }
}
```

Module

각 함수에 “provide” 접두사를 붙인다.

```
@Module
public class CoffeeMakerModule {

    @Provides
    Heater provideHeater() {
        return new MyHeater();
    }

    @Provides
    Pump providePump(MyPump pump) {
        return pump;
    }
}
```

주입 받을 대상

```
public class CoffeeMaker {
    @Inject
    Heater heater;

    @Inject
    Pump pump;

    @Inject
    public CoffeeMaker(Heater heater, Pump pump){
        this.heater = heater;
        this.pump = pump;
    }

    public void brew(){
        heater.on();
```

```
        pump.pump();
        System.out.println("brew!");
        heater.off();
    }
}
```

Component

```
@Component(modules = CoffeeMakerModule.class)
public interface CoffeeComponent {
    CoffeeMaker maker(); // getter 작성
}
```

Context

먼저 Build를 하여 DI가 적용된 코드가 생성되도록 한다.

```
public class CoffeeApp {
    public static void main(String... args) {
        CoffeeComponent coffeeComponent = DaggerCoffeeComponent.create();
        coffeeComponent.maker().brew();
    }
}
```

```
~ ~ ~ heating ~ ~ ~
=> => pumping => =>
brew!
```