

Daca nu oferim un nume unui bean in xml, Spring singur va alege un nume, dar va fi asa NumeleLaClasa#Numar

**@Lookup**

* Rolul anotatiei @Lookup este de a crea o method factory
* Tot ce trebuie sa facem e sa punem anotatia data deasupra unei metode care returneaza un tip al unui object, si metoda data poate returna null, oricum anotatia data va face ca metoda sa fie suprascrisa
* Este folositoare cand vrem sa obtinem beanuri prototype intr-un bean singleton
* Fie ca avem clasa Teacher si Course. Teacher e prototype, course e singleton
* @Autowired tot poate injecta un bean de tip prototype, dar s-ar putea sa avem nevoie in unele metode de la Course sa obtinem mai multe beanuri de tip Teacher ce sunt singleton, si unica optiune ar fi cu un ApplicationContext, dar putem mai bine crea un method factory la apelrea careia spring ca oferi automat un nou bean prototype din container

@Component  
public class Course {  
 private String title;  
  
  
 public String getTitle() {  
 return title;  
 }  
  
 public void setTitle(String title) {  
 this.title = title;  
 }  
 @Lookup  
 public Teacher getTeacher(){  
 return null;  
 }  
  
  
 @Override  
 public String toString() {  
 return "Course{" +  
 "title='" + title + '\'' +  
 '}';  
 }  
}

La apelarea metodei getTeacher, se va returna mereu un nou Teacher creat din container,

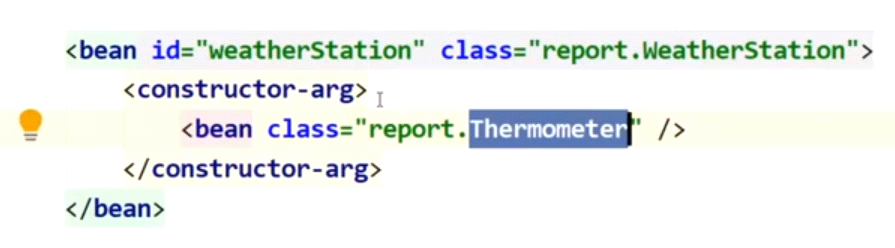
**ObjectFactory**

Este doar o interfata functionala

@FunctionalInterface  
public interface ObjectFactory<T> {  
  
T getObject() throws BeansException;  
  
}

**Custom scope**

**Inline bean definition**



Acest bean e disponibil doar pentru constructor si gata.

* Ordinea in care punem beanurile in constructor-arg nu conteaza. Ordinea nu trebuie sa coincida cu cea din costructor, dar daca in constructor se cer 2 obiecte de acealasi fel, atunci Spring nu va sti ordinea necesara.

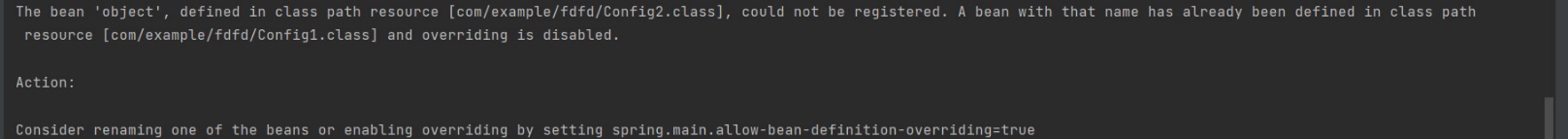
**Ierarhia**

* Cand mai multe clase Configuration declar beanuri cu id identic, aplicatia va da crash, din cauza ca in mod default spring nu permite suprascrierea beanurilor deja existente in Container
* De ex, fie clasele:
* @SpringBootApplication  
  public class FdfdApplication {  
    
   public static void main(String[] args) {  
   SpringApplication.*run*(FdfdApplication.class);  
   }  
   @Bean  
   public Object object(){  
   System.*out*.println("Main class");  
   return new Object();  
   }  
    
  }

@Configuration  
public class Config1 {  
 @Bean  
 public Object object(){  
 System.*out*.println("Config1");  
 return new Object();  
 }  
}

@Configuration  
public class Config2 {  
 @Bean  
 public Object object(){  
 System.*out*.println("Config2");  
 return new Object();  
 }  
}

Vedem ca avem in fiecare config class beanuri cu id identic, si daca vom da run, primim eroare:



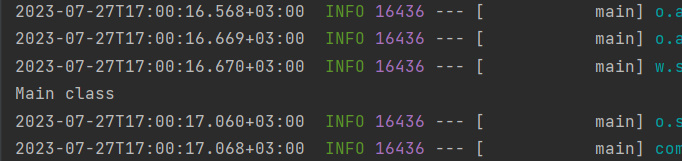
Deci, ne spune ca cand s-a ajuns la config2, nu a putut crea un bean object, caci el deja a fost creat in config1

* Unica solutie este de a permite ca beanurile sa fie suprascrise:

spring.main.allow-bean-definition-overriding=true

Si totul va fi bine

* Acum, cand vom face asta, vom vedea ca nu s-au creat 3 beanuri, ci doar unul, si anume cel din main class:



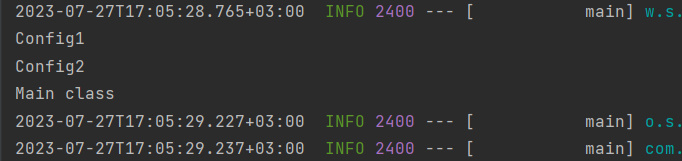
Deci, ordinea a fost asa:

1. Intai s-a dus la Config1, apoi la Config2 si apoi la main class. Asta e din cauza ca in main class se pune @ComponentScan, deci cand se va ajunge la anotatia data, se va incepe scanarea la clasele config, si abea apoi la urma va trece la main class, sau la urmatoarele anotatii.
2. Crearea la toate beanurile deodata ar putea fi costitistoare, de aceea Spring nu creaza deodata beanurile cand citeste un config file. El aduna toate informatiile din toate config classes, si abea dupa ce le scaneaza pe toate creaza beanurile in IoC Container.
3. Intai a colectat datele din Config1, apoi din Config2, si Config2 a suprascris un bean ce vine din config1, apoi s-a dus la main class, si el a suprascris informatia despre object bean din nou. La final IoC Container a citit tot ce a scanat de prin config classes si a creat beanurile, si logic ca daca main class a suprascris ultimul, s-a creat bean din el. Asa se si optimizeaza aplicatiaa

* **Beanurile nu se citesc niciodata incepand cu clasa unde se gaseste @SpringBootApplication! Ea mereu e ultima**
* De ex:
* @SpringBootApplication  
  public class FdfdApplication {  
    
   public static void main(String[] args) {  
   SpringApplication.*run*(FdfdApplication.class);  
   }  
   @Bean  
   public Object object1(){  
   System.*out*.println("Main class");  
   return new Object();  
   }  
    
  }

@Configuration  
public class Config1 {  
 @Bean  
 public Object object2(){  
 System.*out*.println("Config1");  
 return new Object();  
 }  
}

@Configuration  
public class Config2 {  
 @Bean  
 public Object object3(){  
 System.*out*.println("Config2");  
 return new Object();  
 }  
}



**@Autowired la field**

* @Autowired la field nu e recomandat deoarece la teste putem avea probleme. Field injection nu sunt obligaorii, si problema e ca la teste am putea avea probleme din cauza asta, caci s-ar putea sa nu stim ca un bean nu a fost injectat

**Injectarea cu @Bean**

* Avem 2 variante de a injecta beanuri cand cream unul. Prima:

@Bean  
public Teacher teacher(){  
  
 return new Teacher("Baby","Boom");  
}  
@Bean  
public Course course(){  
 Course course = new Course(teacher());  
 course.setTitle("bean");  
 return course;  
}

* Dar mai avem si varianta a 2:

@ComponentScan(basePackages = {"org.java.test"})  
public class AppContext {  
 @Bean  
 public Teacher teacher(){  
  
 return new Teacher("Baby","Boom");  
 }  
 @Bean  
 public Course course(Teacher teacher){  
 Course course = new Course(teacher);  
 course.setTitle("bean");  
 return course;  
 }  
}

Cand Spring va vedea ca este un argument la metoda de creare a beanului, el va intelege deodata ca e vorba e o depenenta si o va cauta.

**@ImportResource**

* Daca intr-o clasa @Config vrem sa importam o configuratie din xml, folosim @ImportResource("fisierXML")
* Pentru a importa o alta configuratie sau componen, daca nu e activa component scan, folosim @Import

**Circular Injection**

**Fie clasa A si clasa B:**

@Component  
public class A {  
 private B b;  
 public A(B b){  
 this.b = b;  
 }  
}

@Component  
public class B {  
 private A a;  
 public B(A a){  
 this.a = a;  
 }  
}

**La rularea aplicatie, vom primi exceptie ca avem o dependenta circulara.**

Problema e ca A cand se creaza, va avea nevoie de B, apoi cand B se va crea, el va avea nevoie de A si nici unul nu reuseste sa se creeze, ci unul mereu incearca sa il creeze pe celalalt. Ideea e ca putem folosi Setter Injection, caci atunci deja beanurile vor fi create.

**Setter injection si Fields injection(nerecomandat) sunt pentru dependente optionale, constructor injection pentru cele obligatorii.**

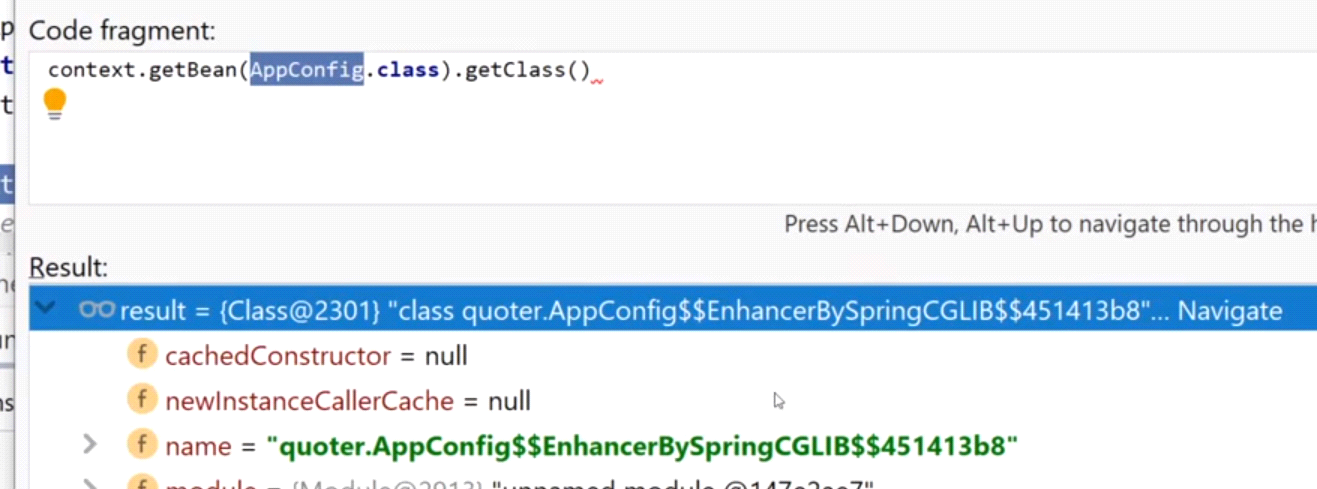
**Proxy**

* **Daca ne uitam la programl dat,**

@Configuration  
@ComponentScan(basePackages = {"org.java.test"})  
public class AppContext {  
 @Bean  
 public Teacher teacher2(){  
 System.*out*.println("New teacher");  
 return new Teacher("Baby","Boom");  
 }  
 @Bean  
 public Course course2(){  
 teacher2();  
 teacher2();  
 teacher2();  
 teacher2();  
  
 Course course = new Course(teacher2());  
 return course;  
 }  
}

apare o confuzie. teacher2() e o metoda, eaa va genera un singleton bean. Totusi, vedem ca la crearea lui course2, metoda course2() apeleaza de 5 ori metoda teacher(). Ar parea ca ar trebui sa fie afisat pe ecran "New teacher" de 5 ori si sa fie create 5 obiecte, dar asta ar incalca singleton principle.

* In realitate, pe ecran a aparut o singura data "New teacher" si a fost creat un singur obiect teacher2, si mesajul a aparut la crearea beanului teacher2, nu la course2
* Asta se intampla din cauza ca clasa AppContext are anotatia @Configuration, si spring a creat un proxy, adica o noua clasa ce extinde clasa AppContext, si acest proxy, sau clasa noua, se asigura sa execute metodele o singura data. Acum, cand vom apela metoda teacher2(), nu se va apela propriu zis metoda noastra, caci apelarea se va face in proxy, si deci spring va modifica metoda, o va suprascrie, si va modifica comportamentul ei ca sa se asigure sa returnezee pur si simplu un obiect si gata,



Vedem ca defapt primim nu un AppConfig, ci un un class AppConfig$$EnchanceryBySpringCGLIB$$451413b8....

@Configuration  
@ComponentScan(basePackages = {"org.java.test"})  
class AppContext {  
 @Bean  
 public Teacher teacher2(){  
 System.*out*.println("New teacher");  
 return new Teacher("Baby","Boom");  
 }  
 @Bean  
 public Course course2(){  
 teacher2();  
 teacher2();  
 teacher2();  
 teacher2();  
  
 Course course = new Course(teacher2());  
 return course;  
 }  
}  
  
  
class AppContextProxy extends AppContext {  
 private Teacher teacherBean;  
 private Course courseBean;  
 public Teacher teacher2(){  
 if(teacherBean==null)  
 teacherBean = super.teacher2();  
 return teacherBean;  
 }  
 public Course course2(){  
 if(courseBean == null)  
 courseBean = super.course2();  
 return courseBean;  
 }  
}

Cand va apela super.course2(), metoda course2() din superclasa va apela metoda teacher2 din superclasa, dar, deoarece metodele sunt virtuale, se va executa metoda teacher2 din clasa proxy.

Putem face ca sa nu se creeze un proxy:

@Configuration(proxyBeanMethods = false)

Si in asa situatie, va aparea pe ecran de 5 ori "New Teacher", dar se incalca Singleton, si deci va fi prototype. Nu folosim asta mai bine

**Inner Beans**

* Putem crea beanuri chiar si in @Component.
* Totusi, atentie, pentru @Component nu se creaza proxy, si deci se va primi iar problema

@Component  
public class Course {  
 private String title = "component";  
 private Teacher teacher;  
  
 public Course() {  
 }  
 @Bean  
 public Course course1(){  
 System.*out*.println("Create course");  
  
 return new Course();  
 }  
 @Bean  
 public String bean(){  
 course1();  
 course1();  
 course1();  
  
 return "Hei";  
 }

}

La ecran a fost afisat de 4 ori "Create course"

Beanurile s-au creat dupa ce s-a creat acest @Component

**Proxy Annotation @Validated**

* @Validated – si el creaza proxy. Il putem pune si la nivel de clasa, fara sa indicam neaparat un grup pentru validare. Asa, pentru clasa deasupra la care e scrisa anotatia, se creaza un proxy ce va face validarile oriunde va gasi anotatii de validare in acea clasa . Validarile se fac gen cu anotatiile @NotNull, @Past etc. **@Validated va permite validarea doar la nivel de parametri de metoda, nu si la nivel de field**

Anotatiile de validare au efect doar cand sunt activate de @Validated sau @Valid, in rest nu au nici-un efect. Nu conteaza ca acest @Valid sau @Validated se afla in exteriorul lor, ele asteapta ca una dintre ele sa le activeze

* **Totusi, el nu va activa si anotatiile altor beanuri/obiecte din acea clasa. De ex:**
* @NoArgsConstructor  
  @Getter  
  @Setter  
  public class Teacher {  
   @Size(min = 10)  
   private String firstName;  
   @Size(min = 10)  
   private String lastName;  
    
  }
* @RestController  
  @Validated  
  public class REST {  
    
   @PostMapping("/api")  
   public String id(@RequestBody Teacher teacher){  
   return "nice";  
   }  
  }

**Chiar de trimitem un post:**

{

    "firstName" : "Test",

    "lastName" : "User"

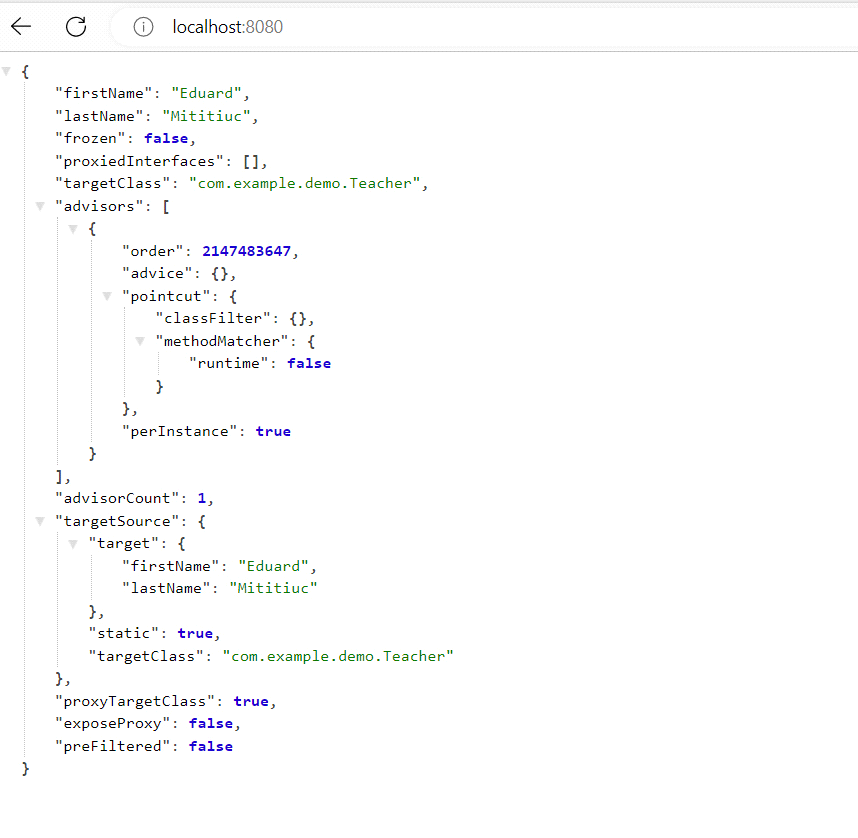
}

validarea nu se va face, si asta din cauza ca pur si simplu nu s-a activat validarea pentru Teacher si gata. Cand vrem sa activam propriu zis validarea, fie folosim anotatii in acea clasa, ca @Min, @Max etc. fie folosim @Valid sau @Validated la acel obiect concret.

* Atentie! Ideea de a pune @Validated deasupra la o clasa ce va fi un Entity de ex e rea de tot. Daca de ex la Teacher am pune @Validated, validarea se va face, dar se va returna obiectul proxy,daca facem un bean, ce are fielduri si metode noi, de aceea, in loc sa primim un simplu teacher cu firstName si Lastname, daca facem asa:
* @Validated  
  @Component  
  public class Teacher {  
   private String firstName;  
   private String lastName;  
    
   public String getFirstName() {  
   return firstName;  
   }  
   public void setFirstName(@Size(min = 5) String firstName) {  
   this.firstName = firstName;  
   }  
    
   public String getLastName() {  
   return lastName;  
   }  
   public void setLastName(@Size(min = 5) String lastName) {  
   this.lastName = lastName;  
   }  
  }

**Atentie! Vedem ca @Size e la nivel de metoda!**

@RestController  
public class REST {  
 @Autowired  
 Teacher teacher;  
  
 @GetMapping("/")  
 public Teacher get(){  
 teacher.setLastName("Mititiuc");  
 teacher.setFirstName("Eduard");  
  
 return teacher;  
 }  
}



* Dar, daca nu avem de gand sa returnam aceste beanuri la user, putem folosi @Validated in clasa lor
* @Validated  
  @Component  
  public class Teacher {  
   private String firstName;  
   private String lastName;  
    
   public String getFirstName() {  
   return firstName;  
   }  
   public void setFirstName( @Size(min = 5) String firstName) {  
   this.firstName = firstName;  
   }  
    
   public String getLastName() {  
   return lastName;  
   }  
   public void setLastName( @Size(min = 5) String lastName) {  
   this.lastName = lastName;  
   }  
  }

@RestController  
public class REST {  
  
 @Autowired  
 Teacher teacher;  
  
 @GetMapping("/api")  
 public Teacher get(){  
 teacher.setFirstName("a");  
 teacher.setLastName("b");  
 get2(teacher);  
  
 return new Teacher();  
 }

**Proxy Annotation @Async**

* Sa zicem ca avem o metoda generateReport() care ia mult timp sa se execute
* Acum, daca o alta metoda, gen createMultipleReports() va apela de

generateReport();

generateReport();

generateReport();

Rularea la aceste 3 reporturi va lua ceva timp, si o solutie buna ar fi sa cream un thread de fiecare data cand metoda generateReport() e apelata. Pentru asta, folosim @Async. Urmam pasii:

* In Configuration punem anotatia:

@EnableAsync

* Cream un bean de tip ThreadPoolTaskExecutor

@Bean  
public ThreadPoolTaskExecutor threadPoolTaskExecutor(){  
 ThreadPoolTaskExecutor poolTaskExecutor = new ThreadPoolTaskExecutor();  
 poolTaskExecutor.setCorePoolSize(3);  
 poolTaskExecutor.setMaxPoolSize(5);  
   
 return poolTaskExecutor;  
}

setCorePoolSize() – setam numarul de threaduri initiale

setMaxPoolSize() – setam cate threaduri poate maximul crea daca cele setate initial nu ajung

* Adaugam @Async asupra la metoda ce sa fie mereu executata intr-un nou thread:

@Component  
public class Count {  
 @Async("threadPoolTaskExecutor")  
 public void count() throws InterruptedException {  
 for(int i = 0;i<10000;i++) {  
 System.*out*.println(i);  
 Thread.*sleep*(500);  
 }  
 }  
}

Punem in @Async numele la bean pentru ThreadPoolTaskExecutor

@Async tot creaza un proxy pentru clasa Count  
**Cacheable in Spring Boot**

* Cacheable ne ajuta prin faptul ca memoreaza cu ce parametri a fost apelata deja o metoda, si daca ea e apelata cumva iar cu aceeasi parametri, se va da deodata rezultatul ce a fost generat odata cu acesti parametri
* Deci, cand apelam o metoda cu niste parametri, acei parametri, impreuna cu rezultatul, se vor memora in cache, si daca metoda va fi apelata din nou, se va cauta prin cache daca acesti parametri nu au fost deja folositi, si daca da, se ia rezultatul deodata din cache, nu se mai executa metoda.
* **Atentie! Cache e bun doar daca suntem siguri ca apelarea unei metode cu aceeasi parametri trebuie sa returneze acelasi rezultat!**
* Cand lucram cu cache abstraction, tinem cont de 2 lucruri:

- **cache declaration** – identificam metodele ce trebuie sa fie cached

- **cache configuration** – cache in care datele sunt stocate si citite

* **Cache** – parte din RAM care face legatura intre aplicatie si baza de date embedded.
* **Este mult mai rapid de a se lua datele din RAM stocate in cache decat a le lua din nou din baza de date**
* De exemplu, Redis tot este un in memeory database
* **Date ce trebuie cached:**

**-** Date ce nu se modifica frecvent

**-** date utilizate frecvent din read query in care rezultatele nu se modifica, cel putin o perioada de timp

* **Tipuri de cache:**
* **- In-Memory Caching -** pastreaza in memorie valori key-value. Un exemplu popular e Redis

**- DataBase Caching -**  mecanizm care genereaza web pages la cerere prin a lua date din baza de date. Cel mai popular database caching este first level cache din Hibernate

**- Web Server Caching** – mecanizm de a stoca datele pentru a fi reutilizate. De exemplu, cand un user acceseaza o pagina, pagina e generata doar prima data, apoi ea este cached, si cand userul mai cere pagina data, ea e preluata deodata din cache.

- **CDN** – vine de la Content Delivery Network. Este o componenta folosita in modern web applications. Imbunatateste transmiterea continutului prin a replica fisierele cerute des folosite, ca HTML pages, images, videos etc. prin difertie caching servers.

* **Pasi de creare:**

1. Punem anotatia @EableCaching in config

2. Punem dependenta:

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-cache</artifactId>

</dependency>

3. Cream un bean de tip CacheManager(**dar el se creaza si automat daca nu o facem noi. Se creaza anume un concurrent HashMap):**

@Bean  
public CacheManager cacheManager(){  
 SimpleCacheManager simpleCacheManager = new SimpleCacheManager();  
 simpleCacheManager.setCaches(List.*of*(new ConcurrentMapCache("report")));  
   
 return simpleCacheManager;  
}

**Pasul 4**

* Putem folosi diferite anotatii in dependenta de situatie
* **@Cacheable(“report”)**

Este pentru a stoca rezultatul returant de metoda pentru parametrii respectivi. “report” e doar un nume la un cache, noi putem pune oricate si oricare. Deci, pe baza la parametrii, se va face un key, si rezultatul va fi value pentru acei parametri. Ofera 2 argumente: value="" si key = "". Value este numele la cache, si key este id pentru valoare. De ex:

@Cacheable(value="cacheStudentInfo", key="#student.id")

public List studentInfo(Student student)

{

//some code

return studentDetails;

}

Dar, putem pune direct numele la cache doar si Spring va genera singur un id @Cacheabale("caheName")

Mai putem pune si condition="conditie" pentru a pastra datele in cache doar in anumite cazuri:

@Cacheable(value="student", condition="#name.length<20")

public Student findStudent(String name)

{

//some code

}

**Inainte de a executa prorpiu zis metoda, se verifica daca nu exista deja un rezultat cached pentru acele argumente in cache**

* **@CacheConfig(cacheNames={nume cache}) –** class level annotation care ofera un comun cache-related setting. El spune lui Spring unde sa stocheze cache pentru acea clasa daca metodele nu ofera un nume la cache in @Cacheable.

@CacheConfig(cacheNames={"employee"})

**public** **class** UserService

{

//some code

}

* **@CacheEvict()** - o folosim pentru a elimina datele din cache care sunt invechite sau neutilizate.

- Pentru a sterge tot cache folosim allEntries=true:

@CacheEvict(allEntries=true)

- Pentru a sterge un entry dupa key:

@CacheEvict(key="#student.stud\_name")

De exemplu, pentru a serge toate datele din cache student\_data

@CacheEvict(value="student\_data", allEntries=true) //removing all entries from the cache

public String getNames(Student student)

{

//some code

}

Daca am folosi @Cacheable peste tot, cache ar creste prea mult, si mai e nevoie sa il curatam.

* **@CachePut("cacheName")** - este pentru a da update la datele din cache pe baza la e returneaza metoda. Putem folosi si argumentele cacheNames si key:

@CachePut(cacheNames="employee", key="#id") //updating cache

public Employee updateEmp(ID id, EmployeeData data)

{

//some code

}

Spre deosebire de @Cacheable, @CachePut mereu va rula metoda si va da update la cache.

Putem sa punem conditie si baza pe rezult return:

@CachePut(value="addresses", unless="#result.length()<64")

public String getAddress(Customer customer) {...}

* **@Caching() -** o folosim cand avem nevoie sa folosim si @CachePut si @CacheEvict, caci Java nu permite folosirea la 2 anotatii de acelasi tip, si @Caching e o solutie:

@Caching(evict = {@CacheEvict("phone\_number"), @CacheEvict(value="directory", key="#student.id") })

public String getAddress(Student student)

{

//some code

}

**Customize existing cache manager**

* Spring Boot deja creaza un ConcurrentMapCacheManager Bean, si daca vrem sa il customizam, sau poate si pe al nostru, extindem clasa CacheManagerCustmizer<ConcurrentMapCacheManager> si folosim metoda customize()

@Component

public class SimpleCacheCustomizer

implements CacheManagerCustomizer<ConcurrentMapCacheManager> {

@Override

public void customize(ConcurrentMapCacheManager cacheManager) {

cacheManager.setCacheNames(asList("users", "transactions"));

}

}

Aici, setCacheNames va adauga noi caches.

**Practic**

@GetMapping("/{id}")  
@Cacheable("student")  
public Student getById(@PathVariable int id){  
 return studentDAO.findById(id).get();  
}

La rularea la localhost:8080/1, doar prima data Spring va face un select din baza de date, mai departe nu va mai face select, ci va lua datele din baza de date

Chiar daca un cache cu numele "student" nu exista, el se va crea automat.

* Totusi, aici apare problema. Daca de exemplu userul sta in aplicatie ceva timp, si in acel timp se modifica useru cu id 1, aceste modificari nu vor fi vazute, caci in cache se pastreaza vechile date. Asta cam e o problema la cache.