Γιατί το χρησιμοποιούμε;

Το **JPA (Java Persistence API)** και ειδικότερα το **ORM (Object-Relational Mapping)** αποτελούν εργαλειοθήκες που εφαρμόζονται για να διευκολύνουν την αλληλεπίδραση μεταξύ **αντικειμενοστραφούς προγραμματισμού** και **σχεσιακών βάσεων δεδομένων**. Πιο συγκεκριμένα, επιτρέπουν την άμεση **αντιστοίχιση αντικειμένων Java σε πίνακες μιας βάσης δεδομένων**.

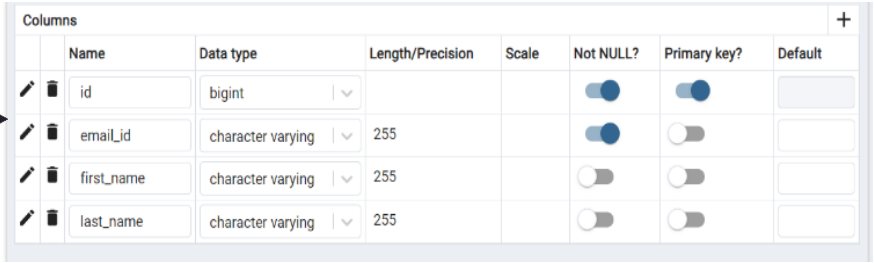
Η χρήση τους μας προσφέρει:

1. Απλοποίηση της αλληλεπίδρασης με τη βάση δεδομένων.
2. Αποφυγή υπερβολικού επαναλαμβανόμενου κώδικα.
3. Ευκολότερες αλλαγές, συντήρηση και ανάπτυξη του κώδικα.
4. Ανεξαρτησία του κώδικα από τη βάση δεδομένων.
5. Επιτάχυνση διαδικασιών μεταφοράς δεδομένων με την βάση.

Πως γίνεται η αντιστοίχιση Object-Oriented εννοιών σε Βάση Δεδομένων;

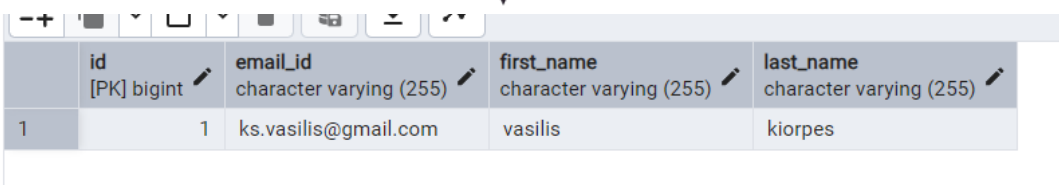
Η κάθε κλάση αντιστοιχείται σε ένα νέο πίνακα στην βάση δεδομένων. Η κάθε παράμετρος της κλάσεως αποτελεί στήλη στον αντίστοιχο πίνακα στην βάση δεδομένων.

```java
public class Employee {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    @Column(name = "first_name")
    private String firstName;

    @Column(name = "last_name")
    private String lastName;

    @Column(name = "email_id", nullable = false, unique = true)
    private String email;
}
```



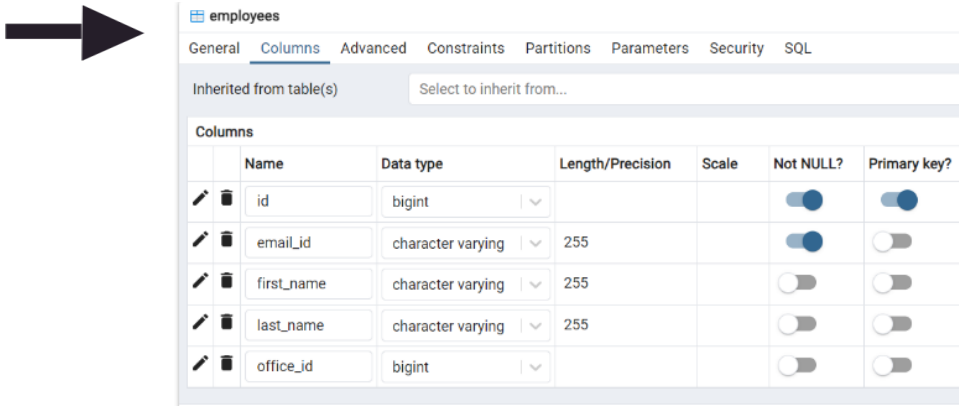
Κάθε αντικείμενο της κλάσεως μπορεί να σωθεί στην βάση δεδομένων ως μια σειρά στον πίνακα αυτό.

```java
Employee myEmployee = new Employee( id: 1L, firstName: "vasilis", lastName: "kiorpes", email: "ks.vasilis@gmail.com");
employeeRepository.save(myEmployee);
```



	id [PK] bigint	email_id character varying (255)	first_name character varying (255)	last_name character varying (255)
1	1	ks.vasilis@gmail.com	vasilis	kiorpes

Οι



συσχετίσεις από πλευρά Object Oriented προγράμματος υλοποιούνται μέσω περιεκτικότητας. Παράλληλα, στη βάση δεδομένων η σύνδεση εκτελείται με ξένο κλειδί που επιλέγεται με κατάλληλα annotation. Θα δούμε την κάθε περίπτωση πιο αναλυτικά στην συνέχεια.

```java
@Table(name = "employees")
public class Employee {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    @Column(name = "first_name")
    private String firstName;

    @Column(name = "last_name")
    private String lastName;

    @Column(name = "email_id", nullable = false, unique = true)
    private String email;

    @ManyToOne
    @JoinColumn(name = "office_id")
    private Office office;
}
```

Κατά τις συσχετίσεις των πινάκων, υπάρχουν 4 Annotation που θα είναι ιδιαίτερα χρήσιμα, αν και δεν δηλώνουν την ίδια τη σχέση. Αυτά είναι:

JoinColumn: Χρησιμοποιείται για να σηματοδοτήσει ένα ξένο κλειδί σε έναν πίνακα. Πρέπει να του δοθεί ένα πεδίο της προς-συσχέτισης οντότητας. Η στήλη εκείνη θα αποτελέσει τη στήλη στην οποία θα δείχνει το ξένο κλειδί.

MappedBy: Χρησιμοποιείται σε πλευρές συσχετίσεων που δεν περιέχουν ξένο κλειδί για να ζητήσουμε από το

```java
@Table(name = "offices")
public class Office {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    @Column(name = "location")
    private String location;

    @Column(name = "email_id")
    private String email;

}
```

πρόγραμμα να ετοιμάσει (για δική μας ευκολία) μια συλλογή από τις σχετιζόμενες οντότητες του.

Fetch: Ορίζει με ποια στρατηγική θα φορτωθούν οι σχετιζόμενες οντότητες από την βάση δεδομένων. Οι κυριότερες μορφές είναι το @FetchType.LAZY (οι σχετιζόμενες οντότητες να φορτωθούν μόνο την στιγμή που θα τις χρειαστούμε) και @FetchType.EAGER (οι σχετιζόμενες οντότητες φορτώνονται αμέσως, μαζί με την αρχική)

Cascade: Ορίζει ποιες αλλαγές σε μία οντότητα θα επηρεάζουν τις συσχετίσεις της. Με το CascadeType.REMOVE π.χ. δηλώνει πως αν σβήσουμε ένα office, σβήνουμε και όλους τους employee που δουλεύουν σε αυτό.

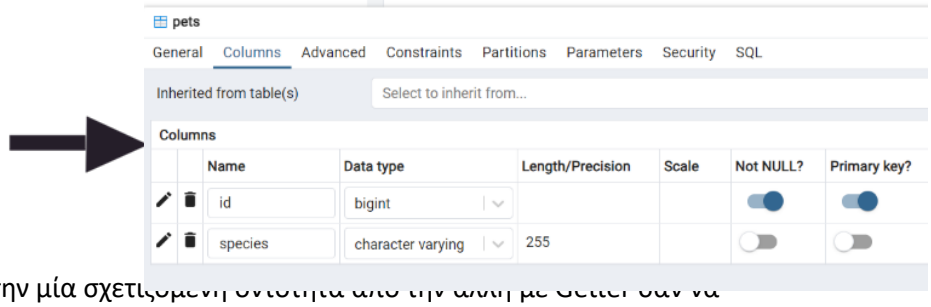
Αντιστοίχιση 1-1 Σχέσεων (@OneToOne)

Αντιστοίχιση 1-1 εφαρμόζεται όταν κάθε σειρά του ενός πίνακα συσχετίζεται με ακριβώς ένα στοιχείο του αλλουνού. Η υλοποίηση συνήθως περιλαμβάνει την χρήση ενός ξένου κλειδιού σε έναν από τους δύο πίνακες.

Ας πούμε π.χ. πως θέλουμε έναν πίνακα για κατοικίδια και έναν πίνακα για κολάρα που περιέχουν γραπτές πληροφορίες. Κάθε κατοικίδιο μπορεί να φοράει ένα μόνο κολάρο, και κάθε κολάρο μπορεί να φοριέται από ένα μόνο ζωάκι.

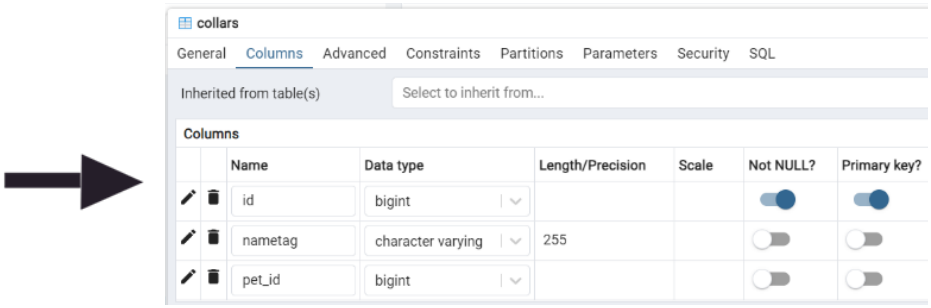
Όπως βλέπουμε, η σχέση εμφανίζεται και στις δύο οντότητες από την πλευρά του κώδικα.

```java
@Table(name = "pets")
public class Pet {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    @Column(name = "species")
    private String species;

    @OneToOne(mappedBy = "pet")
    private Collar collar;
}
```



Μπορούμε να ανακτήσουμε την μία σχετιζόμενη οντότητα από την άλλη με Getter σαν να

```java
@Entity
@Table(name = "collars")
public class Collar {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    @Column
    private String nametag;

    @OneToOne
    @JoinColumn(name = "pet_id")
    private Pet pet;
}
```



ήταν οποιοδήποτε άλλο δεδομένο.

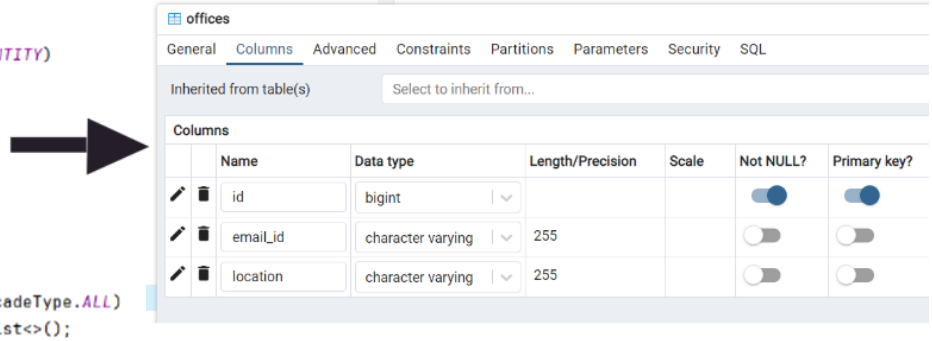
Στην βάση δεδομένων όμως παρουσιάζεται μόνο στον πίνακα που εφαρμόζει @JoinColumn με το όνομα που του έχουμε ζητήσει. Επιλέγει αυτόματα την στήλη Id για να προετοιμάσει το ξένο κλειδί.

Αντιστοίχιση 1-n Σχέσεων (@OneToMany, @ManyToOne)

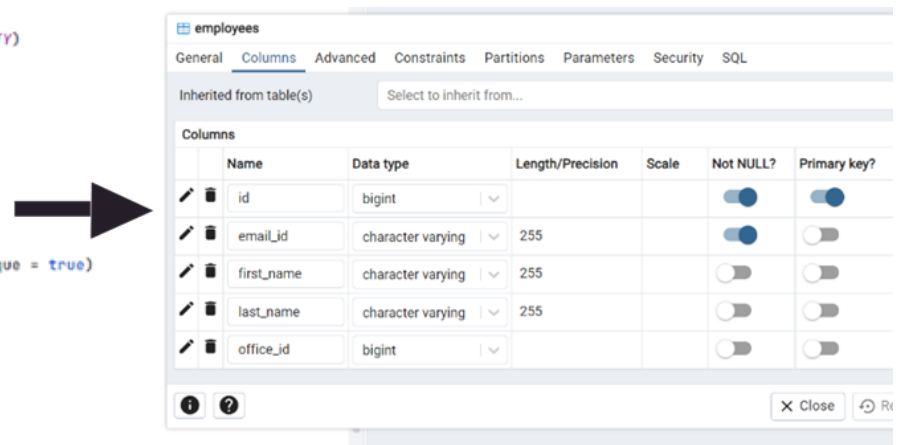
Η σχέση 1-n χαρακτηρίζεται από την μη συμμετρική δομή της. Ένα αντικείμενο της κλάσης ‘α’ συσχετίζεται με πολλαπλά αντικείμενα της κλάσης ‘β’. Το κάθε αντικείμενο της κλάσης ‘β’ όμως συσχετίζεται με μόνο ένα αντικείμενο ‘α’.

Με σωστά annotation είναι δυνατόν (και μάλιστα, ιδιαίτερα εύκολο) να ανακτήσουμε όλα τα σχετιζόμενα αντικείμενα ‘β’ για το κάθε αντικείμενο ‘α’ μέσω μιας λίστας που δεν έχει άμεσο αντίκτυπο στη σειρά ‘α’ στην βάση δεδομένων. Υπάρχει μόνο στον κώδικα, για διευκόλυνση δική μας.

```java
@Table(name = "offices")
public class Office {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    @Column(name = "location")
    private String location;

    @Column(name = "email_id")
    private String email;

    @OneToMany(mappedBy = "office", cascade = CascadeType.ALL)
    private List<Employee> employees = new ArrayList<>();

}
```




```java
@Table(name = "employees")
public class Employee {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    @Column(name = "first_name")
    private String firstName;

    @Column(name = "last_name")
    private String lastName;

    @Column(name = "email_id", nullable = false, unique = true)
    private String email;

    @ManyToOne
    @JoinColumn(name = "office_id")
    private Office office;
}
```



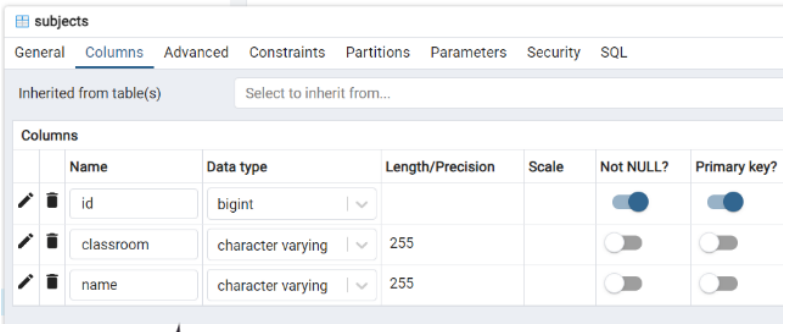
Αντιστοίχιση n-n Σχέσεων (@ManyToMany)

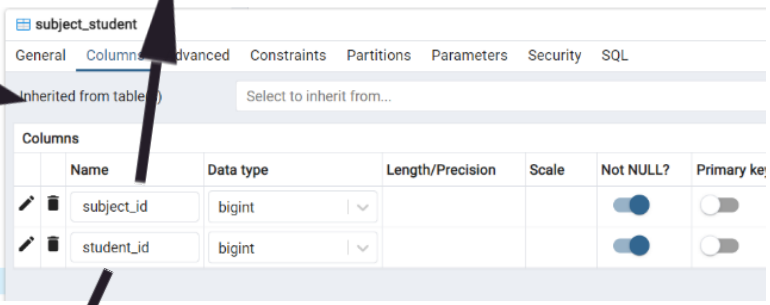
Όπως γνωρίζουμε, η σχέση n-n σε μία βάση δεδομένων υλοποιείται με την χρήση ενός πίνακα-συσχέτιση η οποία συγκρατεί δύο ξένα κλειδιά: ένα για κάθε πίνακα που συνδέει. Κάθε σειρά της αποτελεί συσχέτιση μεταξύ δυο σειρών των γειτονικών της πινάκων.

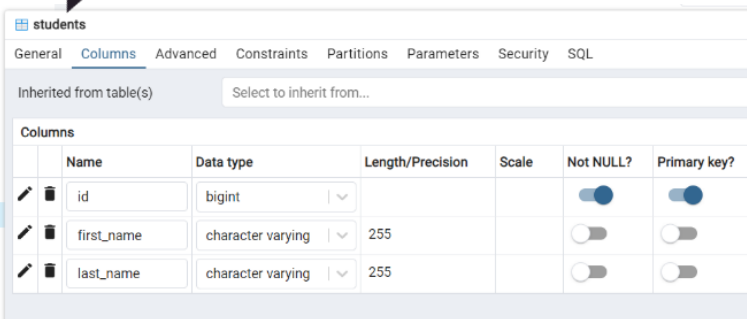
Η υλοποίηση αυτή είναι σχετικά εύκολη με το JPA. Αντί για @JoinColumn, χρησιμοποιούμε το @JoinTable, μέσα στο οποίο δηλώνονται δύο πεδία JoinColumn.

```java
@Table(name = "subjects")
public class Subject {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    @Column
    private String name;

    @Column
    private String classroom;

    @ManyToMany
    @JoinTable(
            name = "subject_student",
            joinColumns = @JoinColumn(name = "subject_id"),
            inverseJoinColumns = @JoinColumn(name = "student_id")
    )
    private List<Student> students = new ArrayList<>();
}
```

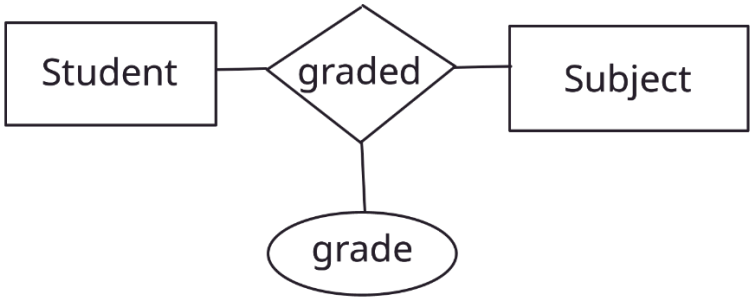
```java
@Table(name = "students")
public class Student {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    @Column(name = "first_name")
    private String firstName;

    @Column(name = "last_name")
    private String lastName;

    @ManyToMany(mappedBy = "students")
    private List<Subject> subjects = new ArrayList<>();
}
```





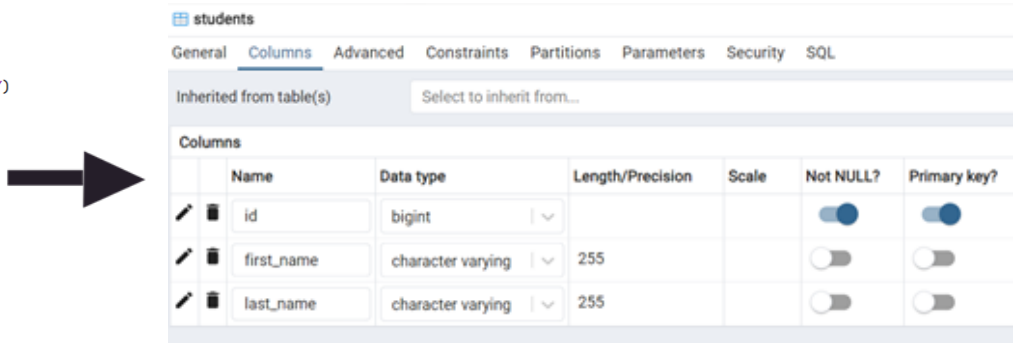


Υπάρχει όμως ένα πρόβλημα. Αν και ιδιαίτερα εύκολη, η υλοποίηση αυτή δεν μας επιτρέπει να προσθέσουμε επιπλέον στήλες στον πίνακα-συσχέτιση. Πράγματι, το JPA δεν περιέχει ανώδυνο τρόπο να το πραγματοποιήσουμε με τέτοιο τρόπο. Ας πούμε όμως πως θέλουμε να βαθμολογήσουμε τους μαθητές ανά μάθημα:

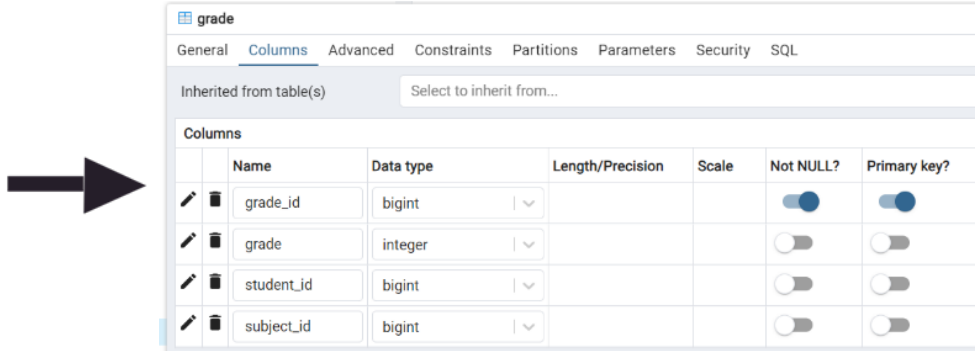


Αν θελήσουμε να εκτελέσουμε την σύνδεση με τέτοιο τρόπο, θα χρειαστούμε μια πιο περίπλοκη υλοποίηση. Προτείνεται να δημιουργήσουμε μία νέα οντότητα που παίρνει τον ρόλο του συνδετήρα, και περιέχει δύο σχέσεις τύπου 1-n με τους γείτονες που συνδέει. Η τεχνική αυτή θα χωρίσει δηλαδή την συσχέτιση n-n σε δύο συσχετίσεις 1-n.

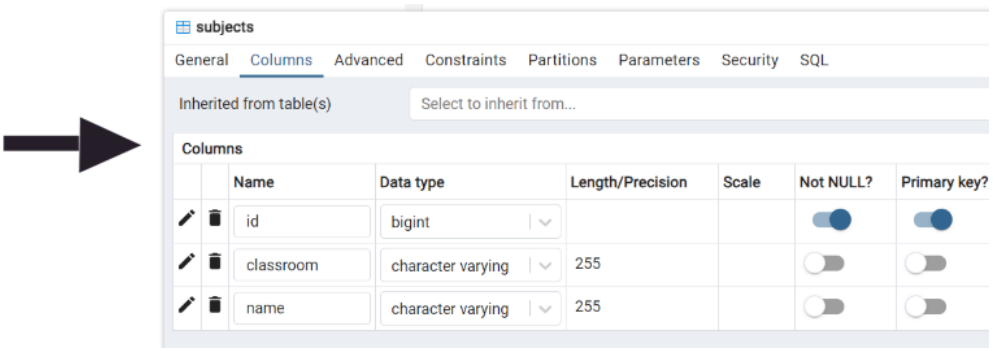
```java
@Table(name = "students")
public class Student {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    @Column(name = "first_name")
    private String firstName;

    @Column(name = "last_name")
    private String lastName;

    @OneToMany(mappedBy = "student")
    private Set<Grade> grades;
}
```




```java
@Table(name = "grade")
public class Grade {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    @Column(name = "grade_id")
    private Long gradeId;

    @ManyToOne
    @JoinColumn(name = "student_id")
    private Student student;

    @ManyToOne
    @JoinColumn(name = "subject_id")
    private Subject subject;

    @Column
    private int grade;
}
```




```java
@Entity
@Table(name = "subjects")
public class Subject {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    @Column(unique = true)
    private String name;

    @Column
    private String classroom;

    @OneToMany(mappedBy = "subject")
    private Set<Grade> grades;
}
```



Χρειάζεται προσοχή με οποιαδήποτε από τις δύο υλοποιήσεις. Προτείνεται να εφαρμόσουμε Lazyloading όπου είναι δυνατό. Είναι ιδιαίτερα εύκολο να συναντήσουμε το πρόβλημα n+1 (το οποίο θα αναλύσουμε αμέσως μετά) και να αυξήσουμε σφοδρά τον φόρτο εργασίας της σύνδεσης με την βάση.

Τι είναι το πρόβλημα “Select N+1”;

Το πρόβλημα N+1 προκύπτει όταν μια οντότητα η οποία περιέχει άλλες οντότητες ανακτά τα περιεχόμενα της, και έπειτα ξεκινά επιπλέον διαδικασίες πάνω σε κάθε μία από τις οντότητες της. Αυτό οδηγεί σε σοβαρή αύξηση του φόρτου εργασιών (ειδικά για μεγάλες βάσεις δεδομένων) και μπορεί να προκαλέσει σημαντικές πτώσεις στην ταχύτητα εκτέλεσης.

Το όνομα προκύπτει από τις εντολές-query που οδηγούν στο πρόβλημα αυτό. Συγκεκριμένα:

1 Query για να επανακτηθούν όλα τα περιεχόμενα αντικείμενα (πλήθος N).

N Query για να εκτελεσθεί μια διεργασία πάνω σε κάθε ένα από τα περιεχόμενα αντικείμενα.

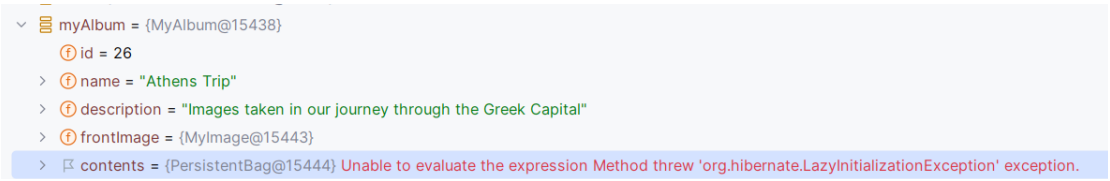
Για να το αντιμετωπίσουμε, είναι σημαντικό εφαρμόζουμε κυρίως Lazy load, ώστε να εκτελούμε δευτερεύουσες αναζητήσεις μόνο σε περιεχόμενα αντικείμενα τα οποία όντως θα χρειαστούμε. Το Eager load το χρησιμοποιούμε μόνο σε περιπτώσεις όπου, εάν φορτώνουμε ένα αντικείμενο, είμαστε σίγουροι πως θα χρειαστεί οπωσδήποτε να εκτελέσουμε και κάποια διεργασία πάνω του.

Π.χ. θέλουμε να φορτώσουμε ένα άλμπουμ. Γνωρίζουμε πως την πρόσοψη (cover Image) θα τη χρειαστούμε σε κάθε εφαρμογή που χρησιμοποιείται το Album, οπότε εφαρμόζουμε Eager Loading. Οι περιεχόμενες εικόνες (contents) του άλμπουμ όμως, θα λειτουργούν με Lazy Loading διότι δεν θα τις χρησιμοποιούμε σε κάθε εφαρμογή.

```java
@Setter  34 usages
@Getter
@NoArgsConstructor
@AllArgsConstructor
@Entity
@Table(name = "albums")
public class MyAlbum {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private long id;
    @Column(nullable = true)
    private String name;
    @Column
    private String description;

    @ManyToOne(fetch = FetchType.EAGER)
    @JoinColumn(name = "imageId")
    private MyImage coverImage;

    @ManyToMany(fetch = FetchType.LAZY)
    @JoinTable(
            name = "contains",
            joinColumns = @JoinColumn(name = "album_Id"),
            inverseJoinColumns = @JoinColumn(name = "image_Id")
    )
    private List<MyImage> contents = new ArrayList<>();
```

Τι είναι το σφάλμα Lazyloading και πως αντιμετωπίζεται;

Το σφάλμα Lazyloading, γνωστό για το Lazy Initialization Exception που προκαλεί, προκύπτει όταν προσπαθήσουμε να τραβήξουμε δεδομένα από μία συσχέτιση η οποία εφαρμόζει Lazy Load στο ανάλογο πεδίο και δεν έχει αποκτήσει τα δεδομένα αυτά.



Εφόσον δεν θέλουμε να εφαρμόσουμε EagerLoading λόγο του N+1 προβλήματος, θα χρειαστεί να πραγματοποιήσουμε 'χειροκίνητα' στον κώδικα μας την φόρτωση των δεδομένων υπό καθεστώς Lazyloading πριν τα χρησιμοποιήσουμε. Ο πιο εύκολος τρόπος επίλυσης του ζητήματος είναι μια @Transactional συνάρτηση.

Μία συνάρτηση με annotation @Transactional έχει την ιδιότητα να συγκρατεί ανοικτή οποιαδήποτε σύνοδο ανοίξει μέσα της, έως ότου ολοκληρωθεί η ίδια. Αυτό την θέτει ικανή να ζητάει τα δεδομένα για οποία ισχύει Lazyloading πολιτική. Αρκεί να καλέσουμε τα δεδομένα που μας ενδιαφέρουν μέσα σε αυτήν. Η σύνοδος απόκτησης δεδομένων δεν θα έχει κλείσει ακόμη, οπότε θα είναι ικανή να τα επιστρέψει!

Βεβαίως, θα χρειάζεται και μια αντίστοιχη, μη-Transactional συνάρτηση για κάθε τέτοια μέθοδο μέσα στην κλάση Service της οντότητας αυτής. Αλλιώς θα πέσουμε πάλι στο πρόβλημα n+1 αφού κάθε αντικείμενο θα ανακτάται με συνάρτηση Transactional με πλήρες ανάκτηση.

```java
@Override  no usages
public MyAlbum getAlbum(Long AlbumId) {
    return repository.findById(AlbumId).orElseThrow();
}

@Override  2 usages
@Transactional
public MyAlbum getFullAlbum(Long albumId) {
    MyAlbum myFullAlbum = repository.findById(albumId).orElseThrow();
    myFullAlbum.getContents().size();
    return myFullAlbum;
}
```

Βασίλης Κιορπές
ΑΕΜ 22038