2018
May
01
TUESDAY

Spring JDBC Template ⇒ obsolete
we need to write explicit queries

with JPA (Java Persistence API) → no need to write explicit queries
(except for certain situations).
→ provides built-in methods for CRUD

① JPQL ⇒ Write (explicit) queries using Entity names.
(class/model
NOT table name)

example:

@NamedQuery (name = "query_get_all_.." , query = "
Select c From Course c")

Annotation

② → Another option is to use "Criteria Query" to build entire query using java API

③ Last option is to use "Native Query" directly

ex: @Autowired
EntityManager em;

=

Query query = em.CreateNativeQuery ("SELECT * from courses",
Course.class);

Digital Transformation

GCP

roles - P

Terraform
Ansible splunk
rest

TCS

big Client

rate ⇒ contractual basis

@API Automation

Sheath ⇒

@API

Dev

Mobile Tester

APRIL
S M T W T F S S M T W T F S S M T W T F S S M T W T H S M T
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30

H2 (in-mem db) :

With latest versions of Spring Boot (2.3+), the H2 db name is randomly generated each time you restart the server.

You can find db name & URL from console log

Recommended :

Make db URL a constant by configuring this in application.properties/yml :

```
spring.datasource.url = jdbc:h2:mem:testdb
spring.data.jpa.repositories.bootstrap-mode = default
```

Add h2, jpa dependencies in pom.xml

Once application starts up, try localhost:8080/h2-console

(make sure you've got this in appln.yml: spring.h2.console.enabled = true)

↓ provided by h2 dependency.

↓ verify db name & other properties (driver, pwd etc)

Creating a table in H2

⇒ Define a "data.sql" file in classpath src/main/resources

↓ This gets called when you launch up the appln to initialize the db.

data.sql ↳

```
create table person
(
  id integer not null,   (as it's a primary key ⇒ not null)
  name varchar(255) not null,
  location varchar(255),
  birth_date timestamp,
  primary key(id)
);
```

Now when we start up the application & access /h2-console endpoint we can see a table "person" created

Insert Query : (Use this in /h2-console) to populate data

```
INSERT INTO PERSON(ID, NAME, LOCATION, BIRTH_DATE)
VALUES(10001, 'Adam', 'BANGALORE', sysdate());
```

→ Add this to data.sql
(insert multiple queries for more data)

Use @Repository Annotation to your dao/dto class

~~pixelcode~~
~~email-id~~

Hibernate is the most popular implementation of JPA
Hibernate implements JPA
JPA specifics/standardizes the ORM & Hibernate implements JPA

[JPA is just a specification, meaning there is no implementation]
JPA is the interface & Hibernate is the implementation.

WRONG

@Entity Annotation @ model class level

optional @Table(name = "tableName") // otherwise tablename = classname
public class person {
}
use this only when tablename & entity names are different

@Column (name = "columnName")
private String location; optional
If table column name is different from entity name

@Id → indicates primary key
@GeneratedValue → sequences & provide record ids.

& have a "No-args" constructor

3 week ~~days~~ 2026264032 2 ⇒ ref no
tracking ~~2-3 days~~

2018 May 04 FRIDAY

& have another constructor without the "id" (@Id) field & JPA takes care of initializing & providing values to the id field

Transactions → whenever we insert/delete rows, we do an update to the dB, transactions become very important.

Whenever you do 3-4 steps/updates in a single transaction, you'd want all of them to be successful or all of them to fail together
(Use @Transactional annotation - class-level)

ex:

```
@Repository
@Transactional
public class PersonJpaRepository {

    //Connect to dB
    @PersistenceContext
    EntityManager entityManager;

    public Person findById(int id) {
        return entityManager.find(Person.class, id);
    }
}
```

in appln.yml

spring.jpa.show-sql=true
to see generated queries

entityManager.merge(person);
↳ returns person
→ to update/~~delete~~ insert (based on id's)

entityManager.remove(person);
↳ returns void
→ to delete

JPQL → Java persistence query language
↳ to write explit queries

Usage

```
TypedQuery<Person> namedQuery = entityManager.createNamedQuery("find-all-persons", Person.class);
namedQuery.getResultList(); → List<Person> return type
```

& create a NamedQuery in the entity POJO/model class

```
@Entity
@NamedQuery(name="find-all-persons", query="select p from Person p")
public class Person{
}
```

queries using entity names rather than actual table names

delete operation

```
@Autowired
EntityManager entityMgr;
entityMgr.remove(id);
```

(delete operation or any DB changes)
→ should be done in something known as "transaction management"
↓
Use @Transactional annotation on class level

so if anything fails, the changes will be rolled back

ex: transfering bank amt from 1 acc to another

20225
4 days

2018 May 06 SUNDAY

To ensure unit tests don't alter dB related changes (example unit test for deleteById), ⇒ use @DirtiesContext (method level annotation)

annotation

[Spring reverts the dB changes due to unit tests & also other unit test don't fail due to subsequent UTCs altering the dB]

insert → entityManager.persist(Obj);
update → entityManager.merge(Obj);
findById → entityManager.find(id); (R U D)
delete → entityManager.remove(id);

entityManager.flush() → push/persist to dB
.detach(obj) → entity Mgr no longer tracks the obj changes & hence subsequent obj changes are no longer persisted to dB
.clear() → clears/detaches ALL objs from entityMgr
.refresh(obj) → resets the obj to its initial stage by ignoring subsequent changes on the obj

JPQL - Java Persistence Query Language

SQL → we query from dB tables
JPQL → we query from entities ⇒ Hibernate converts them into SQL queries

Ex:

SQL : select * from Course

JPQL : select c from Course c

Usage :

```
TypedQuery<Course> query = entityManager.createQuery(
        "Select c From Course c", Course.class);
List<Course> result = query.getResultList();
```

Timestamps → when row was created/updated

using Java 8, ⇒ "LocalDateTime"

In Entity class,

```
@UpdatedTimestamp
private LocalDateTime lastUpdatedDate;

@CreationTimestamp
private LocalDateTime createdDate;
```

In the JPQL example earlier, we've hardcoded the query. To avoid this we can use a "@NamedQuery" → query is referenced by a name

Inside Entity ⇒

```
@Entity
@NamedQuery(name="query_get_all_courses",
            query="Select c from Course c")
public class Course {
}
```

Usage ⇒ `entityManager.createNamedQuery("query_get_all_courses");` Course.class

@NamedQuery is NOT a repeatable annotation (to have multiple queries)

↓

Use @NamedQueries → for multiple named queries

Usage: 28-237

```
@NamedQueries (
    value = {
        @NamedQuery (name = "query-get-all", query = "select c from Course c"),
        @NamedQuery (name = "query_where", query = "sel... where...")
    })
```

Native Queries:

↳ Sending a direct SQL query out from JPA

Usage:

① `Query query = entityManager.createNativeQuery("select * from Course", (Course.class));`
(Native SQL query string)

② `Query query = entityManager.createNativeQuery("select * from Course where id = ?", Course.class);`
`query.setParameter(1, 10001L);`
(1 → position of placeholder (?); 10001L → value of placeholder)

Alternate ⇒ instead of using placeholders,
`"select * from Course where id = :id"`
`query.setParameter("id", 10001L);`

NOTE: In some situations, there's no other option but to fire a Native Query (ex: to use some dB specific feature not available in JPA)

Also for Mass updates (in JPA → get row & update row
Native → 1 query to update all)

Establishing relationships with JPA & Hibernate

- OneToOne

Course ↔ student

id	name

id	name

student ↔ Passport

id	name

id	number

One to One
Student can have only 1 passport & Passport can have only 1 student

Course ↔ Review

id	name

id	rating	description

Many to One
Course can have multiple reviews, Review is always associated with a single Course

Course & students ⇒
Many to Many
Course can have multiple Students
Students can have multiple Courses

One to One

Student

id	Name	passport_id
20001	Oscar	40001
20002	Adam	
20003	Jane	

Passport

id	number	student_id
40001	61235	20001
40002	NRE	
40003	L12V9	

2 options

① Student owns the relationship

② Passport owns the relationship

① Entity
Student

```
@OneToOne
private Passport passport;
```

update data.sql

```
insert into student(id, name, passport_id)
values(20001, 'Oscar', 40001);
```

H2-console

```
select * from student, passport
where
student.passport_id =
    passport.id ;
```

JOIN query

id	name	passport_id	id	number
20001	Oscar	40001	40001	61235

APRIL

S	M	T	W	T	F	S	S	M	T	W	T	F	S	S	M	T	W	T	F	S	S	M	T
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30

2018 May 10 THURSDAY

passport_id → foreign key for "STUDENT" table
↓
primary key for different table ("PASSPORT")

EAGER Vs LAZY fetch

By default, any OneToOne relationship is "eager-fetch"

For example, whenever we retrieve 'student' details, its associated 'passport' details are also fetched/retrieved & made ready.

This might give performance issues (to unnecessary fetch join details)

To avoid ⇒ Use "Lazy-fetch"

```
Student class
@OneToOne(fetch = FetchType.LAZY)
private Passport passport;
```

FetchType. By default ⇒ EAGER

One To One Mapping :
① Uni directional
② Bi directional

Uni-Directional

```
@Entity
Student {
@OneToOne
private Passport passport;
```

Bi-Directional

```
@Entity
Student {
@OneToOne
private Passport passport;
```

```
@Entity
Passport {
@OneToOne
private Student student;
```

if Student owns the relationship, then ⇒

```
@Entity
Passport {
@OneToOne(mappedBy = "passport")
private Student student
```

MAY

T W T F S S M T W T F S S M T W T F S S M T W T F S S M T W T
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31

- Many To One Mapping

ex: Relationship b/w "Course" & "Review" tables.

Course

id	Name	Review_id
10001	JPA	50001, 50002
10002	JPA 2	

Not good design

Review

id	Rating	desc
5001	5	Great
5002	4	Nice

Course Id
10001

This works

So review owns the relationship

```
class Review {
    @ManyToOne
    private Course course;
}
```

```
class Course {
    @OneToMany(mappedBy = "course")
    private List<Review> reviews = new ArrayList();
}
```

Update data.sql

```
insert into review (id, rating, description, course_id) values (5001, '5', 'Great', 10001);
```

H2 console : SELECT * FROM REVIEW, COURSE WHERE COURSE_ID = REVIEW.COURSE_ID

2018 May 12 SATURDAY

Many To Many Mappings

relationship btw "Course" & "Student" tables.

Course

id	name	student id
10001	JPA 50	20001, 20002
10002	Spring 50	
10003	SpringBoot 50	

Student

id	name	passport_id	Course_id
20001	Ranga	40001	10001, 10002
20002	Jane	40002	
20003	Adam	40003	

Not a good design

Hence this is where "JOIN" concepts comes in

we design a JOIN table,

COURSE_STUDENT

student_id	course_id
20001	10001
20002	10001

```
class Course {
    @ManyToMany
    private List<Student> students = new ArrayList<>();
}
```

```
class Student {
    @ManyToMany
    private List<Course> courses =
        new ArrayList<>();
}
```

When you run the appln,
go to H2 console
you can see 2 tables created ⇒
"COURSE_STUDENTS" & "STUDENT_COURSES"

To fix this,
⇒ make one of the entities own the relationship.

```
Course.class
@ManyToMany(mappedBy = "courses")
```

(Student owns it now)

Now there's only 1 table ⇒ "STUDENT_COURSES"

MAY

T	W	T	F	S	S	M	T	W	T	F	S	S	M	T	W	T	F	S	S	M	T	W	T
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31

Many To Many - Customizing the Join Table.

Change table Name,

in the owning side of entity (Student.class)

Use
```
@ManyToMany
@JoinTable(name = "STUDENT_COURSE")
private ---
```

Furthermore,.

```
@JoinTable(name = "STUDENT_COURSE", joinColumn=@JoinColumn(name="STUDENT_ID"),
inverseJoinColumn=@JoinColumn(name="Course_id"))
```

in data.sql

```
insert into student_course(student_id, course_id) values (20001, 10001);
```

in h2-console,

```
SELECT * FROM STUDENT_COURSE, STUDENT, COURSE
WHERE STUDENT_COURSE.STUDENT_ID = STUDENT.ID
AND STUDENT_COURSE.COURSE_ID = COURSE.ID
```

STUDENT_ID	COURSE_ID	ID	NAME	PASSPORT_ID	ID	CREATED_DATE	LAST_UPD_DATE	NAME
20001	10001	20001	Adam	40001	10001	2017-08-04 10:39:15.961	=	JPA in 50 steps