Sring

Spring boot : (Mix Français, Anglais, Arabe).

=> Lors Creation d'un projet spring il faut choisir ces dependences :

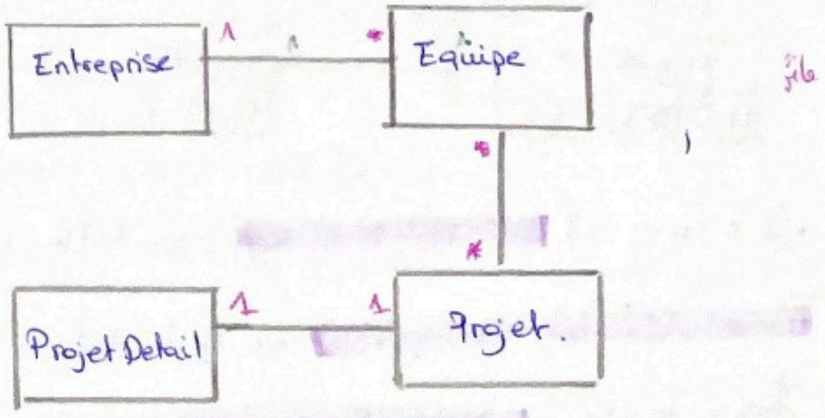
- Lombok ; • web -> Spring Web ; • Spring Data Jpa ;
- MySQL Driver.

=> Creation les dossiers (Entités ; Controlleurs, Models)

- Les entités: just follow the structure of exams.
- all the attributs are private (then go the visibilty we have to do the get and the set methods).
- For Enum (just write the attributs bellow).
- For calling the ~~Enum~~ enum in the entity file we just write like this (@Enumareted Speciality speciality).

I) Associations

- Diagramme en général



- One to One (Unidirectionnelle).

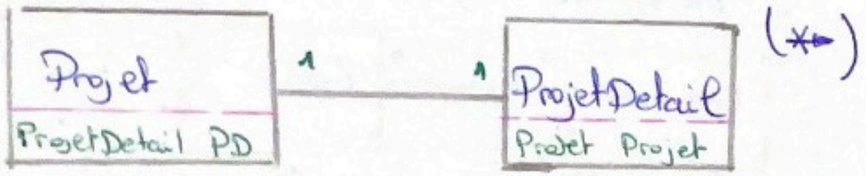
- so the cursor goes to Projet Detail so that means Projet got the visibility on Projet Detail But not the same case of Projet Detail

Exemple of Entity

```
@Entity
public class Projet {
@OneToOne
private Projet_Detail projet_detail;
}
```

Nothing to add in Projet_Detail entity file (only in Projet entity)

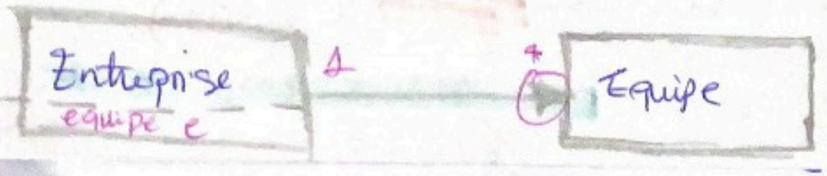
- OneToOne (Bidirectionnelle).



- In case of Bidirectional way, we talk about Parent/child relationship => so to defeny it we have to use "mappedBy" in the child section.

```
public class Projet                    | public class ProjetDetail
                                       |
@OneToOne                              | @OneToOne(mappedBy=
                                       | "projetDetail")
private ProjetDetail                   | private Projet projet
projetDetail                           |
```

- OneToMany (Unidirectionnelle).



=> As the case bellow, "Entreprise" has only the access of the visibility on "Equipe"

- public class Entreprise {

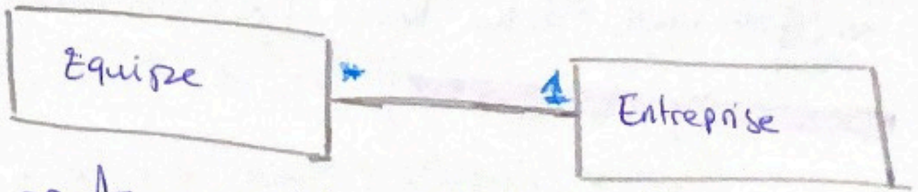
@OneToOne (cascade = CascadeType.ALL)

L'entreprise peut manipuler (ajouter, modifier,) plusieurs équipes.

private set < Equipe > Equipes (many = List)

=> Nothing to add for equipe.

- Many to One (Bidirectionnelle) = OneToMany (bidirectionnel)
- OneToMany Bidirectionnelle = ManyToOne Bidirectionnelle.



=> As we said ... it's Bidirectional way so we have to use ("mappedBy") to define Parent/child who got the many cardinality is the parent

public class Equipe {

@ManyToOne

Entreprise enprise; }

public class Entreprise {

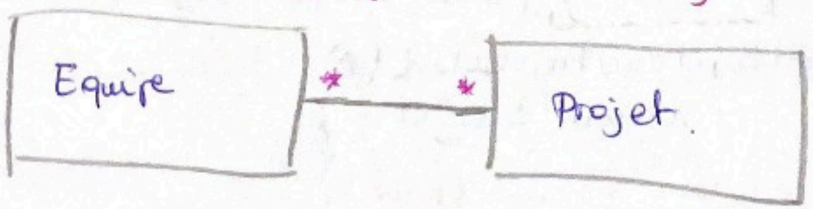
@OneToMany (cascade = CascadeType.All

mapped By = "entreprise") ↑ By logic

private set < Equipe > Equipes; }.

↑ Many

• ManyTo Many (Bidirectionnelle)



public class Equipe {

@ManyToMany (cascade = CascadeType.All)

private set < Projet > projets; }

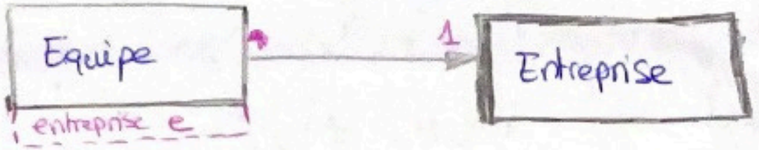
public class Projet { mappedBy = "projets") {

@ManyToMany

private set < Equipe > equipes; }

cascade...

• ManyToOne (Unidirectionnelle)



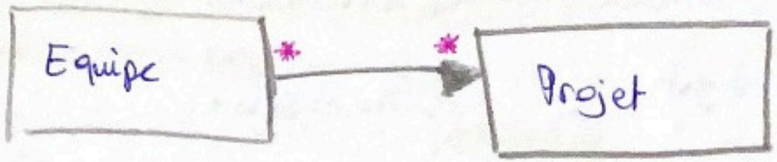
public class Equipe {

@ManyToOne (cascade = CascadeType.All)

Entreprise entreprise; }.

↑ peut manipuler une entreprise

• ManyToMany (Unidirectionnelle)



public class Equipe { cascade = CascadeType.All }

@ManyToMany.

private set < Projet > projets; }

For the importations
we got @Getter and @Setter : so we dont have to wrote to full codes
@NoArgContrustor (constructeur feragh)
@AllArgContrustor (constructeur)
@ToString(ihawel interger l string)
@Entity (ki nhebou yasn3el tableau fil base de donnée donc lezemna l nzidou primary key)
@Id (lil primary key)
@GeneratedValue (strategy = GenerationType.IDENTITY) “for render it autoincrement”

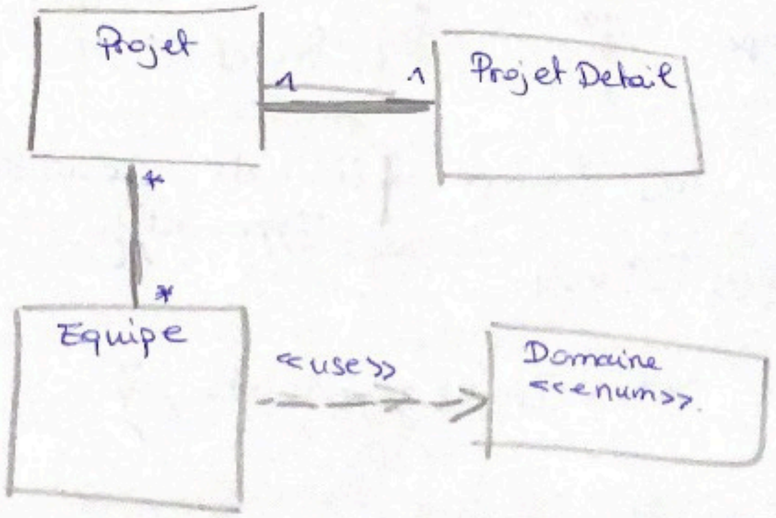
For connection to Database we got to
applicaition.properties file and we add those lines
spring.datasource.url=jdbc:mysql://localhost:3306/medecin?createDataBaseIfNotExist=true
spring.datasource.username=root
spring.datasource.password=
spring.jpa.show-sql=true
spring.jpa.hibernate.ddl-auto=update (update database kol matzid haja jdida)

Fetch

@ManyToMany (fetch = FetchType.Eager)
private set <projet> projets.

- FetchType.Eager (avec impatience) :
Quand on récupère une équipe de la basse de données tous les projets liés à cette equipe seront récuperés eux aussi.

- Affectations :



Cas 1:

Permet d'Ajouter un Projet et un ProjetDetail (Cascade.)
et affecter le projet Detail en projet ← Service + file

=> ServiceImp

public Projet addProjetandProjetDetail (Projet projet) { return projetRepository.save (projet); } → ajout

=> Controller:

@PostMapping ("/ajout-projet-et-projet-detail")
public projet addProjet and ProjetDetail (@Request Body projet p) { projet projet = projetService.
addProjetandProjetDetail (p);
return Projet ; }.

=> Entity (Parent (Projet))
@OneToOne (Cascade = CascadeType.All)
private ProjetDetail projetdetail.

=> Entity (fels (projetDetail))
@OneToOne (mapped By = "ProjetDetail")
@ToString. Exclude
@JsonIgnore
private Projet projet ;

(4

to add repositories we gonna need to add this line over here <nameoftheclass,type of the id >
public interface RendezVousRepository extends JpaRepository <RendezVous,Long> {

Now for service (interface for faible couplage)
for serviceExamenImpl
package tn.esprit.medecin.service;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;
import tn.esprit.medecin.entity.Clinique;
import tn.esprit.medecin.repository.CliniqueRepository;
import tn.esprit.medecin.repository.MedecinRepository;
import tn.esprit.medecin.repository.PatientRepository;
import tn.esprit.medecin.repository.RendezVousRepository;

@Service
public class ServiceExamenImpl implements ServiceExamen {

```
@Autowired
 private CliniqueRepository cliniqueRepository;
 @Autowired
 private MedecinRepository medecinRepository;
 @Autowired
 private PatientRepository patientRepository;
 @Autowired
 private RendezVousRepository rendezVousRepository;
 @Override
 public Clinique addClinique(Clinique clinique) {
 return cliniqueRepository.save(clinique);(.save to add)
 }
}
```

For affectaion now w e can and affect to it at the same time (add a doctor and effecte to clinique) for case Many to Many

```
@Override
public Medecin addMedecinAndAssignToClinique(Medecin medecin, Long IdClinique) {
   //awel lawej par id clinique kn medecin mawjouda wala le
Clinique c = cliniqueRepository.findById(IdClinique).orElse(null);
   List<Medecin> list = new ArrayList<>();
   list.add(medecin);
  ://in case mafamesh nzidouh
 if (c.getMedecins() == null) {
       c.setMedecins(list);
   } else {
       c.getMedecins().add(medecin);
   }
   return medecinRepository.save(medecin);
}}
```

For now case One to Many (rendezvous is the parent)

```
@Override
public RendezVous addRDVandAssignMedAndPatient(RendezVous rendezVous, Long idMedecin, Long
idPatient) {
```

```
 Medecin m = medecinRepository.findById(idMedecin).orElse(null);
 Patient p = patientRepository.findById(idPatient).orElse(null);
 rendezVous.setPatient(p);
 rendezVous.setMed(m);
 return rendezVousRepository.save(rendezVous);
}
```

For schelure :

AOP :
besh nkasmou l parte logique de code

Affection cas 2:

=> Récuperer le projet Detail et un projet de base de donnée et affecter le projet Detail (fils) au parent (projet).

=> Service

```
public void affecterPandPD (Long idP, Long idPD) {
projet projet = projetRepository.findById(projetId).get()
projet.setProjetDetail(projetDetail); //on set fils dans parent
ProjetRepository.save(projet);
```

=> Controller

```
@PutMapping("/affecter-projet-a-projet-detail
/{pdjet-id}/{projet-detail-id}
public void affecterPandPD(@PathVariable
("Projet-id") Long idP, @PathVariable("Projet-detai
l-id") Long idPD) {
projetService.affeterPandPD(idP, idPD); }
```

=> ManyToMany (Case)

=> Récuperer de la base de données (Fetch) l'equipe et le projet puis affecter le projet (fil) et l'equipe (parent), puis faire une mise à jour

- Service:

```
public void assignProjetToEquipe (Long idP,
Long idE) {
projet projet = projetRepository.findById
(idP).get();
Equipe equipe = equipeRepository.findById
(idE).get();
equipe.getProjet().add(projet);
equipeRepository.save(equipe)
```

Entity Parent

```
@ManyToMany (cascade = CascadeType.ALL, fetch = FetchType.EAGER)
private List<Projet> projet = new ArrayList
<Projet>();
```

=> Controller (Basically The same of Affection2)

Cas 4: Affectation: Ajouter un projet et affect un projet detail à ce projet

Service:

```
public projet esm fonction (projet projet, idPD){
  projet Detail projetdetail = projetDetailRepository
.findbyId (idPD).get();
  projet.setProjetDetail (projetDetail);
  return projet Repository.save(projU).
```

Cas 5: Désaffecter :

```
public Projet Desaffecter ProjetDetail from
Projet (Long idP) {
Projet projet = projetRepository.findbyId(idP).
  get();
    projet.set ProjetDetail (null).
     return projetRepository.save(projU) }
```

=> Controller

```
@Put Mapping ("/desaffection-projU-affecter-
projet / {projet-id}")
  public void Desaffecter ProjetDetail from Projet(
@Path Variable ("projet-id") Long idP)
    {
projetservice.DesaffecterProjetProjet from Projet
(idP)} }
```

Service

```
public void desaffetionProjet fromEquipe (Long idP,
Long idE) {
Projet projU = projetRepository.findbyId (idP).get
  ();
equipe equipe = equipe Repository.findbyid
(idE).get();
  //affecter le fils dans parent.
equipe.get Projet.remove (projet);
equipe Repository.Save (equipe)
```

In Entity (add the Fetch function in the parent) (6

=> to make all the affectation and desaffetion work you have to add in Service Imp @Service @AllArgCons

JPQL : Key words :

. Afficher la liste des Chambres qui ont un type donné. (double par exemple)

@Repository

public interface ChambreRepository extends JpaRepository <Chambre, Long> {

List<Chambre> findAllByTypeChambre (TypeChambre tc); }

=> JPQL : Select :

récuperer les entreprises d'aprés une @ donné.

in Repository file :

@Query ("select e from Enthoprise e where e.adresse := adresse").

List < Entreprise > retrieveEnthepnise ByAdresse (@Param ("adresse") String adresse);

=> récuperer les entreprises qui ont une équipe avec une spécialité

• JPQL

@Query ("Select entreprise From Entreprise entreprise, Equipe equipe where entreprise.id = equipe.entreprise.idand equipe.specialite = specialite")

List < Entreprise > retrieve Enthepnise (@Param ("specialite") String specialite).

• Native Query. (SQL et non JPQL)

@Query (value = "Select * From T_entreprise INNER JOIN T_Equipe equipe on entreprise.entreprise.id = equipe.Entreprise_Enthepnise_id where equipe.Equipe_Specialite := specialite", nativeQuery = true)

List < Entreprise > retrieve Entreprise (@Param ("specialite") String speciality).

JPQL : SELECT

Ces méthodes permettent de récupérer les entreprises qui ont une équipe avec une spécialité donnée :

JPQL :

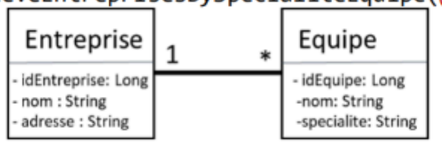
```
@Query("SELECT entreprise FROM Entreprise entreprise , Equipe equipe where
entreprise.id = equipe.entreprise.id and equipe.specialite =:specialite")

List<Entreprise> retrieveEntreprisesBySpecialiteEquipe(@Param("specialite")
String specialite);
```

Native Query (SQL et non JPQL) :

```
@Query(value = "SELECT * FROM T_ENTREPRISE entreprise INNER JOIN T_EQUIPE equipe
ON entreprise.ENTREPRISE_ID = equipe.ENTREPRISE_ENTREPRISE_ID where
equipe.EQUIPE_SPECIALITE =:specialite", nativeQuery = true)

List<Entreprise> retrieveEntreprisesBySpecialiteEquipe(@Param("specialite")
String specialite);
```



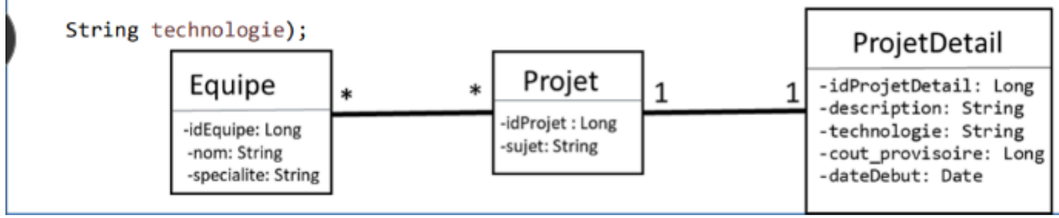
@Query("SELECT entreprise FROM Entreprise entreprise , Equipe equipe where entreprise.id = equipe.entreprise.id and equipe.specialite =:specialite")
List<Entreprise> retrieveEntreprisesBySpecialiteEquipe(@Param("specialite")
String specialite);

Native Query (SQL et non JPQL) :
@Query(value = "SELECT * FROM T_ENTREPRISE entreprise INNER JOIN T_EQUIPE equipe ON entreprise.ENTREPRISE_ID = equipe.ENTREPRISE_ENTREPRISE_ID where equipe.EQUIPE_SPECIALITE =:specialite", nativeQuery = true)
List<Entreprise> retrieveEntreprisesBySpecialiteEquipe(@Param("specialite")
String specialite);

JPQL : SELECT

- Cette méthode permet d'afficher les équipes qui travaillent sur une technologie donnée dont le projet n'a pas encore commencé.
- **JPQL :**

```
@Query("SELECT equipe FROM Equipe equipe"
        + " INNER JOIN equipe.projets projet"
        + " INNER JOIN Chambre detail"
        + " ON detail.idChambre = projet.projetDetail.idChambre"
        + " where detail.dateDebut > current_date"
        + " and detail.technologie =:technologie")
List<Equipe> retrieveEquipesByProjetTechnologie(@Param("technologie")
String technologie);
```



@Query("SELECT equipe FROM Equipe equipe"
+ " INNER JOIN equipe.projets projet"
+ " INNER JOIN Chambre detail"
+ " ON detail.idChambre = projet.projetDetail.idChambre"
+ " where detail.dateDebut > current_date"
+ " and detail.technologie =:technologie")
List<Equipe> retrieveEquipesByProjetTechnologie(@Param("technologie")
String technologie);

JPQL : UPDATE

Si nous souhaitons faire un **UPDATE, DELETE et INSERT**, nous devons ajouter l'annotaion **@Modifying** pour activer la modification de la base de données.

Cette méthode permet de mettre à jour l'adresse de l'entreprise.

JPQL :

```
@Modifying
@Query("update Entreprise e set e.adresse = :adresse where e.idEntreprise =
:idEntreprise")
int updateEntrepriseByAdresse(@Param("adresse") String adresse,
@Param("idEntreprise")
Long idEntreprise);
```

@Modifying
@Query("update Entreprise e set e.adresse = :adresse where e.idEntreprise = :idEntreprise")
int updateEntrepriseByAdresse(@Param("adresse") String adresse,
@Param("idEntreprise")
Long idEntreprise);

JPQL : DELETE

Cette méthode permet de supprimer les entreprises qui ont une adresse donnée :

JPQL :

```
@Modifying
@Query("DELETE FROM Entreprise e WHERE e.adresse= :adresse")
int deleteEntreprisebyadresse(@Param("adresse") String adresse);
```

C'est équivalent à :

```
@Modifying
@Query("DELETE FROM Entreprise e WHERE e.adresse= ?1")
int deleteFournisseurByCategorieFournisseur(String adresse);
```

@Modifying
@Query("update Entreprise e set e.adresse = :adresse where e.idEntreprise = :idEntreprise")
int updateEntrepriseByAdresse(@Param("adresse") String adresse,
@Param("idEntreprise")
Long idEntreprise);

JPQL : INSERT

- Cette méthode permet d'insérer des projets dans la table T_Projet:
- **JPQL :** Nous utilisons Spring Data JPA. Or INSERT ne fait pas partie des spécifications JPA. Donc, nous sommes obligés d'utiliser les Natives Query pour le INSERT.
- **Pas de JPQL pour les requêtes INSERT.**
- **Native Query (SQL et non JPQL) :**

```
@Modifying
@Query(value = "INSERT INTO T_Projet(projet_sujet) VALUES (:projetsujet)",
  nativeQuery = true)
void insertProjet(@Param("projetsujet") String projetsujet);
```

Cette méthode permet d’insérer des projets dans la table T_Projet:
• JPQL : Nous utilisons Spring Data JPA. Or INSERT ne fait pas partie des spécifications JPA. Donc, nous sommes obligés d’utiliser les Natives Query pour le INSERT.
• Pas de JPQL pour les requêtes INSERT.
• Native Query (SQL et non JPQL) :
@Modifying
@Query(value = "INSERT INTO T_Projet(projet_sujet) VALUES (:projetsujet)",
nativeQuery = true)
void insertProjet(@Param("projetsujet") String projetsujet);

Scheduler

Introduction

- La planification(scheduling) consiste à exécuter les tâches pendant une période de temps spécifique.
- Spring Boot Scheduling est une fonctionnalité pratique qui nous permet de planifi des tâches dans nos applications Spring Boot.
- Par exemple, si vous voulez que votre application exécute une tâche après un intervalle fixe ou en fonction d'un calendrier.

- **Le scheduler fait partie du module Core du framework Spring (Pas de dépendances à ajouter dans le pom.xml).**

@Scheduled Annotation

- Spring Boot utilise l'annotation **@Scheduled** pour la planification des tâches.
- Il faut respecter certaines règles lors de l'utilisation de cette annotation : Les méthodes doivent être sans paramètre.
- Pour activer la planification (scheduling), il faut ajouter l'annotation **@EnableScheduling** à la classe main.

```
@EnableScheduling
@SpringBootApplication
public class TpFoyerApplication {
    public static void main(String[] args) {
        SpringApplication.run(TpFoyerApplication.class, args);
    }
}
```

@Scheduled Annotation

- Spring Boot permet de créer facilement une tâche de planification(scheduling task).

➢ Il suffit d'annoter la méthode avec l'annotation **@Scheduled**.

- L'annotation **@Scheduled** définit la planification (par exemple, quand la méthode sera exécutée, etc.).

- Nous pouvons passer certains paramètres à l'annotation pour personnaliser le comportement.

Fixed Rate

Pour planifier un déclenchement de méthode à une date fixe, nous pouvons utiliser le paramètre **fixedRate** dans l'annotation **@Scheduled.**

Fixed Rate : permet à Spring d'exécuter la tâche à des intervalles périodiques, même si la dernière exécution est en cours.


```
@Scheduled(fixedRate = 60000)
public void fixedRateMethod() {
system.out.println("Method with fixed Rate");
}
```

Fixed Delay

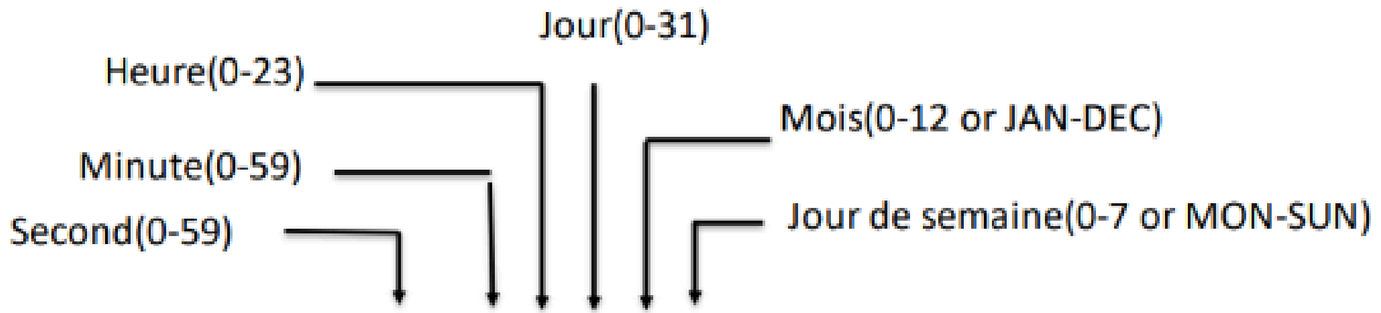
- Pour fixer un délai fixe entre la dernière exécution et le début de l'exécution suivante, nous pouvons utiliser le paramètre **fixedDelay**.
- Ce paramètre compte le délai après l'exécution de la dernière invocation.

Les tâches sont déclenchées avec un retard de 60 secondes.

```
@Scheduled(fixedDelay = 60000)
public void fixedDelayMethod() {
system.out.println("Method with fixed delay");
}
```

Cron expression

- L'expression **Cron** est une façon flexible et puissante pour planifier les tâches.




```
@Scheduled(cron = "*/60 * * * * *" )
public void cronMethod() {
system.out.println("Method with cron expression");
}
```

Affichage avec @Slf4j (Lombok) sur la console

```
@Service
@AllArgsConstructor
@Slf4j
public class ChambreServiceImpl implements IChambreService {

    ChambreRepository chambreRepository;

    @Scheduled(fixedDelay = 10000) // 10000 millisecondes
    public List<Chambre> retrieveAllChambres() {
        List<Chambre> listC = chambreRepository.findAll();
        for (Chambre c: listC) {
            log.info("Chambre :" + c);
        }
        return listC;
    }
```

```
@Service
@AllArgsConstructor
@Slf4j
public class ChambreServiceImpl implementsIChambreService {
 ChambreRepository chambreRepository;
 @Scheduled(fixedDelay = 10000) // 10000 millisecondes
 public List<Chambre> retrieveAllChambres() {
 List<Chambre> listC = chambreRepository.findAll();
 for (Chambre c: listC) {
 log.info("Chambre :" + c);
 }
 return listC;
 }
```