# Common Module:

Projeden bağmsız bir pom’a sahip olmalı. Parent vs bilgisi ona göre kurgulanmalıdır.

* **Common Module’den entityl’ler nasıl scan edilir?**

İlgili service common module’den kendi entity’lerini çekmek için @EntityScan anotasyonunu kullanır.(@EntityScan({“common.module.package”}))

* **Common Module’den componentler nasıl scan edilir?**

İlgili service common module’den componentleri çekmek için @ComponentScan anotasyonunu kullanır.(@ComponentScan ({“common.module.package”}))

Auth filter gibi bir dependency’i eklenmişse o dependency’nin tüm package’ları @ComponentScan ile scan edilmelidir aksi halde dependency çalışmayacaktırç

* **@ControllerAdvice** -> Global exception handling yapmaya yarayan bir anotasyondur.

Fırlatılan hata client’a gitmeden @ControllerAdvice altından tanımlanan method ile yakalanır ve client’a response döner. Common module’de tanımlanmış @ControllerAdvice sınıfları @Import(commonModuleClass.class, …) ile kullanılır.

Örneğin:

@ControllerAdvice  
@Import({CustomCommonExceptionHandler.class, validationHandler.class})  
public class CustomizeExceptionHandling {  
 @ExceptionHandler(UserSameInformationException.class)  
 public ResponseEntity<Object> UserSameInformationException(UserSameInformationException exception, WebRequest webRequest) {  
 return (ResponseEntity<Object>) CommonVariables.ResponseHandler(ExceptionMessage.USER\_SAME\_INFORMATION\_EXCEPTION, HttpStatus.BAD\_REQUEST, new ArrayList<>());  
 }  
}

* Common module’de tanımlanmış FeignClientlar @EnableFeignClients({common.module.package}) ile ilgili service’te kullanıma hazır hale getirilir.
* **@FeignClient nedir?** -> Feign client micro-service mimarisinde serviclerin kendi içlerinde birbirlerinin kopyasını oluşturmasıdır. Eureka ile entegredir. Euraka service name’i ile direk işlem yapılabilir.

Örneğin:

@FeignClient(value = Constants.*AuthServerEurekaName*, path = Constants.*UserServiceFeignClientRequestMapping*)  
public interface AuthServerFeignClient {  
  
 @GetMapping(Constants.*GetEntity*+"{id}")  
 CommonUserEntity getEntity(@RequestHeader(Constants.*AuthorizationHeader*) String token, @PathVariable("id") Long id);  
}

# Admin Server:

Admin server service’lerin statu’lerini ve süreçlerini takip etmemizi sağlayan bir platformdur.

<dependency>  
 <groupId>de.codecentric</groupId>  
 <artifactId>spring-boot-admin-starter-server</artifactId>  
 <version>2.3.1</version>  
</dependency>

İle projeye eklenir.

@SpringBootApplication  
@EnableDiscoveryClient  
@EnableAdminServer

Anotasyonları main class’a eklenir.

Eureka ile senktron çalışabilmektedir. Euraka’da register olmuş service’leri actuator ile health check yaparak kontrol edebilmektedir. Bu şekilde service’ler bir UI’dan kontrol edilebilir.

# Spring Security Auth Filter:

Auth işlemlerini yönetmemizi sağlar.

<dependency>  
 <groupId>org.springframework.boot</groupId>  
 <artifactId>spring-boot-starter-security</artifactId>  
 <version>2.7.3</version>  
</dependency>

İle projeye eklenir.

Conf. İçin ilgili class’a @EnableWebSecurity anotasoynu eklenir ve WebSecurityConfigurarerAdapter extend edilir.

@Override  
protected void configure(HttpSecurity http) throws Exception {  
 http.csrf().disable();  
 http.authorizeRequests().antMatchers(((List<String>) (context.getBean("permitURL"))).toArray(new String[0])).permitAll();//Auth. gerekmeden erişilebilen url'ler.AdminServer'için actuator serbest bırakıldı.  
 //http.authorizeRequests().antMatchers("/userService/\*\*").permitAll();//Auth. gerekmeden erişilebilen url'ler.  
 //http.authorizeRequests().antMatchers("/bike/\*\*").access("hasRole('ROLE\_admin')"); //admin rolune sahip birinin bu url'e erişebileceğini belirttik.  
 //http.authorizeRequests().antMatchers("/car/\*\*").access("hasRole('ROLE\_client')"); //admin rolune sahip birinin bu url'e erişebileceğini belirttik. http.authorizeRequests().anyRequest().authenticated();//Tüm url'ler için authenticationa ihtiyaç duyulmasını sağlar.  
 http.authorizeRequests().anyRequest().authenticated();  
 http.sessionManagement();  
 http.sessionManagement().sessionCreationPolicy(SessionCreationPolicy.*STATELESS*);//Session oluşturmayı ortadan kaldırıyoruz. Bunu JWT ile sağlayacağız.  
 http.addFilterBefore(jwtFilter, UsernamePasswordAuthenticationFilter.class);  
}

# Config Server:

Service’lerin config. File’larını yönetmeye yarayan bir yapıdır. Git entegrasyonu ile servicel’lerin file’ları git üzerinde tutulur ve service’ler ayağa kalkarken bu fileları çekip ayağa kalkar.

Main class’a bu anotasyonlar eklenmelidir.

@SpringBootApplication  
@EnableConfigServer

Server için:

<dependency>  
 <groupId>org.springframework.cloud</groupId>  
 <artifactId>spring-cloud-config-server</artifactId>  
</dependency>

spring:  
 application:  
 name: CONFIG-SERVER  
 security:  
 user:  
 name: ${CONFIG\_SERVER\_USERNAME}  
 password: ${CONFIG\_SERVER\_PASSWORD}  
 cloud:  
 config:  
 server:  
 git:  
 uri: ${GIT\_URL}  
 clone-on-start: true  
 username: ${GIT\_USER}  
 password: ${GIT\_PASSWORD}  
 search-paths: ${GIT\_DIRECTORY}  
 encrypt:  
 enabled: false

Client için:

<dependency>  
 <groupId>org.springframework.cloud</groupId>  
 <artifactId>spring-cloud-starter-config</artifactId>  
</dependency>

<dependency>  
 <groupId>org.springframework.retry</groupId>  
 <artifactId>spring-retry</artifactId>  
</dependency>

spring:  
 application:  
 name: commentService  
 cloud:  
 config:  
 profile: dev  
 uri: ${CONFIG\_SERVER}  
 username: ${CONFIG\_SERVER\_USERNAME}  
 password: ${CONFIG\_SERVER\_PASSWORD}  
 failFast: true  
 retry:  
 initialInterval: 3000 *#baslangic re-connection periyodu* multiplier: 1.3 *#her fail connection'dan sonra bir önce ki connection periyodunu 1.3 ile carpip okadar bekler* maxInterval: 10000 *#1.3 ile carpilan degerin sonucu 10000'den büyükse 10000 olarak kabul et* maxAttempts: 100 *#connection deneme sayisi*

Kullanılır. Retry ile re-connection süreçleri kontrol edilebilmektedir.

Eureka ile senkron çalışır. Eureka service name’i ile aynı yml file’i çeker. Profile göre farklı yml dosyalarını yollabilir. Profile ile yapılan tanımya göre yml dosyası değişecektir.

Örneğin:

1) myConfig-dev.yml = dev ortamının env bilgilerini tutar.  
2) myConfig-test.ylm = test ortamının env bilgilerini tutar.

# Eureka Server:

Micro service’lerin register olmasını, birbirleri arasında haberleşme sağlamasını, load balancing’i sağlayan bir yapıdır. Micro service mimarisinde spring ekosistemi düşünülünce en önemli management service’lerden biridir. Main class’a bu anostasyonlar eklenmelidir.

@SpringBootApplication  
@EnableEurekaServer

Server için:

<dependency>  
 <groupId>org.springframework.cloud</groupId>  
 <artifactId>spring-cloud-starter-netflix-eureka-server</artifactId>  
</dependency>

server:

port: 8001

spring:

application:

name: EUREKA-SERVER

security:

user:

name: grkem

password: 1234

zipkin:

base-url: ${zipkinServerURL}

eureka:

instance:

hostname: localhost

preferIpAddress: **true**

healthCheckUrl: http://${eureka.instance.hostname}:${server.port}/actuator/health

homePageUrl: http://${eureka.instance.hostname}:${server.port}

server:

getWaitTimeInMsWhenSyncEmpty: 2

client:

registerWithEureka: **true**

fetchRegistry: **false**

serviceUrl:

defaultZone: http://grkem:1234@myDomain:8011/eureka

Client İçin:

<dependency>  
 <groupId>org.springframework.cloud</groupId>  
 <artifactId>spring-cloud-starter-netflix-eureka-client</artifactId>  
</dependency>

server:

port: 8602

spring:

application:

name: COMMENT-SERVICE

eureka:

client:

registerWithEureka: **true**

fetchRegistry: **true**

serviceUrl:

defaultZone: http://${eurakaUsername}:${eurakaPassword}@${eurakaIP}

healthcheck:

enabled: **true**

şeklimde oluşturulur.

username-password ile register olunmalıdır.  
http://{Username}:{Password}@{Ip}:{Port}/eureka/

registerWithEureka = true -> 30sn'de bir server'e kalp atışı yollar  
leaseRenewalIntervalInSeconds -> Kalpt atışı hızını değiştirmek için kullanılır  
leaseExpirationDurationInSeconds -> Kalp atışı gelmezse setlenen süre kadar daha bekler ve kayıdı registry'den siler  
fetchRegistry -> Eureka'ye register olan client, aktif register olan tüm client'ların listesini çeker  
registryFetchIntervalSeconds -> Server'dan aktif client listesnin çekme periyodu.  
getWaitTimeInMsWhenSyncEmpty -> 5 defa hata alan client'i setlenen dakika boyunca kabul etmez  
maxThreadsForPeerReplication -> Replication'a yollanacak verileri için oluşturulan thread sayısı.  
eureka.client.enabled -> Client'i devre dışı braıkmayı sağlar.  
  
eureka.client.enabled: false ile eureka.client.registerWithEureka: false'un farkı registerWithEureka:false her işlemi yapabilir ancak eureka'ya kayıt olmaz.  
Örneğin fetchRegistry:true ve registerWithEureka:false olursa eureka'ya kayıt olamz ancak kayıtlı olan registry'leri çekebilir.  
eureka.client.enabled: false ise komple eureka'yı o client için iptal eder ve hiçbir işlem yapmaz.  
  
Production ortamında genelde 2 eureke sunucu ayağa kaldırılır ve registerWithEureka: true olarak setlenir.  
Bu durum olası ağ hareketlerini azaltmaya yöneliktir.

# Gateway Server:

Sunucuda public olacak tek url’dir. Gelen isteklerini istenilen service’lere route eder. Eureka ile senkron olabilir. Eureka service insance name’e göre route işlemi yapabilmektedir. Fallback entegrasyonu sayesinde bozuluklar istenilen şekilde kontrol edilebilir.

Main class’a eklenmesi gereken anotasyonlar

@SpringBootApplication

Server için:

<dependency>  
 <groupId>org.springframework.cloud</groupId>  
 <artifactId>spring-cloud-starter-gateway</artifactId>  
</dependency>

<dependency>  
 <groupId>org.springframework.cloud</groupId>  
 <artifactId>spring-cloud-starter-circuitbreaker-reactor-resilience4j</artifactId>  
</dependency>

server:

port: 8002

spring:

main:

web-application-type: reactive

application:

name: GATEWAY-SERVER

cloud:

gateway:

routes:

- id: authServer

uri: lb://AUTH-SERVER

predicates:

- Path=/auth/\*\*

filters:

- RewritePath=/(?<segment>.\*), /$\{segment}

- name: CircuitBreaker

args:

name: fallbackAuth

fallbackUri: forward:/fallbackAuthServer

- id: userDetailService

uri: lb://USER-DETAIL

predicates:

- Path=/userDetail/\*\*

filters:

- RewritePath=/(?<segment>.\*), /$\{segment}

- name: CircuitBreaker

args:

name: fallbackUser

fallbackUri: forward:/fallbackUserDetailService

- id: eventService

uri: lb://EVENT-SERVICE

predicates:

- Path=/event/\*\*

filters:

- RewritePath=/(?<segment>.\*), /$\{segment}

- name: CircuitBreaker

args:

name: fallbackEvent

fallbackUri: forward:/fallbackEventService

- id: commentService

uri: lb://COMMENT-SERVICE

predicates:

- Path=/comment/\*\*

filters:

- RewritePath=/(?<segment>.\*), /$\{segment}

- name: CircuitBreaker

args:

name: fallbackComment

fallbackUri: forward:/fallbackCommentService

- id: participantService

uri: lb://PARTICIPANT-SERVICE

predicates:

- Path=/participant/\*\*

filters:

- RewritePath=/(?<segment>.\*), /$\{segment}

- name: CircuitBreaker

args:

name: fallbackParticipant

fallbackUri: forward:/fallbackParticipantService

resilience4j:

circuitbreaker:

instances:

fallbackAuth:

slidingWindowType: COUNT\_BASED #Kayıt altına alınan data kıstası. Saniye olarakta belirlenebilmektedir.

slidingWindowSize: 100 #Son x datanın dikkate alınmasını sağlar.

minimumNumberOfCalls: 50 #Asgari kayıt sayısını belirler. Bu kayıt sayısından sonra algoritmalar çalışacaktır.

permittedNumberOfCallsInHalfOpenState: 20 #Sigora yarı açıkken yapılacak max kayıt sayısı.

automaticTransitionFromOpenToHalfOpenEnabled: true #Sigortanın açık konumdan yarı-açık konuma gelmesini üsrecinin atomatize edilmesi sağlar.

failureRateThreshold: 50 #% cinsinden ne kadar bir fail oranı olursa sigorta açık konuma gelecek bilgisi girilir.

waitDurationInOpenState: 10s #Açık olan devre saniye sonra yarı açık hal gelecektir.

slowCallDurationThreshold: 10s #Saniye üzerinde dönen request'leri yavaş olarak kabul edecektir.

slowCallRateThreshold: 90 #Yavaş request sayısı %x'i aşarsa sigorta yarı-açık konuma gelecektir.

registerHealthIndicator: true

fallbackUser:

slidingWindowType: COUNT\_BASED #Kayıt altına alınan data kıstası. Saniye olarakta belirlenebilmektedir.

slidingWindowSize: 100 #Son x datanın dikkate alınmasını sağlar.

minimumNumberOfCalls: 50 #Asgari kayıt sayısını belirler. Bu kayıt sayısından sonra algoritmalar çalışacaktır.

permittedNumberOfCallsInHalfOpenState: 20 #Sigora yarı açıkken yapılacak max kayıt sayısı.

automaticTransitionFromOpenToHalfOpenEnabled: true #Sigortanın açık konumdan yarı-açık konuma gelmesini üsrecinin atomatize edilmesi sağlar.

failureRateThreshold: 50 #% cinsinden ne kadar bir fail oranı olursa sigorta açık konuma gelecek bilgisi girilir.

waitDurationInOpenState: 10s #Açık olan devre saniye sonra yarı açık hal gelecektir.

slowCallDurationThreshold: 10s #Saniye üzerinde dönen request'leri yavaş olarak kabul edecektir.

slowCallRateThreshold: 90 #Yavaş request sayısı %x'i aşarsa sigorta yarı-açık konuma gelecektir.

registerHealthIndicator: true

fallbackEvent:

slidingWindowType: COUNT\_BASED #Kayıt altına alınan data kıstası. Saniye olarakta belirlenebilmektedir.

slidingWindowSize: 100 #Son x datanın dikkate alınmasını sağlar.

minimumNumberOfCalls: 50 #Asgari kayıt sayısını belirler. Bu kayıt sayısından sonra algoritmalar çalışacaktır.

permittedNumberOfCallsInHalfOpenState: 20 #Sigora yarı açıkken yapılacak max kayıt sayısı.

automaticTransitionFromOpenToHalfOpenEnabled: true #Sigortanın açık konumdan yarı-açık konuma gelmesini üsrecinin atomatize edilmesi sağlar.

failureRateThreshold: 50 #% cinsinden ne kadar bir fail oranı olursa sigorta açık konuma gelecek bilgisi girilir.

waitDurationInOpenState: 10s #Açık olan devre saniye sonra yarı açık hal gelecektir.

slowCallDurationThreshold: 10s #Saniye üzerinde dönen request'leri yavaş olarak kabul edecektir.

slowCallRateThreshold: 90 #Yavaş request sayısı %x'i aşarsa sigorta yarı-açık konuma gelecektir.

registerHealthIndicator: true

fallbackComment:

slidingWindowType: COUNT\_BASED #Kayıt altına alınan data kıstası. Saniye olarakta belirlenebilmektedir.

slidingWindowSize: 100 #Son x datanın dikkate alınmasını sağlar.

minimumNumberOfCalls: 50 #Asgari kayıt sayısını belirler. Bu kayıt sayısından sonra algoritmalar çalışacaktır.

permittedNumberOfCallsInHalfOpenState: 20 #Sigora yarı açıkken yapılacak max kayıt sayısı.

automaticTransitionFromOpenToHalfOpenEnabled: true #Sigortanın açık konumdan yarı-açık konuma gelmesini üsrecinin atomatize edilmesi sağlar.

failureRateThreshold: 50 #% cinsinden ne kadar bir fail oranı olursa sigorta açık konuma gelecek bilgisi girilir.

waitDurationInOpenState: 10s #Açık olan devre saniye sonra yarı açık hal gelecektir.

slowCallDurationThreshold: 10s #Saniye üzerinde dönen request'leri yavaş olarak kabul edecektir.

slowCallRateThreshold: 90 #Yavaş request sayısı %x'i aşarsa sigorta yarı-açık konuma gelecektir.

registerHealthIndicator: true

fallbackParticipant:

slidingWindowType: COUNT\_BASED #Kayıt altına alınan data kıstası. Saniye olarakta belirlenebilmektedir.

slidingWindowSize: 100 #Son x datanın dikkate alınmasını sağlar.

minimumNumberOfCalls: 50 #Asgari kayıt sayısını belirler. Bu kayıt sayısından sonra algoritmalar çalışacaktır.

permittedNumberOfCallsInHalfOpenState: 20 #Sigora yarı açıkken yapılacak max kayıt sayısı.

automaticTransitionFromOpenToHalfOpenEnabled: true #Sigortanın açık konumdan yarı-açık konuma gelmesini üsrecinin atomatize edilmesi sağlar.

failureRateThreshold: 50 #% cinsinden ne kadar bir fail oranı olursa sigorta açık konuma gelecek bilgisi girilir.

waitDurationInOpenState: 10s #Açık olan devre saniye sonra yarı açık hal gelecektir.

slowCallDurationThreshold: 10s #Saniye üzerinde dönen request'leri yavaş olarak kabul edecektir.

slowCallRateThreshold: 90 #Yavaş request sayısı %x'i aşarsa sigorta yarı-açık konuma gelecektir.

registerHealthIndicator: true

timelimiter:

instances:

fallbackAuth:

timeoutDuration: 20s #Request kaç saniye sonra time-out olacak bilgisi girilir. İlgili request time-out'tan sonra fallback'e düşecektir.

fallbackUser:

timeoutDuration: 20s

fallbackEvent:

timeoutDuration: 20s

fallbackComment:

timeoutDuration: 20s

fallbackParticipant:

timeoutDuration: 20s

eureka:

client:

registerWithEureka: true

fetchRegistry: true

registryFetchIntervalSeconds: 5

serviceUrl:

defaultZone: http://${eurakaUsername}:${eurakaPassword}@${eurakaIP}

healthcheck:

enabled: true

management:

endpoints:

web:

exposure:

include: "\*"

logging:

config: ${logbackURL}

# Spring Boot İle ilgili:

1. Feign Client nedir?

Feign client service’lerin birbiri içinde replica oluşturmasıdır. BU şekilde service’ler arası iletişim daha rahat kodlanabilir hale gelmektedir.

@FeignClient(value = Constants.*AuthServerEurekaName*, path = Constants.*UserServiceFeignClientRequestMapping*)  
public interface AuthServerFeignClient {  
  
 @GetMapping(Constants.*GetEntity*+"{id}")  
 CommonUserEntity getEntity(@RequestHeader(Constants.*AuthorizationHeader*) String token, @PathVariable("id") Long id);  
}

1. RestTemplate sınıfı service’ler arası istek yollamamızı sağlar. Feign client gibi aynı problemi gidermek için kullanılır. LoadBalancerClient sınıfı ile eureka’da balancing işlemin’e göre servicei discover eder.

@Autowired  
LoadBalancerClient loadBalancerClient;//LoadBalancerClient eureka registerları arasından balancing'e göre service'i bulur.  
 \*/  
  
/\*@Bean("getUserEntityByID")  
@Scope(value = "prototype")  
public CommonUserEntity getUserEntityByID(Long userId, String token) {  
 ServiceInstance serviceInstance = loadBalancerClient.choose(Constants.AuthServerEurekaName);  
 if (serviceInstance != null) {  
 String url = serviceInstance.getUri().toString();  
 url = url.concat(Constants.GetEntity.concat(userId.toString()));  
 RestTemplate restTemplate = new RestTemplate();  
 HttpHeaders headers = new HttpHeaders();  
 headers.add(Constants.AuthorizationHeader, token);  
 HttpEntity<String> request = new HttpEntity<String>(headers);  
 try {  
 ResponseEntity<CommonUserEntity> response = restTemplate.exchange(url, HttpMethod.GET, request, CommonUserEntity.class);  
 return response.getBody();  
 } catch (Exception e) {  
 throw new UserNotFoundException(Constants.UserID.concat(userId.toString()));  
 }  
 } else {  
 throw new ServiceNotActive(Constants.AuthServerEurekaName);  
 }  
}

1. @Aspect ile istenilen methodun, class’in öncesinde sonrasında vs işlem yapılabilir.

@Target(ElementType.*METHOD*)  
@Retention(RetentionPolicy.*RUNTIME*)  
public @interface myAn {  
 String name();  
}

@Aspect  
@Configuration  
public class aop {  
 @Before("execution(\* com.controller.aopCOntroller.\*(..)) && @annotation(sampleAnnotation)")  
 public void before1(JoinPoint jp, myAn sampleAnnotation){  
 System.*out*.println(sampleAnnotation.name());  
 System.*out*.println("before method");  
 }  
  
 @After("execution(\* com.controller.aopCOntroller.\*(..)) && @annotation(sampleAnnotation)")  
 public void after1(JoinPoint jp, myAn sampleAnnotation){  
 System.*out*.println(sampleAnnotation.name());  
 System.*out*.println("before method");  
 }  
  
 @Around("execution(\* com.controller.aopCOntroller.\*(..)) && @annotation(sampleAnnotation)")  
 public void around1(ProceedingJoinPoint jp, myAn sampleAnnotation) throws Throwable {//ProceedingJoinPoint.proceed() return value'yu alır.

System.*out*.println(jp.proceed());  
 System.*out*.println(sampleAnnotation.name());  
 System.*out*.println("after method");  
 }  
  
 @AfterReturning("execution(\* com.controller.aopCOntroller.\*(..))")  
 public void afterReturning1(JoinPoint jp){//ProceedingJoinPoint.proceed() return value'yu alır.  
 System.*out*.println("after method");  
 }  
  
 @AfterThrowing("execution(\* com.controller.aopCOntroller.\*(..))")  
 public void afterThrowing1(JoinPoint jp){//ProceedingJoinPoint.proceed() return value'yu alır.  
 System.*out*.println("after method");  
 }  
}

@Before: Method devreye girmeden önce çalışır.  
@AfterReturning: Method başarılı sonuçlandıktan sonra çalışır.  
@AfterThrowing: Methodun exception dönmesi durumunda çalışır.  
@After: Returning ve Throwing her iki durumdada çalışır. (finally)  
@Around: Method devreye girmeden önce ve metod bittikten sonra çalışır*.*

1. hasCode - == - equals()

hasCode -> objenin heap’te tutulan adresine göre oluşturulan code.

== -> objelerin heap’teki adreslerini kıyaslar.

Equals() -> Object sınıfı için default == ile aynı çalışır ancak override edilerek class’ların içeriğinde ki field’lar kıyaslanabilir. Equals’ı true dönen nesnelerin hashCode’u aynı olmalıdır! Equals false olanların hasCode’u aynı olabilir ancak hasp map gibi collectionslar için farklı olmalıdır.

1. Functional Interface – Double Colon Operator

public class Main {  
 public static void main(String[] args) {  
 functionalInterface x = () -> {  
 *myMethod*();  
 };  
 x.testMethod();  
  
 functionalInterface x2 = Main::*myMethod*;//Double colon format -> ClassName::Mehod name. Double colon method referance olarak kullanılır.  
 *genericMethod*(x2);  
  
  
 }  
 interface functionalInterface{//Functional interface ile methodlar parametre olarak geçilir.  
 void testMethod();  
 }

public static void myMethod(){  
 System.*out*.println("Functional Interface!");  
 }  
 public static void genericMethod(functionalInterface a){  
 System.*out*.println("Before method!");  
 a.testMethod();  
 System.*out*.println("After method!");  
 }  
}

Şeklinde oluşturulur. Amac interface’i istenilen yere parametre geçmektedir. Bu şekilde interface’in içinde oluşturulan yapı istenilen yerde kullanılır.

1. Dependency Injection

Amac generic bir sınıfa istenilen sınıfları parametre geçmektedir.

public class Main {  
 public static void main(String[] args) {  
 Automative bmw = new Automative(new BMW());  
 bmw.drive();  
 Automative mercedes = new Automative(new Mercedes());  
 mercedes.drive();  
 }  
  
 interface Skill {  
 void drive();  
 }  
 static class Automative implements Skill{  
 Skill skill;  
 public Automative(Skill skill) {  
 this.skill = skill;  
 }  
  
 @Override  
 public void drive() {  
 skill.drive();  
 }  
 }  
 static class BMW implements Skill {  
  
 @Override  
 public void drive() {  
 System.*out*.println("BMW Skill");  
 }  
 }  
 static class Mercedes implements Skill {  
  
 @Override  
 public void drive() {  
 System.*out*.println("Mercedes Skill");  
 }  
 }  
}

1. Scheduler Async (Jobs)

Belirli sürelerde çalışmasını isteğimiz fonks. Job deriz. Örneğin her saat rapor al vs.

private static final int *DEFAULTPOOL* = 5;  
private static final int *MAXPOOL* = 30;  
private static final int *SURVIVETIME* = 10;  
private static final int *BUFFERSIZE* = 30;  
private static final String *THREADNAMEPREFIX* = "asyncSchedule-";  
@Bean("asyncScheduleBean")  
public ThreadPoolTaskExecutor getAsynsScheduled()  
{  
 ThreadPoolTaskExecutor executor = new ThreadPoolTaskExecutor();  
 executor.setCorePoolSize(*DEFAULTPOOL*);  
 executor.setMaxPoolSize(*MAXPOOL*);  
 executor.setQueueCapacity(*SURVIVETIME*);  
 executor.setKeepAliveSeconds(*BUFFERSIZE*);  
 executor.setThreadNamePrefix(*THREADNAMEPREFIX*);  
 executor.setRejectedExecutionHandler(new ThreadPoolExecutor.CallerRunsPolicy());  
 executor.initialize();  
 return executor;  
}  
@Scheduled(cron="\*/10 \* \* \* \* \*" )//Every 10 seconds.  
@Async("asyncScheduleBean")  
public void reflesh(){  
 System.*out*.println(System.*currentTimeMillis*());  
}

1. OOP Temel Presipleri
   1. -> Encapsulation : private public protected gibi variable’ları muhafa etmek anlamına gelir.
   2. -> Abstraction -> Sınıfların methodların yapılan işlemi soyutlamasına output odaklı davranmalarını sağlar.
   3. -> Inheritance -> Sınıflar arasından child – parent ilişkiler kurmamızı sağlar.
   4. -> Polymorphism -> methodlara farklı çıktılar vermelerini sağlar. @Overriding olarak blinir.
2. SOLID Prensipleri
   1. -> Single Responsibility -> Sınıfların sadece bir sorumululuğu olmalı. Birden fazla sorumuluk karışıklığa sebep olur.
   2. Open Closed Principle -> class’lar değişime kapalı gelişime açık olmalı.
   3. Liskov Principle -> Kalıtım alınan sınıflar ilgili class’ta kullanılmamalı Örn tavuk kuş class’ından inherit olmamalı.
   4. Interface Segregation -> Interfaceler olabildiğince küçük ve anlaşılır şekilde bölünmelidir. Herşey 1 interface üzerinde yürütülmemelidir.
   5. Dependency Inversion -> Üst seviyeli class’lar alt seviyeli class’lara muhtaç olmamalı bu ilişki interface – abstract class’lar ile sağlanmalıdır.
3. Spring Data @DynamicUpdate

Değiştirilen entity’nin db’de güncellenmesi için .save(updatedEntity) kullanılıbilir ancak bu şekilde değiştirilmemiş olan fieldlarda update query’sinde yer alacaktır.

Örn:

@Entity

Class insan {

String name;

String Lastname;

Integer yas;

}

İnsan currentInsan = insanRepo.findbyId(1); -> mevcut olan insan çağırıldı.

currentInsan.setAge(20);//entity’i güncellendi

insanRepo.save(currentInsan) -> db’ye güncel hali kayıt edildi.

Bu şekilde arka planda spring data tarafından aşağıda ki gibi bir query generate edilecektir.

Update tableInsan set name = ?, lastName = ?, yas = ? where id = ?

* Sadece yas field’i değişmesine rağmen tüm fieldlar güncellenecektir.

@Entity

@DynamicUpdate

Class insan …. -> şekilinde yapılırsa sadece değişen field için update query generate edilecektir yani -> update tableInsan set yas = ? where id = ?

* Normalde spring data app ayağa kalktığında tüm sql query’lerini generate eder yani updat query’si app ayağa kalkarken oluşturulur ve ilgili field’lar parametre olarak bu query’de kullanılır. Ancak @DynamicUpdate’de değişen field’lara göre o esnade query generate edilir dolayısıyla bu kısımda perfornmans gerektiren bir işlem olacaktır. Spring data’nın o entity’i sürekli izlemesi ve değişen field’lara göre update query’si generate etmesi masraflı bir iş olabilmektedir. Dolayısıyla @DynmaicUpdate gerçekten gerekli yerlerde özellikle çok field ve ilişkisi fazla olan entity’ler için kullanımı mantıklıdır. Diğer türlü maliyetli bir işlem olacaktır.

1. Stream API nedir?

Stream api collection’lar üzerinde işlem yapmamızı sağlayan bir api’dir. Lambda expr. Formatı ile işlem yapar.

paralelStream() methodu önemli bir methoddur. Async olarak list’de ki elemanları döndürür. Yani örneğin 10 elemanlı bir listede işlem yapacak olursak ve her işlem 1 sn olacaksa o işlem toplam 10sn sürer ancak paralelStream ile yapılan işlemlerde async olarak her eleman farklı thread tarafından işlem görecektir ve çok daha hızlı sonuçlanakcaktır.

1. Try – Resource

Amaç açılan connection’ların otomatiklar oalrak kapanmasıdır. Connection exception fırlatsa bile kesin kapanacaktır.

**try** (**Scanner** scanner = **new** **Scanner**(**new** **File**("testRead.txt")); **PrintWriter** writer = **new** **PrintWriter**(**new** **File**("testWrite.txt"))) { **while** (scanner.hasNext()) { writer.print(scanner.nextLine()); } }

şeklinde oluşturulur.

1. Spring Caching

**@Cacheable(“cacheName”)** → Daha önce ilgili key değerine (cacheName) değer yazılmamışsa önce önbelleğe yazılır. Daha sonra kullanıcıya ilgili değerler döner. Eğer önbellekte ilgili bilgi varsa önbellekten çağrılacaktır. Örneğin, bir metodun üzerinde @Cacheable(“movies”) ifadesi yazılıysa bu metot ilk çağrıldığında veri tabanından gelecektir. İkinci defa çağrıldığında önbellekten gelecektir.

**@CacheEvict(“cacheName”)** → Önbellekte ilgili key ile yazılmış değer varsa onu silecektir. Örneğin, @Cacheable(“movies”) adında bir metot çağırdık ve önbelleğe film listesini yazdı. Daha sonra @CacheEvict(“movies”) ifadesi olan metodu çalıştırırsak, bellekteki film listesini silecektir.

**@CachePut(“cacheName”)** → Eğer bir metodun her defasında çalışmasını istiyorsak, bu anotasyonun bulunduğu metodu kullanırız. Bu sayede önbellekte her zaman en güncel veriler olacaktır. Örneğin, @CachePut(“movies”) yazılı metot çalıştırıldığında veri tabanına gider ve veri tabanındaki en güncel bilgiyi önbelleğe yazar. Daha sonra Cacheable(“movies”) anotasyonu bulunan metot çağrılırsa, en güncel bilgiyi verecektir.

Bu kullanımlara ek olarak, farklı kullanım şekilleri de mevcuttur. Direkt olarak örnek üzerinde anlatayım.  
**@Cacheable(value = “movies”, key = “#movieId”)**Şeklinde de kullanabiliriz. Bu işlem tüm filmleri değil sadece movieId değeri key olarak verilmiş olan filmleri önbelleğe yazar ve oradan getirir.

Eğer uygulamadan önbellek özelliğini kaldırmak istersek, tüm anotasyonları silmek yerine **application.properties** dosyasına**spring.cache.type=none** satırı eklenir.

1. Spring Boot Repository Null Param Query

Bu şekilde query yazarak filtreleme yapılarbilir. Önemli bir konu. Örneğin:

@Query(value = "SELECT u from ProductEntity u where (:region is null or u.storage.address.region = :region) and (:city is null or u.storage.address.city = :city) and (:fullAddress is null or u.storage.address.fullAddress = :fullAddress) and (:category is null or u.category = :category)")  
List<ProductEntity> filterProductEntity(@Param("region") Regions region, @Param("city") String city, @Param("fullAddress") String fullAddress, @Param("category") CategoryEntity category);

1. @Where

Entity’nin üzerinde kullanılır. Repository o condition’a uyarak dataları getirir. Örneğin

@Where(clause=”deleted = false”) gibi. Bunu ekledikten sonra repository deleted field’i false olan dataları getirecektir her zaman.

@Entity

@Where(clause = “deleted = false”)