1 - SPRING INTRO (BA 19/06/2023)

Roadmap

- Spring Boot
- Hibernate (JPA)
- Spring Modules
- Web App
- Rest API
- MVC
- React
- Unit Test
- Integration Test
- Docker
- Cloud

Neden Spring?

Java programlamayı daha hızlı, daha basit ve herkes için daha güvenli yapar. Spring'in hız, basitlik ve verimlilik odaklı olması onu dünyanın en popüler Java frameworkü haline getirmiştir.

(*Spring is flexible*) At it's core, Spring Framework's ***Inversion of Control (IoC)*** and ***Dependency Injection (DI)*** features provide the foundation for a wide-ranging set of features and functionality.

Spring projects also support the ***reactive (nonblocking) programming model*** for even greater efficiency.

Tomcat Server

Server'a neden ihtiyacımız olur?

Web teknolojilerinde bir client, bir de server oluyordu. Client, Server'a istekte bulunuyor. Server içerisinde işletim sistemi, eğer bir web sunucusu olacaksa web sunucusu özelliğine sahip bir yazılım var. Spring'de Tomcat embedded olarak var. Bu yapı sayesinde bir sürü ayarla vs. ile uğraşmamış oluyoruz ve şaşırtıcı derecede hızlı bir yapı sunmuş oluyor. Normalde herhangi bir dilde, örneğin PHP'de .NET'te bir program yazarken mutlaka bir sunucu olması lazım, o sunucuya bunların yüklenmesi lazım, ayarların yapılması lazım. Ama burada Tomcat ayarlanmış bir şekilde bulunduğundan uygulama direkt ayağa kalkabiliyor.

API

API'ler, iki yazılım bileşiminin belirli tanımlar ve protokoller aracılığıyla birbirleriyle iletişim kurmasına olanak sağlayan mekanizmalardır. Örneğin, meteoroloji müdürlüğünün yazılım sistemi, günlük hava durumu verilerini içerir. Telefonlardaki hava durumu uygulaması, API'ler aracılığıyla bu sistemle konuşur ve telefonda günlük hava durumu güncellemelerini gösterir.

API Örnekleri

İş Bankası API Portal

Binance APIs

REST API

REST, Temsili Durum Aktarımı anlamına gelen Representational State Transfer ifadesinin kısaltmasıdır. REST, istemcilerin sunucu verilerine erişirken kullanabilmesi için GET, PUT, DELETE gibi belirli işlevler kullanır. İstemciler ve sunucular, HTTP üzerinden veri alışverişi yapar.

REST API'sinin ana özelliği durumsuz olmasıdır. Durumsuz olması, sunucuların istekler arasında istemci verilerini kaydetmemesi anlamına gelir. İstemcinin sunucuya gönderdiği istekler, bir web sitesini ziyaret etmek için tarayıcınıza yazdığınız URL'lere benzer. Gelen yanıt ise bir web sayfasında görmeye alışık olduğumuz grafikleri içermeyen sade verilerdir.

REST API'lerinin sunduğu dört temel avantaj vardır:

Entegrasyon

API'ler, yeni uygulamaların mevcut yazılım sistemlerine entegre edilmesi için kullanılır. Her bir işlevi sıfırdan yazmaya gerek duyulmadığından geliştirme hızı artar. Mevcut kodlardan yararlanmaya devam etmek için API'leri kullanabilirsiniz.

İnovasyon

Yeni bir uygulamanın gelişi, bütün bir sektörü değişime zorlayabilir. İşletmelerin hızlı yanıt vermesi ve yenilikçi hizmetlerin hızlı dağıtımını desteklemesi gerekir. Bunu, tüm kodu yeniden yazmak yerine API düzeyinde değişiklikler yaparak gerçekleştirebilirler.

Genişleme

API'ler, müşterilerinin ihtiyaçlarını birden fazla platformda karşılamak isteyen işletmelere benzersiz bir fırsat sunar. Örneğin, haritalar API'si harita bilgileri entegrasyonunun web siteleri, Android, iOS vb. aracılığıyla yapılabilmesini sağlar. Her işletme, ücretsiz veya ücretli API'leri kullanarak kendi dahili veritabanlarına benzer şekilde erişim verebilir.

Bakım kolaylığı

API, iki sistem arasında bir ağ geçidi vazifesi görür. Her bir sistem, API'nin etkilenmemesi için dahili değişiklikler yapma gereği duyar. Bu sayede, taraflardan birinin gelecekte yapacağı kod değişiklikleri, diğer tarafı etkilemez.

EXAMPLE PROJECT

Spring Initializr

https://start.spring.io/:

- Project: Gradle-Groovy
- Language: Java
- Spring Boot: Last one (2.7.14)
- Project Metadata
    - Artifact: Project Name
    - Package name: org.hulyam

Dependencies

- Lombok
- Spring Boot Dev Tools
- Spring Web
- Spring Data JPA
- PostgreSQL Driver

application.yml & application.properties

Proje ile alakalı genel ayarların tutulduğu dosyalardır.

.properties veya YAML (.yml) dosyası olabilir. YAML dosyası olması yazımı kolaylaştırır. Daha sonra cloudla çalışırken yaml üzerinden gideceğimiz için şimdiden yaml dosyası oluşturuyoruz.

Öncelikle parametre ardından iki nokta üst üste ve **bir boşluk** bırakıldıktan sonra değer girilik. ***Eğer boşluk bırakılmazsa app hata verecektir.***

Alt kırılımlarda da **bir TAB** boşluk olmalıdır.

application.yml:

```yaml
server:
  port: 9090

spring:
  datasource:
    driver-class-name: org.postgresql.Driver
    username: postgres
    password: root
    url: jdbc:postgresql://localhost:5432/DbSpringIntro
  jpa:
    hibernate:
      ddl-auto: create-drop
```

application.properties:

```properties
server.port=9090

spring.datasource.driver-class-name=org.postgresql.Driver
spring.datasource.username=postgres
spring.datasource.password=root
spring.datasource.url=jdbc:postgresql://localhost:5432/DbSpringIntro

spring.jpa.hibernate.ddl-auto=create-drop
```

Main Layers and Annotations

> ***NOT:*** *Projeyi https://start.spring.io/ üzerinden oluşturduğumuz için Main (SpringExercies1IntroApplication) class'a @SpringBootApplication anotasyonunu ve "SpringApplication.run..." kodunu otomatik olarak ekliyor.*
>
> ***Main****:*

```
package org.hulyam;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication
public class SpringExercies1IntroApplication {

 public static void main(String[] args) {
     SpringApplication.run(SpringExercies1IntroApplication.class,
args);
 }

}
```

Repository

Entity

Burada her zaman eklediğimiz Lombok Entity anotasyonlarını "User" örnek clasına ekliyoruz.

User:

```
package org.hulyam.repository.entity;

import lombok.AllArgsConstructor;
import lombok.Builder;
import lombok.Data;
import lombok.NoArgsConstructor;

import javax.persistence.*;

@Builder
@Data
@NoArgsConstructor
@AllArgsConstructor
@Entity
@Table(name = "tbluser")
public class User {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    Long id;
    String name;
```

```
    String address;
}
```

IUserRepository Interface

- **Spring Annotation:** @Repository
- extends JpaRepository<User, Long>

```
package org.hulyam.repository;

import org.hulyam.repository.entity.User;
import org.springframework.data.jpa.repository.JpaRepository;
import org.springframework.stereotype.Repository;

@Repository
public interface IUserRepository extends JpaRepository<User, Long> {
}
```

Service

- **Spring Annotation:** @Service
- **Lombok Annotation:** @RequiredArgsConstructor

```
package org.hulyam.service;

import lombok.RequiredArgsConstructor;
import org.hulyam.repository.IUserRepository;
import org.hulyam.repository.entity.User;
import org.springframework.stereotype.Service;

import java.util.List;

@Service
@RequiredArgsConstructor
public class UserService {
    private final IUserRepository userRepository;

    public User save(User user) {
        return userRepository.save(user);
    }

    public List<User> findAll() {
        return userRepository.findAll();
    }
```

```
}
```

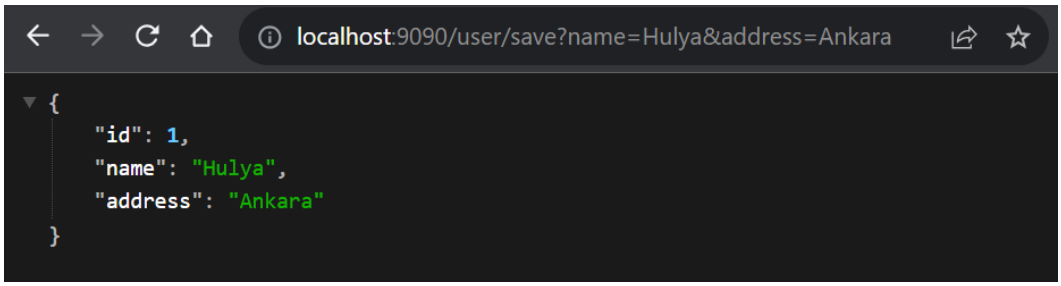
Controller

- **Spring Annotation:** @RestController
- **Spring Annotation:** @RequestMapping("/user")
- **Lombok Annotation:** @RequiredArgsConstructor

```
package org.hulyam.controller;

import lombok.RequiredArgsConstructor;
import org.hulyam.repository.entity.User;
import org.hulyam.service.UserService;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;

import java.util.List;

@RestController
@RequestMapping("/user")
@RequiredArgsConstructor
public class UserController {
    private final UserService userService;



    @GetMapping("/save")
    public ResponseEntity<User> save(String name, String address) {
        return ResponseEntity.ok(userService.save(
                User.builder().name(name).address(address).build()
        ));
    }

    @GetMapping("/findall")
    public ResponseEntity<List<User>> findAll() {
        return ResponseEntity.ok(userService.findAll());
    }
}
```

> ***@GetMapping(/save)*** *UserController'da yazdığımız save metodunun "//localhost:9090/user/save" bağlantısı ile çağırılabileceğini belirtir.*

> ***Metoda bağlantı üzerinden parametre gönderme***
>
> *//localhost:9090/user/save?name=Hulya&address=Ankara*
>
> *Yukarıdaki bağlantıya gidildiğinde Controller'daki save metodundan dönen JSON objesi:*



ResponseEntity

Controller'da returnler artık ResponseEntity olarak girilir. Kullanıcıya dönülecek olan datanın Entity olarak dönmesini ve bunun JSON formatında dönmesini sağlamak için kullanılır.

ResponseEntity sayesinde Controller'daki metotlar geriye JSON objesi dönerler. Kullanılma sebebi de budur.

Neden JSON?

JSON alternatifi XML veri dönüşüdür. XML eskiden daha çok kullanılıyordu ve dosya boyutu JSON'a göre daha fazladır. Bu nedenle JSON artık neredeyse standarta dönüşmüş durumdadır.

Mapping Annotations

- @GetMapping
- @PostMapping
- @DeleteMapping
- @RequestMapping
- @PatchMapping
- @PutMapping

Gelen isteğin türüne göre uygun işaretlenmiş metotlarla yönlendirme gerçekleştirirler.

> *Tarayıcı üzerinden GET request dışında bir istekte bulunamayız. Deneysel olarak requestlerin incelenmesi için **Postman** ve **Swagger** gibi uygulamalar kullanılabilir.*

Request Types:

- GET
- POST
- PUT
- PATCH
- DELETE
- HEAD

2 - SPRING - COUPLING / IoC

20.06.23

When we talk about coupling, we describe the degree to which classes within our system depend on each other. Our goal during the development process is to reduce coupling.

Tight Coupling

When a group of classes is highly dependent on each other, or we've classes that assume a lot of responsibilities is called tight coupling.

Loose Coupling

Loose coupling is when an object gets the object to be used from external sources (e.g. Calculator class uses IShape interface instead of Triangle or Square classes)

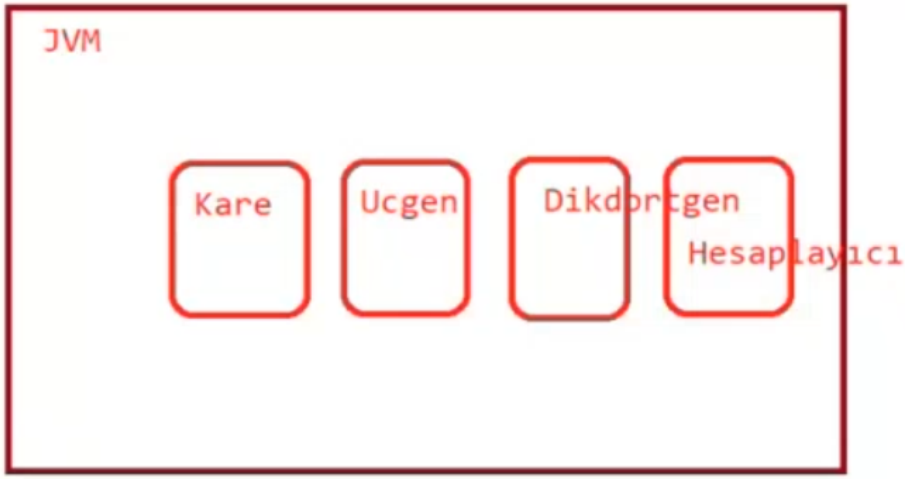
Dependency Injection (DI)

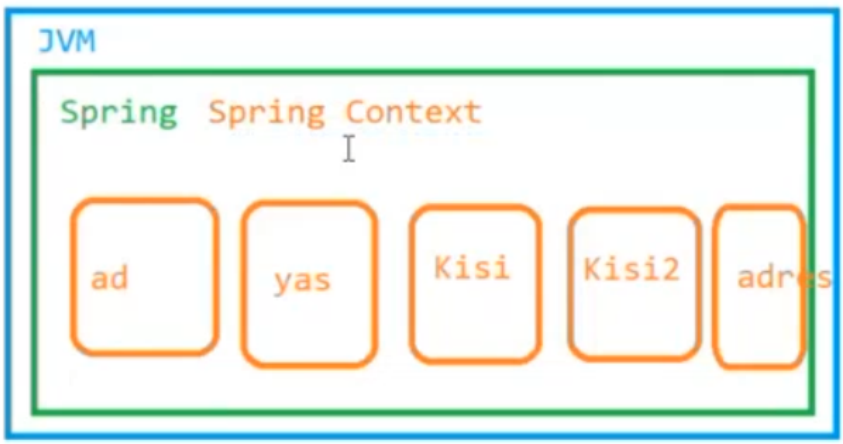
In software engineering, dependency injection is **a programming technique in which an object or function receives other objects or functions that it depends on**.

```
public class Main {
    public static void main(String[] args) {
        var shape = new Triangle(); // Creating object
        var calculator = new Calculator(shape); // Creating object +
Wiring of Dependencies
        // Here, shape is a dependency for calculator.
        // shape object is injected in calculator object. ==> DI:
Dependency Injection
        calculator.calculate();
    }
}
```

Inversion of Control (IoC)

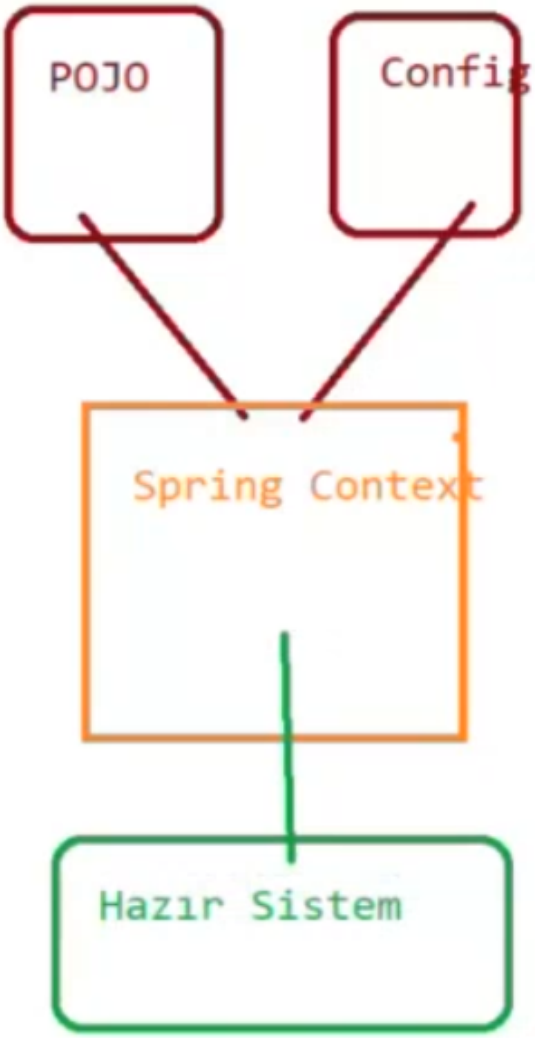
Inversion of Control is a principle in software engineering which transfers the control of objects or portions of a program to a container or framework.





Spring Context - Spring Container - IoC Container

Hepsi aynı kavramı tanımlayan kavramlardır. Amaçları Spring beanleri ve onların yaşam döngülerini yönetmektir:



IoC and DI Example

Sınıfları, özellikleri, nesneleri (Spring Beans olarak geçer bunlar. Yukarıdaki resimde turuncu ile gösterilenler) yaratma ve birbirine bağlama işlerini Spring'in devralabilmesi için Spring Context (Spring Container veya IoC Container) oluşturulur.

Spring Bean: Spring tarafından yönetilen Java objeleridir. ***@Bean*** anotasyonu ile oluşturulur.

Java Bean: Bir sınıfın Java Bean olabilmesi için kesinlikle sahip olması gereken 3 özellik vardır:

1. Public boş constructor

2. Getters & Setters
3. Serializable interface'ten implement alması

> ***ejb:*** *Enterprise Java Bean*
>
> ***Neden kullanılıyordu?***

Pojo: Pojo'da ise hiçbir şey olma zorunluluğu yoktur. İçinde constructor, getter&setter vs. olabilir de olmayabilir de. Herhangi bir kısıtlaması olmayan Java sınıflarıdır.

build.gradle

Spring'in projemizde çalışabilmesi için öncelikle gerekli dependencies https://mvnrepository.com/'dan eklenmelidir.

- Spring Core (Versiyon 5'ten sonuncusunu seçtik)
- Spring Context (Yine versiyon 5'ten sonuncusu) - @Configuration anotasyonunu kullanabilmek ve SpringConfiguration classında Spring için ayarları belirtebilmek için ekliyoruz.

```
plugins {
    id 'java'
}

group = 'org.hulyam'
version = '1.0-SNAPSHOT'

repositories {
    mavenCentral()
}

dependencies {
    //
https://mvnrepository.com/artifact/org.springframework/spring-core
    implementation 'org.springframework:spring-core:5.3.29'
    //
https://mvnrepository.com/artifact/org.springframework/spring-
context
    implementation 'org.springframework:spring-context:5.3.29'

}
```

Spring Context and SpringConfigutation

SpringConfiguration:

Spring'in ayar dosyasıdır. ***@Configuration*** anotasyonu ile belirlenir. Spring tarafından yönetilmesini istediğimiz beanleri burada belirtiriz.

```
package org.hulyam;

import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;

@Configuration
public class SpringConfiguration {

    // @Bean anotasyonu ile Spring Context tarafından yönetilecek
bir bean oluşturuluyor.
    @Bean
    public String name() {
        return "Mr. Bean";
    }

    @Bean
    public int age(){
        return 50;
    }

}
```

> ***record:*** *Java 14 ile beraber gelmiş bir özelliktir. Yeni bir class oluşturmayı sağlar.*
>
> ```
> record Person(String name, int age);
> ```
>
> *İçerisinde name ve age özelliklerini barındıran bir sınıf oluşturmuş oluyoruz.* ***Getter, setter ve constructorı kendisi oluşturur****.*

SpringConfiguration.class:

```
package org.hulyam;

import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;

record Person(String name, int age){};
record Address(String city, String country){};
```

```
@Configuration
public class SpringConfiguration {

    // @Bean anotasyonu ile Spring Context tarafından yönetilecek
bir bean oluşturuluyor.
    @Bean
    public String name() {
        return "Mr. Bean";
    }

    @Bean
    public int age() {
        return 50;
    }

    @Bean
    public Person person(){
        return new Person("Hulya",32);
    }

    @Bean("addr.")
    public Address address(){
        return new Address("Ankara","Turkiye");
    }
}
```

Main:

```
package org.hulyam;

import
org.springframework.context.annotation.AnnotationConfigApplicationCo
ntext;

public class Main {
    public static void main(String[] args) {
        // SpringConfiguration dosyasında oluşturduğumuz beane
ulaşmaya çalışıyoruz.
        // Spring Context'i oluşturmak için kullanılır, içine class
olarak ayar dosyası verilir:
        var context = new
// BEAN CALLS
// With Method Names
```

```
AnnotationConfigApplicationContext(SpringConfiguration.class);
        System.out.println(context.getBean("name")); //same as the
method name in SpringConfiguration.class
        System.out.println(context.getBean("age"));
        System.out.println(context.getBean("person"));
        System.out.println(context.getBean("addr."));
    }
}
```

> ***Application Context:*** *Spring Context/Spring Container/IoC Container'a ulaşmanın bir yoludur. Başka yollar da vardır ama en çok bu kullanılır.*
>
> *NEDENİ:*
>
> - *Kurumsal yapılara daha uygun*
> - *Web uygulamaları için de uygun*
> - *Aspect Oriented Programming (AOP) ile de uyumludur.*

Bean içinde bean çağırma

SpringConfiguration:

```
    @Bean
    public String name() {
        return "Mr. Bean";
    }

    @Bean
    public int age() {
        return 50;
    }

    @Bean
    public Person person2(){
        return new Person(name(),age());
    }
```

```
record Person2(String name, int yas, Address address){};

@Configuration
public class SpringConfiguration {

    @Bean
    public String name() {
```

```
        return "Mr. Bean";
    }

    @Bean
    public int age() {
        return 50;
    }

    @Bean("addr.")
    public Address address(){
        return new Address("Los Angeles","USA");
    }

    @Bean
    public Person2 person2(){
        return new Person2("Nihal",40,new
Address("Istanbul","Turkiye"));
    }

    @Bean
    public Person2 person2MethodCalling(){
        return new Person2(name(),age(),address());
    }
}
```

Beanlerin Method Calling İçin Önceliklendirilmesi

@Primary

Method Calling yaparken Main'de "contect.getBean(Address.class)" kod parçacığı ile class name kullanılarak çağrı yapılırsa, iki veya daha fazla aynı sınıftan dönüş yapan bean varsa, bu "expected single matching bean but found 2" hatasına neden olacaktır. Bunu gidermek ve hangisinin öncelikli olarak çağırılacağını belirtmek için @Primary anotasyonu kullanılır.

SpringConfiguration:

```
    @Bean("addr.")
    public Address address(){
        return new Address("Los Angeles","USA");
    }

    @Bean
    @Primary
    public Address address2(){
        return new Address("Ankara","Turkiye");
    }
}
```

Main:

```
package org.hulyam;

import
org.springframework.context.annotation.AnnotationConfigApplicationCo
ntext;

public class Main {
    public static void main(String[] args) {
        var context = new
AnnotationConfigApplicationContext(SpringConfiguration.class);
        // BEAN CALLS
        // With Class Names
        System.out.println(context.getBean(Address.class));
    }
}
```

@Qualifier

SpringConfiguration:

```
package org.hulyam;

import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.context.annotation.Primary;

record Address(String city, String country){};

record Person2(String name, int yas, Address address){};
@Configuration
```

```
public class SpringConfiguration {

    @Bean
    public String name() {
        return "Mr. Bean";
    }

    @Bean
    public int age() {
        return 50;
    }

    @Bean("addr.")
    public Address address(){
        return new Address("Los Angeles","USA");
    }

    @Bean
    public Person2 person2(){
        return new Person2("Nihal",40,new
Address("Istanbul","Turkiye"));
    }

    @Bean
    @Primary
    public Address address2(){
        return new Address("Ankara","Turkiye");
    }

    @Bean
    public Person2 person2Qualifier(String name, int age, Address
address){
        return new Person2(name,age,address);
    }
}
```

Burada yeni Person2 oluştururken parametre olarak metod isimlerini girdiğimiz için ve address2 metodunda @Primary anotasyonu olduğu için Main sınıfında aşağıdaki bean call yapıldığında:

Main:

```
package org.hulyam;

import
org.springframework.context.annotation.AnnotationConfigApplicationCo
ntext;

public class Main {
    public static void main(String[] args) {
        var context = new
AnnotationConfigApplicationContext(SpringConfiguration.class);
        // BEAN CALLS
        // With Method Names
        System.out.println(context.getBean("person2Qualifier"));
```

sonuç aşağıdaki şekilde olur ve name, age metodlarından gelen değerler ile @Primary anotasyonunun kullanıldığı adres ile Person2 oluşturulur:

ÇIKTI:

```
Person2[name=Mr. Bean, yas=50, address=Address[city=Ankara,
country=Turkiye]]
```

Ama biz primary adresi değil de başka bir adres alarak bir Person2 yaratmak istiyorsak burada @Qualifier anotasyonunu kullanırız:

SpringConfiguration:

```
@Bean
    @Primary
    public Address address2(){
        return new Address("Ankara","Turkiye");
    }

    @Bean
    @Qualifier("romeQualifier")
    public Address address3(){
        return new Address("Rome","Italy");
    }

    @Bean
    public Person2 person2Qualifier(String name, int age,
@Qualifier("romeQualifier") Address address){
        return new Person2(name,age,address);
    }
```

Spring Framework Tarafından Yönetilen Tüm Beanler

```
public class Main {
    public static void main(String[] args) {
        var context = new
AnnotationConfigApplicationContext(SpringConfiguration.class);

        // All Beans Controlled by Spring Framework
Arrays.stream(context.getBeanDefinitionNames()).forEach(System.out::
println);

    }
}
```

3 - MONOLITHIC ARCHITECTURE

Layers/Packages List

- constant
- controller
- dto (Data Transfer Object)
    - request
    - response
- exception (Exception Management - Aspect Oriented Programming example on this project)
- mapper (Used for the conversion between entities and dtos)
- repository
    - entity
- service
- utility

Monolitik Mimari: Tek bir yapı kurulur. Bunun çıktısı tek bir jar dosyası / tek bir uygulama olacaktır. Bunun alternatifi Microservice Mimarisidir. Microservice bu yapılardan birçoğunu içerir.

Spring Data JPA Query Methods

Controller:

```
@RestController
@RequestMapping("/user")
@RequiredArgsConstructor
public class UserController {
    private final UserService userService;

    // http://localhost:9090/user/findbyname?name=Hulya
    @GetMapping("/findbyname")
    public ResponseEntity<User> findByName(String name) {
        return ResponseEntity.ok(userService.findByName(name)
        );
    }

    @GetMapping("/findbynameandaddress")
    public ResponseEntity<User> findByNameAndAddress(String name,
String address){
        return
ResponseEntity.ok(userService.findByNameAndAddress(name,address));
    }

    @GetMapping("/findbynameoraddress")
    public ResponseEntity<List<User>> findByNameOrAddress(String
name, String address){
        return
ResponseEntity.ok(userService.findByNameOrAddress(name,address));
    }
}
```

Service:

```
@Service
@RequiredArgsConstructor
public class UserService {
    private final IUserRepository userRepository;
```

```
    public User findByName(String name) {
        return userRepository.findByName(name);
    }

    public User findByNameAndAddress(String name, String address) {
        return userRepository.findByNameAndAddress(name,address);
    }

    public List<User> findByNameOrAddress(String name, String
address){
        return userRepository.findByNameOrAddress(name,address);
    }
}
```

Repository:

```
@Repository
public interface IUserRepository extends JpaRepository<User, Long> {
    // select * from tbluser where name=?
    User findByName(String name);

    User findByAddress(String address);

    // select * from tbluser where name=? and address=?
    User findByNameAndAddress(String name, String address);

    // select * from tbluser where name=? or address=?
    List<User> findByNameOrAddress(String name, String address);

    User findByNameAndAddressAndTel(String name, String address,
String tel);

    List<User> findAllByCity(String city);

    List<User> findAllByNameStartsWith(String word);

    Optional<User> findOptionalByNameAndTel(String name, String
tel);

    List<User> findAllByNameOrderByNameDesc(String name);

    // SQL - limit 3
    List<User> findTop3ByNameOrderByName(String name);
```

Repository Interface - Spring Data JPA Query Methods Writing Rules

find: aramalar için find kelimesi ile metoda başlanması gerekir.

By: B büyük, y küçük yazılmak zorundadır. İşlem yapılacak bileşen(değişken) adından önce yazılması gerekir.

[değişken adı]/(name): Arama yapılmak istenen field adı, yine büyük ile başlayıp küçük harfle devam etmeli. Bu şekilde User entitysi içinde "name" parametresi için arama yapar. Hangi repositoryde ise ona göre değişken araması yapar.

* Metotta geri dönüş tipi olmalıdır.

* Birden fazla değişken ile arama yapılıyorsa, parametre giriş sırası mutlaka metot adındaki sıraya göre yazılmalıdır: findBy**Name**orAddress(**String name**, String address)

Package Details

Utility

IService Interface - ServiceManager

Kod tekrarından kaçınmak ve her entity için aynı metotları yazmakla uğraşmamak için utilty package içerisine Service Interface eklenmeli ve entity servisleri bu interfaceden implement edilmeli.

IService:

```
package org.hulyam.utility;


import java.util.Optional;


public interface IService<T,ID> {
    T save(T t);
    Iterable<T> saveAll(Iterable<T> t);
    T update(T t);
    void delete(T t);
    void deleteById(ID id);
    Optional<T> findById(ID id);
    Iterable<T> findAll();
}
```

Bu tekrar eden metodlar ile yapılacak standart işlemleri belirlemek için de ServiceManager sınıfı eklenir:

ServiceManager:

```
package org.hulyam.utility;

import org.springframework.data.jpa.repository.JpaRepository;

import java.util.Optional;

public class ServiceManager<T,ID> implements IService<T,ID>{
    private final JpaRepository<T,ID> jpaRepository;

    public ServiceManager(JpaRepository<T, ID> jpaRepository) {
        this.jpaRepository = jpaRepository;
    }

    @Override
    public T save(T t) {
        return jpaRepository.save(t);
    }

    @Override
    public Iterable<T> saveAll(Iterable<T> t) {
        return jpaRepository.saveAll(t);
    }

    @Override
    public T update(T t) {
        return jpaRepository.save(t);
    }

    @Override
    public void delete(T t) {
        jpaRepository.delete(t);
    }

    @Override
    public void deleteById(ID id) {
        jpaRepository.deleteById(id);
    }

    @Override
```

```
    public Optional<T> findById(ID id) {
        return jpaRepository.findById(id);
    }

    @Override
    public Iterable<T> findAll() {
        return jpaRepository.findAll();
    }
}
```

Artık Entity servislerimiz ServiceManager sınıfından extend alacak. Bu durumda super içeren constructor oluşturulması gerekeceğinden, IUserRepository'i parametre olarak alan constructor oluşturulur. Yukarıda proje başlangıcında ilk oluşturduğumuz UserService sınıfındaki Lombok @RequiredArgsConstructor kullanımına gerek kalmaz.

UserService:

```
package org.hulyam.service;

import org.hulyam.repository.IUserRepository;
import org.hulyam.repository.entity.User;
import org.hulyam.utility.ServiceManager;
import org.springframework.stereotype.Service;

import java.util.List;

@Service
public class UserService extends ServiceManager<User,Long> {
    private final IUserRepository userRepository;

    public UserService(IUserRepository userRepository) {
        super(userRepository);
        this.userRepository = userRepository;
    }
}
```

APPENDIX A - LINKS

Name	Link
Spring	https://spring.io/
Spring Initializr	https://start.spring.io/
Spring Training and Certification (VMware)	https://spring.academy/paths/spring-certified-professional-2023
Postman (REST API Test App)	https://www.postman.com/
Swagger (REST API Documentation App)	https://swagger.io/

APPENDIX B - SPRING ANNOTATIONS

Spring, anotasyonlar ile sistemi yönetir. Nesne yaratmak için belli anotasyonlar kullanır.

Annotation	Description
@SpringBootApplication	Spring'in diğer anotasyonları fark edip projeyi ona före çalıştırmasını sağlayan asıl anotasyondur. Bu anotasyon sayesinde içinde bulunduğu package ve onun altındaki packageların içindeki tüm dosyalarda diğer anotasyonları arar ve bulduklarıyla gerekli işlemleri yapar.
@Repository	
@Service	

Annotation	Description
@RestController	MVC ile çalışırken "@Controller" anotasyonu kullanılır. REST API projesinde ise bu anotasyon kullanılır. REST API'de "findall" gibi bir istek atıldığında geriye data döner. REST API'de yapı data transferi üzerine kuruludur. Bu nedenle çok daha hızlıdır. Bu nedenle bir çok kuruluş arka planda REST API kullanmakta veya bu yapıya geçmektedir.
@RequestMapping	Uygulamaya gelen isteklerin URL içinden analiz edilip ulaşması gereken sınıfa yönlendirilmesi için kullanılır. Bir istek geldiği zaman bu sınıf hangi isteği karşılayacak bunu belirttiğimiz anotasyondur. Aslında arka planda *filtreleme* işi yapar.
@GetMapping	GET isteğinde bulunulursa karşılayacak metot işaretlenir.