Day 01 (2019-01-07)

1. Java 기본

(1) 객체지향

객체지향의 3요소

- Encapsulation (캡슐화)
- Inheritance (상속)
    - IS-A, HAS-A
    - 상속보다는 조립
- Polymorphism (다형성)

객체지향의 5원칙 (SOLID 원칙) (참고자료)

- SRP (Single Responsibility Principle) - 단일책임 원칙
    - 하나의 클래스는 하나의 책임만 가져야 한다
      → 어떤 변화 (요구사항의 변화 등)에 의해 클래스를 변경해야 하는 이유는 오직 하나이어야 한다
    - 나머지 4원칙의 기초가 되는 원칙으로 SRP만 잘 지키면 다른 책임의 변경으로 인한 연쇄작용을 방지할 수 있다.
    - ReviewController, ShopService(x)
    - WriteReviewController, RemoveShopService(o)

- OCP (Open-Closed Principle) - 개방-폐쇄 원칙
    - SW의 구성요소(모듈, 컴포넌트, 클래스, 메소드)는 확장에는 열려있고 변경에는 닫혀있어야 한다.
      → 변경을 위한 비용은 가능한 줄이고 확장을 위한 비용은 가능한 극대화 한다
    - OCP를 가능하게 하는 중요 메커니즘은 추상화와 다형성
    - 새로운 기능이 추가되는 경우 기존에 제공하던 클래스(또는 메소드)를 수정하는 것이 아니라 새로운 클래스(또는 메소드)를 추가해서 기능을 확장한다.

- LSP (The Liskov Subsitution Principle) - 리스코프 치완 원칙
    - 서브 클래스는 언제나 슈퍼 클래스를 대체할 수 있다.
      → 슈퍼 클래스가 들어갈 자리에 서브 클래스를 넣어도 원래대로 잘 작동해야 한다
    - 상속의 오용을 방지하게 하는 원칙
    - 슈퍼 클래스와 서브 클래스의 동작이 일관성 있게 동작해야 함

```
public interface Shape {
    Integer area();
}

class Rectangle implements Shape {
    private Integer width;
    private Integer height;

    public Integer getWidth() {
        return width;
    }

    public void setWidth(Integer width) {
        this.width = width;
    }

    public Integer getHeight() {
        return height;
    }

    public void setHeight(Integer height) {
        this.height = height;
    }

    @Override
    public Integer area() {
        return this.width * this.height;
    }
}

class Square extends Rectangle {

    @Override
    public Integer getWidth() {
        return super.getWidth();
    }

    @Override
    public void setWidth(Integer width) {
        super.setWidth(width);
        super.setHeight(width);
    }

    @Override
    public Integer getHeight() {
        return super.getHeight();
    }

    @Override
    public void setHeight(Integer height) {
        super.setWidth(height);
        super.setHeight(height);
    }

    @Override
    public Integer area() {
        return super.area();
    }
}

//....
@Test
public void __(Shape s){
   s.setWidth(5);
   s.setHeight(4);
   s.area(); //???
}
```

- ISP (Interface Segregation Principle) – 인터페이스 분리 원칙
    - 한 클래스는 자신이 사용하지 않는 인터페이스는 구현하지 않아야 한다.
      → 하나의 일반적인 인터페이스보다는, 여러 개의 구체적인 인터페이스가 낫다.
    - SRP가 클래스의 단일책임을 강조한다면 ISP는 인터페이스의 단일책임을 강조함.

- DIP (Dependency Inversion Principle) – 의존성 역전의 원칙
    - 고차원 모듈은 저차원 모듈에 의존하면 안된다 (자주 변경되는 또는 변경될 만한 구현체에 의존하지 말 것)
      → Layered Architecture와 같이 상하의 관계가 존재하는 구조에서 하위 레벨의 변경이 상위 레벨의 변경을 요구하는위계관계를 끊는 것

      고차원 모듈이 저차원 모듈에 직접적으로 의존할 때(직접적으로 SDK 구현체 사용 및 Model 참조)

```
@Service
public class FileService {
    @Autowired
    private AmazonS3SDKImpl amazonS3SDKImpl;

    @Autowired
    private AlibabaS3SDKImpl alibabaS3SDKImpl;

    public void save(Path path, ServiceType serviceType) {
        if(ObjectUtils.nullSafeEquals(serviceType, ServiceType.AMAZON)){
            this.amazonS3SDKImpl.save(new AmazonS3SDKImpl.AmazonS3Object(path));
        } else if(ObjectUtils.nullSafeEquals(serviceType, ServiceType.ALIBABA)){
                this.alibabaS3SDKImpl.save(new AlibabaS3SDKImpl.AlibabaS3Object(path));
        }
    }

    public void find(){
        //...
    }

    public void delete(){
        //...
    }
}

enum ServiceType {
    AMAZON, ALIBABA;
}

//SDK 1.0    ?
class AmazonS3SDKImpl {
    public void save(AmazonS3Object s3Object) {
        //Amazon SDK
    }

    public static class AmazonS3Object {
        private Path path;

        public AmazonS3Object(Path path) {
            this.path = path;
        }
    }
}

//SDK 1.1 AlibabaS3Object Deprecated ?
class AlibabaS3SDKImpl {
    public void save(AlibabaS3Object s3Object) {
        //Alibaba SDK
    }

    public static class AlibabaS3Object {
        private Path path;

        public AlibabaS3Object(Path path) {
            this.path = path;
        }
    }
}
```

고차원 모듈이 저차원 모듈에 의존하지 않고 저차원 모듈을 추상화 한 고차원 모듈에 의존할 때
(FileRepositoryFacade를 통해 저차원 모듈을 추상화 한 FileRepository 인터페이스를 의존)

```
@Service
public class AdvancedFileSaveService {
    @Autowired
    private FileRepositoryFacade facade;

    public void save(Path path, FileRepositoryFactory.ServiceType serviceType) {
        this.facade.save(serviceType, new ApplicationFile(path), applicationFile -> {
            //callback
        });
    }
}

@Component
class FileRepositoryFacade {
    private FileRepository amazonS3Repository;
    private FileRepository alibabaS3Repository;

    public FileRepositoryFacade (FileRepository amazonS3Repository, FileRepository
alibabaS3Repository) {
        this.amazonS3Repository = amazonS3Repository;
        this.alibabaS3Repository = alibabaS3Repository;
    }

    public enum ServiceType {
        AMAZON, ALIBABA;
    }

    public void save(ServiceType serviceType, ApplicationFile applicationFile,
Consumer<ApplicationFile> callback){
        this.getInstance(serviceType).save(applicationFile);
        callback.accept(applicationFile);
    }

    private FileRepository getInstance(ServiceType serviceType) {
                if(ObjectUtils.nullSafeEquals(serviceType, ServiceType.AMAZON)){
            return this.amazonS3Repository;
        } else if(ObjectUtils.nullSafeEquals(serviceType, ServiceType.ALIBABA)){
            return this.alibabaS3Repository;
                } else {
            throw new IllegalArgumentException();
        }
    }
}

//Interface  (Concrete)
interface FileRepository {
    void save(ApplicationFile applicationFile);
}

//Amazon S3 SDK
@Repository
class AmazonS3Repository implements FileRepository {
    //Amazon SDK dependency ...
    private AmazonS3SDKImpl amazonS3SDK;

    @Override
    public void save(ApplicationFile applicationFile) {
        //applicationFile AmazonS3SDK
    }
}

//Alibaba S3 SDK
@Repository
class AlibabaS3Repository implements FileRepository {
    //Alibaba SDK dependency ...
    private AlibabaS3SDKImpl alibabaS3SDK;

    @Override
    public void save(ApplicationFile applicationFile) {
        //applicationFile AlibabaS3SDK
    }
```

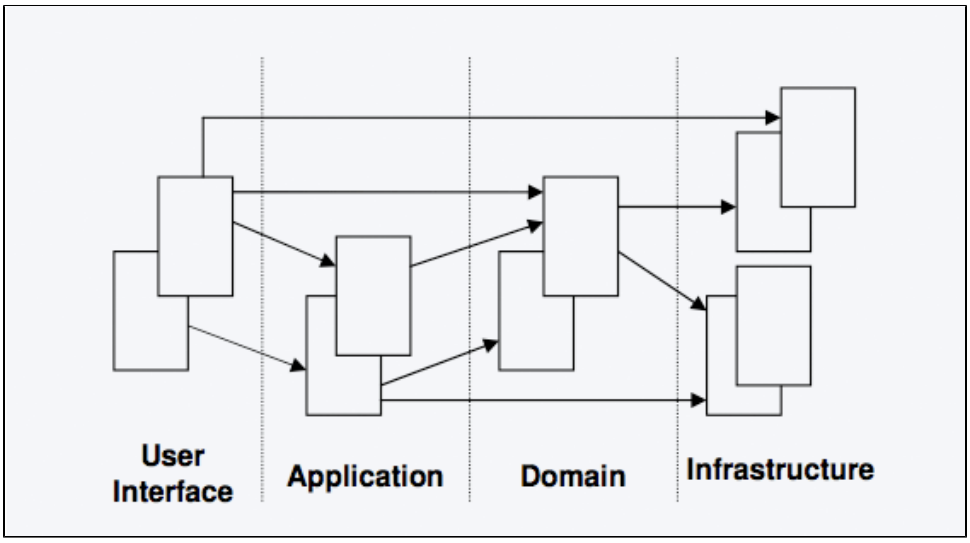
```
}

// SDK  Model
@Data
@AllArgsConstructor
class ApplicationFile {
    private Path path;

    public Boolean isAvaliableFile() {
        return this.path != null && Files.exists(this.path);
    }
}
```

※ DDD(Domain Driven Design) ?

- **Ubiquitous Language(보편적 의사소통 언어),**
- **Domain Model(**Entity, Value Object, Aggregate, Service, Repository, Factory, Domain Events**)**
- **Layered Architecture(**User interface, Application, Domain, Infrastructure**)**



```
interface
  ㄴ RetrieveShopController.java
  ㄴ RetrieveShopRequest.java
  ㄴ ShopResponse.java

application
  ㄴ RetrieveShopService.java

domain
  ㄴ Shop.java
  ㄴ ShopFactory.java

infrastructure
  ㄴ FindShopDao.java
  ㄴ ShopSearcherByDaumSearch.java
```

- Bounded Context

```
interface
  ㄴ shop
  ㄴ reservation

application
  ㄴ shop
  ㄴ product

domain
  ㄴ shop
  ㄴ product
  ㄴ reservation

dao
  ㄴ product
  ㄴ reservation
  ㄴ review
```

```
shopsearch
  ㄴ interface
  ㄴ application
  ㄴ domain
  ㄴ infrastructure

reservation
  ㄴ interface
  ㄴ application
  ㄴ domain
  ㄴ infrastructure

review
  ㄴ interface
  ㄴ application
  ㄴ domain
  ㄴ infrastructure
```

(2) Java에 대해 어디까지?

Future, Generic, Fork/Join Framework, NIO, Type inference

Functional Interface, Default method, Lambda, Stream, Map(computeIfAbsent, computeIfPresent ...), CompletableFuture, LocalDate, ZonedDateTime

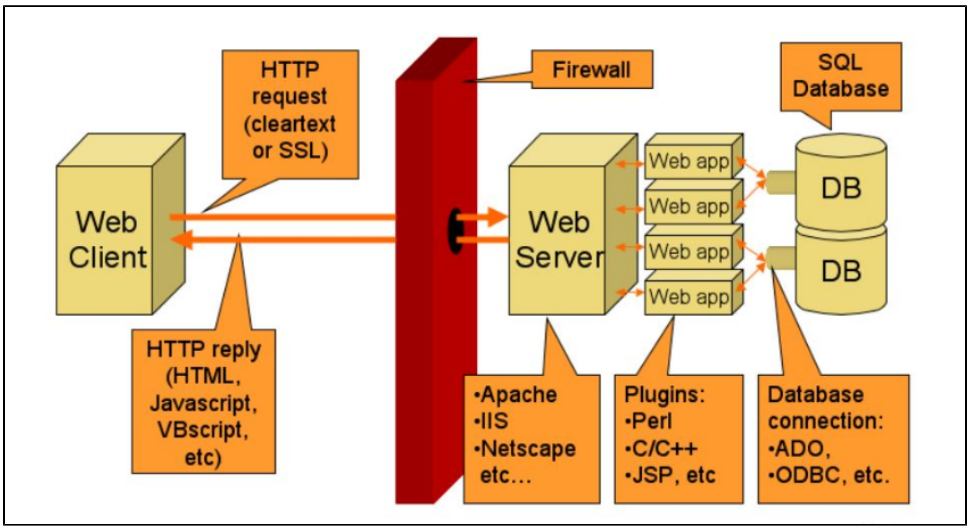
Module, Project Jigsaw, Flow

Local-Variable Type Inference

2. Spring 기본

(1) 들어가기에 앞서

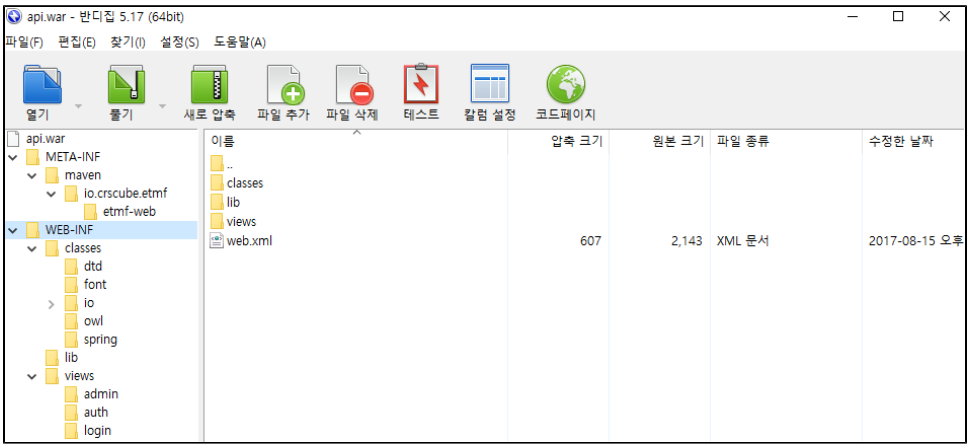
- Web Server와 WAS의 차이 (참고자료)



  - Web Server
    - 정적인 리소스
    - Apache, IIS, NginX
  - Web Application Server
    - Web Server + Web Container (Servlet Container)
    - 동적인 리소스
    - 비즈니스 로직 수행
    - JBOSS, Tomcat, Jeus, Jetty
  - 분리하는 이유?

- Web application

- Deployment Descriptor




```
<?xml version="1.0" encoding="UTF-8"?>
<web-app version="3.0" xmlns="http://java.sun.com/xml/ns/javaee" xmlns:xsi="http://www.w3.org/2001
/XMLSchema-instance"
    xsi:schemaLocation="http://java.sun.com/xml/ns/javaee http://java.sun.com/xml/ns/javaee/web-
app_3_0.xsd">

    <display-name>eTMF Web</display-name>
    <description>CRScube eTMF Web Service</description>

    <context-param>
        <param-name>spring.profiles.active</param-name>
        <param-value>dev</param-value>
    </context-param>

    <context-param>
        <param-name>contextConfigLocation</param-name>
        <param-value>
                /WEB-INF/config/easycompany-service.xml,/WEB-INF/config/easycompany-dao.
xml
        </param-value>
     </context-param>

    <servlet>
        <servlet-name>dispatcher</servlet-name>
        <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
        <init-param>
            <param-name>contextConfigLocation</param-name>
            <param-value>classpath:spring/context-etmf-web.xml</param-value>
        </init-param>
        <load-on-startup>1</load-on-startup>
    </servlet>
    <servlet-mapping>
        <servlet-name>dispatcher</servlet-name>
        <url-pattern>/</url-pattern>
    </servlet-mapping>

    <welcome-file-list>
        <welcome-file>/index.jsp</welcome-file>
    </welcome-file-list>

    <filter>
        <filter-name>CorsFilter</filter-name>
        <filter-class>io.crscube.etmf.web.filter.CorsFilter</filter-class>
    </filter>
    <filter-mapping>
        <filter-name>CorsFilter</filter-name>
        <url-pattern>/*</url-pattern>
    </filter-mapping>
```

(2) 3대 핵심기술

- IOC/DI (Inversion OfControl / Dependency Injection) - 제어의 역전 / 의존성 주입
    - Application 내의 컴포넌트 간 **낮은 결합도와 높은 응집도**를 지키게 하는 원리
    - Application을 구성하는 객체(Bean)의 생성과 소멸을 개발자가 아닌 Container가 관리함 (라이프사이클에 대한 제어의 역전)
    - Bean과 Bean 사이의 의존 관계를 Container가 처리 함 (의존성 주입을 통한 제어의 역전)

- AOP (Aspect Oriented Programming)
    - 관점 지향 프로그래밍
    - DI가 의존에 대한 주입이라면 AOP는 로직에 대한 주입
    - 모든 어플리케이션마다 공통적으로 필요한 기능(**횡단 관심사**)를 분리
        - 중복의 제거를 통해 중요 비즈니스 로직(**종단 관심사**)에 대해 더욱 관심을 가지게 함
        - 횡단 관심사 → Logging, 인증, 트랜잭션 처리, 비동기 처리
        - 종단 관심사 → 과제 생성, 과제 사용자 등록
- PSA (Portable Service Abstraction)
    - 환경의 변화와 관계 없이 일관된 방식으로 기술에 접근 할 수 있는 환경을 제공하려는 추상화 구조
      → Spring은 언어가 아닌 기술(구현체의 기술)에 얽매이는 것에 큰 반감을 가짐
      → Spring에서 동작 할 수 있는 Library들은 POJO 기반으로 구현되어 있음
    - 대표적인 추상화의 예
        - JPA의 구현체인 Hibernate, Eclipse Link를 추상화 하는 Spring Data JPA
        - Mybatis를 추상화하는 Spring-mybatis (Spring Data 시리즈가 아님을 유의)

※ POJO?

Plain Old Java Object의 약자로 순수하게 Getter, Setter로만 이뤄진 Value Object 성의 Bean을 의미하며 아래의 특징을 가진다.

- **특정 규약에 종속되지 않는다**
  → POJO는 자바 언어와 꼭 필요한 API 외에는 종속되지 않아야 한다.
  따라서 EJB와 같이 특정 규약을 따라 비즈니스 컴포넌트를 만들어야 하는 경우는 POJO가 아니다.
  특정 규약을 따라 만들게 하는 경우는 대부분 규약에서 제시하는 특정 클래스를 상속하도록 요구한다.
  대표적으로 Servlet을 개발하려면 HttpServlet을 상속해야 하는 사례이다.

- **특정 환경에 종속되지 않는다.**
  → 특히 비즈니스 로직을 담고 있는 POJO 클래스는 웹이라는 환경정보나 웹 기술을 담고 있는 클래스나 인터페이스를 사용해서는 안된다.
  대표적으로 비즈니스 로직을 담은 코드에 HttpServletRequest나 HttpSession과 관련된 API가 등장하는 경우 진정한 POJO라고 볼 수 없다