**Question 01:**

Consider a database schema with two entities: **Employee** and **Department**. The **Employee** entity has attributes such as **id** (primary key), **name**, **salary**, and a many-to-one relationship with the **Department** entity. The **Department** entity has attributes **id** (primary key) and **name**. Formulate a set of tasks using the Criteria API to perform the following operations:

1. Retrieve a list of employees whose salary is within a specified range.
2. Obtain a list of employees ordered by their salary in descending order.
3. Retrieve employees based on the name of their associated department.
4. Retrieve employees whose names match a given pattern.

Provide the corresponding answers with well-explained examples in Java using the Hibernate Criteria API, ensuring proper session management and exception handling.

Answer:

import javax.persistence.\*;

import java.util.List;

@Entity

@Table(name = "departments")

public class Department {

@Id

@GeneratedValue(strategy = GenerationType.IDENTITY)

@Column(name = "id")

private Long id;

@Column(name = "name")

private String name;

@OneToMany(mappedBy = "department")

private List<Employee> employees;

// Constructors, getters, and setters

}

import javax.persistence.\*;

@Entity

@Table(name = "employees")

public class Employee {

@Id

@GeneratedValue(strategy = GenerationType.IDENTITY)

@Column(name = "id")

private Long id;

@Column(name = "name")

private String name;

@Column(name = "salary")

private double salary;

@ManyToOne

@JoinColumn(name = "department\_id")

private Department department;

// Constructors, getters, and setters

}

import org.hibernate.Session;

import org.hibernate.SessionFactory;

import org.hibernate.Transaction;

import org.hibernate.cfg.Configuration;

import javax.persistence.criteria.\*;

import java.util.List;

public class CriteriaApiExample {

private static SessionFactory sessionFactory;

static {

try {

// Initialize Hibernate SessionFactory

Configuration configuration = new Configuration().configure();

sessionFactory = configuration.buildSessionFactory();

} catch (Exception e) {

e.printStackTrace();

}

}

public static void main(String[] args) {

try {

// First add some rows

// Task 1: Retrieve employees with a specified salary range

List<Employee> employeesInSalaryRange = getEmployeesInSalaryRange(50000, 80000);

System.out.println("Employees in salary range:");

printEmployees(employeesInSalaryRange);

// Task 2: Retrieve employees ordered by salary in descending order

List<Employee> employeesOrderBySalaryDesc = getEmployeesOrderBySalaryDesc();

System.out.println("\nEmployees ordered by salary in descending order:");

printEmployees(employeesOrderBySalaryDesc);

// Task 3: Retrieve employees based on the name of their associated department

List<Employee> employeesByDepartment = getEmployeesByDepartmentName("IT");

System.out.println("\nEmployees in the IT department:");

printEmployees(employeesByDepartment);

// Task 4: Retrieve employees whose names match a given pattern

List<Employee> employeesByNamePattern = getEmployeesByNamePattern("John");

System.out.println("\nEmployees with names matching the pattern 'John':");

printEmployees(employeesByNamePattern);

} finally {

// Close the sessionFactory to release resources

if (sessionFactory != null) {

sessionFactory.close();

}

}

}

public static List<Employee> getEmployeesInSalaryRange(double minSalary, double maxSalary) {

Session session = sessionFactory.openSession();

CriteriaBuilder criteriaBuilder = session.getCriteriaBuilder();

CriteriaQuery<Employee> criteriaQuery = criteriaBuilder.createQuery(Employee.class);

Root<Employee> root = criteriaQuery.from(Employee.class);

criteriaQuery.select(root)

.where(

criteriaBuilder.between(root.get("salary"), minSalary, maxSalary)

);

List<Employee> employees = session.createQuery(criteriaQuery).getResultList();

session.close();

return employees;

}

public static List<Employee> getEmployeesOrderBySalaryDesc() {

Session session = sessionFactory.openSession();

CriteriaBuilder criteriaBuilder = session.getCriteriaBuilder();

CriteriaQuery<Employee> criteriaQuery = criteriaBuilder.createQuery(Employee.class);

Root<Employee> root = criteriaQuery.from(Employee.class);

criteriaQuery.select(root)

.orderBy(criteriaBuilder.desc(root.get("salary")));

List<Employee> employees = session.createQuery(criteriaQuery).getResultList();

session.close();

return employees;

}

public static List<Employee> getEmployeesByDepartmentName(String departmentName) {

Session session = sessionFactory.openSession();

CriteriaBuilder criteriaBuilder = session.getCriteriaBuilder();

CriteriaQuery<Employee> criteriaQuery = criteriaBuilder.createQuery(Employee.class);

Root<Employee> root = criteriaQuery.from(Employee.class);

// Join with the Department entity

Join<Employee, Department> departmentJoin = root.join("department");

criteriaQuery.select(root)

.where(criteriaBuilder.equal(departmentJoin.get("name"), departmentName));

List<Employee> employees = session.createQuery(criteriaQuery).getResultList();

session.close();

return employees;

}

public static List<Employee> getEmployeesByNamePattern(String namePattern) {

Session session = sessionFactory.openSession();

CriteriaBuilder criteriaBuilder = session.getCriteriaBuilder();

CriteriaQuery<Employee> criteriaQuery = criteriaBuilder.createQuery(Employee.class);

Root<Employee> root = criteriaQuery.from(Employee.class);

criteriaQuery.select(root)

.where(criteriaBuilder.like(root.get("name"), "%" + namePattern + "%"));

List<Employee> employees = session.createQuery(criteriaQuery).getResultList();

session.close();

return employees;

}

private static void printEmployees(List<Employee> employees) {

for (Employee employee : employees) {

System.out.println("Employee ID: " + employee.getId()

+ ", Name: " + employee.getName()

+ ", Salary: " + employee.getSalary()

+ ", Department: " + employee.getDepartment().getName());

}

}

}

**Question 02:**

Consider a database schema with two entities: **Course** and **Student**. The **Course** entity has attributes such as **id** (primary key) and **name**, and a one-to-many relationship with the **Student** entity. The **Student** entity has attributes **id** (primary key), **name**, **grade**, and a many-to-one relationship with the **Course** entity.

**Answer:**

import javax.persistence.\*;

import java.util.List;

@Entity

@Table(name = "courses")

public class Course {

@Id

@GeneratedValue(strategy = GenerationType.IDENTITY)

@Column(name = "id")

private Long id;

@Column(name = "name")

private String name;

@OneToMany(mappedBy = "course")

private List<Student> students;

// Constructors, getters, and setters

}

import javax.persistence.\*;

@Entity

@Table(name = "students")

public class Student {

@Id

@GeneratedValue(strategy = GenerationType.IDENTITY)

@Column(name = "id")

private Long id;

@Column(name = "name")

private String name;

@Column(name = "grade")

private int grade;

@ManyToOne

@JoinColumn(name = "course\_id")

private Course course;

// Constructors, getters, and setters

}

To add some rows

import org.hibernate.Session;

import org.hibernate.SessionFactory;

import org.hibernate.Transaction;

import org.hibernate.cfg.Configuration;

import java.text.ParseException;

import java.text.SimpleDateFormat;

import java.util.Date;

public class AddRowsExample {

private static SessionFactory sessionFactory;

static {

try {

// Initialize Hibernate SessionFactory

Configuration configuration = new Configuration().configure();

sessionFactory = configuration.buildSessionFactory();

} catch (Exception e) {

e.printStackTrace();

}

}

public static void main(String[] args) {

try {

// Open a session

Session session = sessionFactory.openSession();

Transaction transaction = session.beginTransaction();

// Add 100 courses

for (int i = 1; i <= 100; i++) {

Course course = new Course();

course.setName("Course " + i);

session.save(course);

}

// Add 500 students

for (int i = 1; i <= 500; i++) {

Student student = new Student();

student.setName("Student " + i);

student.setEnrollmentDate(getRandomEnrollmentDate());

// Assign a random course to the student

Course randomCourse = getRandomCourse(session);

student.setCourse(randomCourse);

session.save(student);

}

// Commit the transaction

transaction.commit();

System.out.println("Rows added successfully!");

} catch (Exception e) {

e.printStackTrace();

} finally {

// Close the sessionFactory to release resources

if (sessionFactory != null) {

sessionFactory.close();

}

}

}

private static Date getRandomEnrollmentDate() {

long startDate = toDate("2020-01-01").getTime();

long endDate = System.currentTimeMillis();

long randomDate = startDate + (long) (Math.random() \* (endDate - startDate));

return new Date(randomDate);

}

private static Date toDate(String dateString) {

try {

return new SimpleDateFormat("yyyy-MM-dd").parse(dateString);

} catch (ParseException e) {

e.printStackTrace();

return null;

}

}

private static Course getRandomCourse(Session session) {

// Retrieve a random course from the database

long courseId = (long) (Math.random() \* 100) + 1;

return session.get(Course.class, courseId);

}

}

Formulate a task using the Hibernate Criteria API to perform the following operation:

1. Retrieve a list of students who are enrolled in a specific course and have a grade greater than or equal to a specified minimum value.

Answer:

import org.hibernate.Session;

import org.hibernate.SessionFactory;

import org.hibernate.Transaction;

import org.hibernate.cfg.Configuration;

import javax.persistence.criteria.\*;

import java.util.List;

public class CriteriaApiExample {

private static SessionFactory sessionFactory;

static {

try {

// Initialize Hibernate SessionFactory

Configuration configuration = new Configuration().configure();

sessionFactory = configuration.buildSessionFactory();

} catch (Exception e) {

e.printStackTrace();

}

}

public static void main(String[] args) {

try {

// Previous exercises...

// Task 6: Retrieve students by course name and minimum grade

List<Student> studentsByCourseAndGrade = getStudentsByCourseAndMinGrade("Math", 80);

System.out.println("\nStudents in Math course with a grade greater than or equal to 80:");

printStudents(studentsByCourseAndGrade);

} finally {

// Close the sessionFactory to release resources

if (sessionFactory != null) {

sessionFactory.close();

}

}

}

public static List<Student> getStudentsByCourseAndMinGrade(String courseName, int minGrade) {

Session session = sessionFactory.openSession();

CriteriaBuilder criteriaBuilder = session.getCriteriaBuilder();

CriteriaQuery<Student> criteriaQuery = criteriaBuilder.createQuery(Student.class);

Root<Student> root = criteriaQuery.from(Student.class);

// Join with the Course entity

Join<Student, Course> courseJoin = root.join("course");

criteriaQuery.select(root)

.where(

criteriaBuilder.equal(courseJoin.get("name"), courseName),

criteriaBuilder.ge(root.get("grade"), minGrade)

);

List<Student> students = session.createQuery(criteriaQuery).getResultList();

session.close();

return students;

}

// Previous methods...

private static void printStudents(List<Student> students) {

for (Student student : students) {

System.out.println("Student ID: " + student.getId()

+ ", Name: " + student.getName()

+ ", Grade: " + student.getGrade()

+ ", Course: " + student.getCourse().getName());

}

}

}

1. Retrieve a list of students who are enrolled in courses with names starting with a specified prefix and joined the course before a certain date.

import org.hibernate.Session;

import org.hibernate.SessionFactory;

import org.hibernate.Transaction;

import org.hibernate.cfg.Configuration;

import javax.persistence.criteria.\*;

import java.text.ParseException;

import java.text.SimpleDateFormat;

import java.util.Date;

import java.util.List;

public class CriteriaApiExample {

private static SessionFactory sessionFactory;

static {

try {

// Initialize Hibernate SessionFactory

Configuration configuration = new Configuration().configure();

sessionFactory = configuration.buildSessionFactory();

} catch (Exception e) {

e.printStackTrace();

}

}

public static void main(String[] args) {

try {

// Previous exercises...

// Task: Retrieve students enrolled in courses with names starting with a specified prefix

// and joined the course before a certain date

List<Student> studentsByCourseAndDate = getStudentsByCoursePrefixAndEnrollmentDate("CS", toDate("2022-01-01"));

System.out.println("\nStudents enrolled in CS courses before 2022-01-01:");

printStudents(studentsByCourseAndDate);

} finally {

// Close the sessionFactory to release resources

if (sessionFactory != null) {

sessionFactory.close();

}

}

}

public static List<Student> getStudentsByCoursePrefixAndEnrollmentDate(String coursePrefix, Date maxEnrollmentDate) {

Session session = sessionFactory.openSession();

CriteriaBuilder criteriaBuilder = session.getCriteriaBuilder();

CriteriaQuery<Student> criteriaQuery = criteriaBuilder.createQuery(Student.class);

Root<Student> root = criteriaQuery.from(Student.class);

// Join with the Course entity

Join<Student, Course> courseJoin = root.join("course");

criteriaQuery.select(root)

.where(

criteriaBuilder.like(courseJoin.get("name"), coursePrefix + "%"),

criteriaBuilder.lessThan(root.get("enrollmentDate"), maxEnrollmentDate)

);

List<Student> students = session.createQuery(criteriaQuery).getResultList();

session.close();

return students;

}

// Previous methods...

private static void printStudents(List<Student> students) {

for (Student student : students) {

System.out.println("Student ID: " + student.getId()

+ ", Name: " + student.getName()

+ ", Enrollment Date: " + student.getEnrollmentDate()

+ ", Course: " + student.getCourse().getName());

}

}

private static Date toDate(String dateString) {

try {

return new SimpleDateFormat("yyyy-MM-dd").parse(dateString);

} catch (ParseException e) {

e.printStackTrace();

return null;

}

}

}

1. Retrieve a list of courses with the highest number of enrolled students.

import org.hibernate.Session;

import org.hibernate.SessionFactory;

import org.hibernate.Transaction;

import org.hibernate.cfg.Configuration;

import javax.persistence.criteria.\*;

import java.util.List;

public class CriteriaApiExample {

private static SessionFactory sessionFactory;

static {

try {

// Initialize Hibernate SessionFactory

Configuration configuration = new Configuration().configure();

sessionFactory = configuration.buildSessionFactory();

} catch (Exception e) {

e.printStackTrace();

}

}

public static void main(String[] args) {

try {

// Previous exercises...

// Task: Retrieve courses with the highest number of enrolled students

List<Course> coursesWithHighestEnrollment = getCoursesWithHighestEnrollment();

System.out.println("\nCourses with the highest number of enrolled students:");

printCourses(coursesWithHighestEnrollment);

} finally {

// Close the sessionFactory to release resources

if (sessionFactory != null) {

sessionFactory.close();

}

}

}

public static List<Course> getCoursesWithHighestEnrollment() {

Session session = sessionFactory.openSession();

CriteriaBuilder criteriaBuilder = session.getCriteriaBuilder();

CriteriaQuery<Course> criteriaQuery = criteriaBuilder.createQuery(Course.class);

Root<Course> root = criteriaQuery.from(Course.class);

// Join with the Student entity and count the number of students per course

Join<Course, Student> studentJoin = root.join("students", JoinType.LEFT);

criteriaQuery.multiselect(root, criteriaBuilder.count(studentJoin).alias("enrollmentCount"))

.groupBy(root.get("id"))

.orderBy(criteriaBuilder.desc(criteriaBuilder.count(studentJoin)));

// Retrieve the course with the highest enrollment count

List<Course> courses = session.createQuery(criteriaQuery)

.setMaxResults(1)

.getResultList();

session.close();

return courses;

}

// Previous methods...

private static void printCourses(List<Course> courses) {

for (Course course : courses) {

System.out.println("Course ID: " + course.getId()

+ ", Name: " + course.getName()

+ ", Enrollment Count: " + course.getStudents().size());

}

}

}

1. Retrieve a list of courses with the highest number of enrolled students.
2. Retrieve a list of courses that have at least one student enrolled before a specified date and also have at least one student enrolled after a different specified date.

Answer for above two questions:

import org.hibernate.Session;

import org.hibernate.SessionFactory;

import org.hibernate.Transaction;

import org.hibernate.cfg.Configuration;

import javax.persistence.criteria.\*;

import java.text.ParseException;

import java.text.SimpleDateFormat;

import java.util.Date;

import java.util.List;

public class CriteriaApiExample {

private static SessionFactory sessionFactory;

static {

try {

// Initialize Hibernate SessionFactory

Configuration configuration = new Configuration().configure();

sessionFactory = configuration.buildSessionFactory();

} catch (Exception e) {

e.printStackTrace();

}

}

public static void main(String[] args) {

try {

// Previous exercises...

// Exercise 1: Retrieve courses with the highest number of enrolled students

List<Course> coursesWithHighestEnrollment = getCoursesWithHighestEnrollment();

System.out.println("\nCourses with the highest number of enrolled students:");

printCourses(coursesWithHighestEnrollment);

// Exercise 2: Retrieve courses with at least one student enrolled before a specified date

// and at least one student enrolled after a different specified date

Date startDate = toDate("2021-01-01");

Date endDate = toDate("2022-01-01");

List<Course> coursesWithEnrollmentRange = getCoursesWithEnrollmentRange(startDate, endDate);

System.out.println("\nCourses with at least one student enrolled before 2021-01-01 "

+ "and at least one student enrolled after 2022-01-01:");

printCourses(coursesWithEnrollmentRange);

} finally {

// Close the sessionFactory to release resources

if (sessionFactory != null) {

sessionFactory.close();

}

}

}

public static List<Course> getCoursesWithHighestEnrollment() {

Session session = sessionFactory.openSession();

CriteriaBuilder criteriaBuilder = session.getCriteriaBuilder();

CriteriaQuery<Course> criteriaQuery = criteriaBuilder.createQuery(Course.class);

Root<Course> root = criteriaQuery.from(Course.class);

// Join with the Student entity and count the number of students per course

Join<Course, Student> studentJoin = root.join("students", JoinType.LEFT);

criteriaQuery.multiselect(root, criteriaBuilder.count(studentJoin).alias("enrollmentCount"))

.groupBy(root.get("id"))

.orderBy(criteriaBuilder.desc(criteriaBuilder.count(studentJoin)));

// Retrieve the course with the highest enrollment count

List<Course> courses = session.createQuery(criteriaQuery)

.setMaxResults(1)

.getResultList();

session.close();

return courses;

}

public static List<Course> getCoursesWithEnrollmentRange(Date startDate, Date endDate) {

Session session = sessionFactory.openSession();

CriteriaBuilder criteriaBuilder = session.getCriteriaBuilder();

CriteriaQuery<Course> criteriaQuery = criteriaBuilder.createQuery(Course.class);

Root<Course> root = criteriaQuery.from(Course.class);

// Join with the Student entity and add conditions for enrollment date range

Join<Course, Student> studentJoin = root.join("students", JoinType.LEFT);

Predicate enrollmentBeforeStartDate = criteriaBuilder.lessThan(studentJoin.get("enrollmentDate"), startDate);

Predicate enrollmentAfterEndDate = criteriaBuilder.greaterThan(studentJoin.get("enrollmentDate"), endDate);

criteriaQuery.select(root)

.where(

criteriaBuilder.exists(criteriaQuery.subquery(Student.class)

.select(studentJoin)

.where(enrollmentBeforeStartDate)),

criteriaBuilder.exists(criteriaQuery.subquery(Student.class)

.select(studentJoin)

.where(enrollmentAfterEndDate))

);

List<Course> courses = session.createQuery(criteriaQuery).getResultList();

session.close();

return courses;

}

private static void printCourses(List<Course> courses) {

for (Course course : courses) {

System.out.println("Course ID: " + course.getId()

+ ", Name: " + course.getName());

}

}

private static Date toDate(String dateString) {

try {

return new SimpleDateFormat("yyyy-MM-dd").parse(dateString);

} catch (ParseException e) {

e.printStackTrace();

return null;

}

}

}

1. Retrieve a list of students who are enrolled in courses that have the letter 'A' in their names.

**Answer:**

public static List<Student> getStudentsInCoursesWithLetterA() {

Session session = sessionFactory.openSession();

CriteriaBuilder criteriaBuilder = session.getCriteriaBuilder();

CriteriaQuery<Student> criteriaQuery = criteriaBuilder.createQuery(Student.class);

Root<Student> root = criteriaQuery.from(Student.class);

// Join with the Course entity and add conditions for course name containing 'A'

Join<Student, Course> courseJoin = root.join("course", JoinType.INNER);

Predicate courseNameContainsA = criteriaBuilder.like(courseJoin.get("name"), "%A%");

criteriaQuery.select(root)

.where(courseNameContainsA);

List<Student> students = session.createQuery(criteriaQuery).getResultList();

session.close();

return students;

}

1. Retrieve a list of courses that have an average student grade greater than a specified value.

**Answer:**

public static List<Course> getCoursesWithAverageGradeGreaterThan(double minAverageGrade) {

Session session = sessionFactory.openSession();

CriteriaBuilder criteriaBuilder = session.getCriteriaBuilder();

CriteriaQuery<Course> criteriaQuery = criteriaBuilder.createQuery(Course.class);

Root<Course> root = criteriaQuery.from(Course.class);

// Join with the Student entity and calculate the average grade

Join<Course, Student> studentJoin = root.join("students", JoinType.LEFT);

Expression<Double> averageGrade = criteriaBuilder.avg(studentJoin.get("grade"));

criteriaQuery.select(root)

.groupBy(root.get("id"))

.having(criteriaBuilder.greaterThan(averageGrade, minAverageGrade));

List<Course> courses = session.createQuery(criteriaQuery).getResultList();

session.close();

return courses;

}

1. Retrieve a list of students who are enrolled in a specific course and have a grade greater than or equal to a specified minimum value.

**Answer:**

public static List<Student> getStudentsByCourseAndMinGrade(String courseName, int minGrade) {

Session session = sessionFactory.openSession();

CriteriaBuilder criteriaBuilder = session.getCriteriaBuilder();

CriteriaQuery<Student> criteriaQuery = criteriaBuilder.createQuery(Student.class);

Root<Student> root = criteriaQuery.from(Student.class);

// Join with the Course entity

Join<Student, Course> courseJoin = root.join("course");

criteriaQuery.select(root)

.where(

criteriaBuilder.equal(courseJoin.get("name"), courseName),

criteriaBuilder.ge(root.get("grade"), minGrade)

);

List<Student> students = session.createQuery(criteriaQuery).getResultList();

session.close();

return students;

}

1. Retrieve a list of courses that have students with grades in a specified range (e.g., between 70 and 90).

**Answer:**

public static List<Course> getCoursesWithStudentsInGradeRange(int minGrade, int maxGrade) {

Session session = sessionFactory.openSession();

CriteriaBuilder criteriaBuilder = session.getCriteriaBuilder();

CriteriaQuery<Course> criteriaQuery = criteriaBuilder.createQuery(Course.class);

Root<Course> root = criteriaQuery.from(Course.class);

// Join with the Student entity and add conditions for grade range

Join<Course, Student> studentJoin = root.join("students", JoinType.LEFT);

Predicate gradeInRange = criteriaBuilder.between(studentJoin.get("grade"), minGrade, maxGrade);

criteriaQuery.select(root)

.distinct(true) // To avoid duplicate courses in case a course has multiple students in the grade range

.where(gradeInRange);

List<Course> courses = session.createQuery(criteriaQuery).getResultList();

session.close();

return courses;

}

1. Retrieve a list of students who are not enrolled in any course.

**Answer:**

public static List<Student> getStudentsNotEnrolledInAnyCourse() {

Session session = sessionFactory.openSession();

CriteriaBuilder criteriaBuilder = session.getCriteriaBuilder();

CriteriaQuery<Student> criteriaQuery = criteriaBuilder.createQuery(Student.class);

Root<Student> root = criteriaQuery.from(Student.class);

criteriaQuery.select(root)

.where(criteriaBuilder.isEmpty(root.get("course")));

List<Student> students = session.createQuery(criteriaQuery).getResultList();

session.close();

return students;

}

**Question 03:**

Suppose you have a database schema with two entities: **Book** and **Author**. The **Book** entity has attributes such as **id** (primary key), **title**, **publicationYear**, and a many-to-one relationship with the **Author** entity. The **Author** entity has attributes **id** (primary key), **name**, and a one-to-many relationship with the **Book** entity.

**Answer:**

import javax.persistence.\*;

import java.util.Objects;

@Entity

@Table(name = "books")

public class Book {

@Id

@GeneratedValue(strategy = GenerationType.IDENTITY)

@Column(name = "id")

private Long id;

@Column(name = "title")

private String title;

@Column(name = "publication\_year")

private int publicationYear;

@ManyToOne

@JoinColumn(name = "author\_id")

private Author author;

// Constructors, getters, and setters

@Override

public boolean equals(Object o) {

if (this == o) return true;

if (o == null || getClass() != o.getClass()) return false;

Book book = (Book) o;

return Objects.equals(id, book.id);

}

@Override

public int hashCode() {

return Objects.hash(id);

}

}

import javax.persistence.\*;

import java.util.List;

import java.util.Objects;

@Entity

@Table(name = "authors")

public class Author {

@Id

@GeneratedValue(strategy = GenerationType.IDENTITY)

@Column(name = "id")

private Long id;

@Column(name = "name")

private String name;

@OneToMany(mappedBy = "author")

private List<Book> books;

// Constructors, getters, and setters

@Override

public boolean equals(Object o) {

if (this == o) return true;

if (o == null || getClass() != o.getClass()) return false;

Author author = (Author) o;

return Objects.equals(id, author.id);

}

@Override

public int hashCode() {

return Objects.hash(id);

}

}

To insert rows into books and authors

import org.hibernate.Session;

import org.hibernate.SessionFactory;

import org.hibernate.Transaction;

import org.hibernate.cfg.Configuration;

import java.util.Arrays;

import java.util.List;

import java.util.Random;

public class AddRealRowsExample {

private static SessionFactory sessionFactory;

static {

try {

// Initialize Hibernate SessionFactory

Configuration configuration = new Configuration().configure();

sessionFactory = configuration.buildSessionFactory();

} catch (Exception e) {

e.printStackTrace();

}

}

public static void main(String[] args) {

try {

// Open a session

Session session = sessionFactory.openSession();

Transaction transaction = session.beginTransaction();

// Add real authors and books data (using fictional names)

List<String> authorNames = Arrays.asList(

"Alice Johnson", "Bob Smith", "Catherine Davis", "David Miller", "Emma Wilson",

"Frank White", "Grace Brown", "Henry Jones", "Ivy Robinson", "Jack Thompson",

"Katherine Lee", "Leo Martinez", "Mia Garcia", "Nathan Hall", "Olivia Wright"

);

List<String> bookTitles = Arrays.asList(

"The Enchanted Forest", "Echoes of Eternity", "Lost in Dreams", "Beyond the Stars",

"The Midnight Serenade", "Whispers in the Wind", "Eternal Love", "The Forgotten Realm",

"A Tale of Two Cities", "The Secret Garden", "Shadows of Destiny", "A Symphony of Stars",

"Ripples of Time", "The Art of Silence", "The Whispering Pines"

);

for (String authorName : authorNames) {

Author author = new Author();

author.setName(authorName);

session.save(author);

for (String bookTitle : bookTitles) {

Book book = new Book();

book.setTitle(bookTitle);

book.setPublicationYear(2020 + new Random().nextInt(5)); // Random publication year between 2020 and 2024

book.setAuthor(author);

session.save(book);

}

}

// Commit the transaction

transaction.commit();

System.out.println("Rows added successfully!");

} catch (Exception e) {

e.printStackTrace();

} finally {

// Close the sessionFactory to release resources

if (sessionFactory != null) {

sessionFactory.close();

}

}

}

}

Formulate a task using the Hibernate Criteria API to perform the following operation:

1. Retrieve a list of books published after a certain year, written by authors who have more than a specified number of books.
2. Top of Form

import org.hibernate.Session;

import org.hibernate.SessionFactory;

import org.hibernate.Transaction;

import org.hibernate.cfg.Configuration;

import javax.persistence.criteria.\*;

import java.util.List;

public class CriteriaApiExample {

private static SessionFactory sessionFactory;

static {

try {

// Initialize Hibernate SessionFactory

Configuration configuration = new Configuration().configure();

sessionFactory = configuration.buildSessionFactory();

} catch (Exception e) {

e.printStackTrace();

}

}

public static void main(String[] args) {

try {

// Task: Retrieve books published after a certain year, written by authors with more than a specified number of books

int minBookCount = 3;

int publicationYear = 2010;

List<Book> books = getBooksByAuthorBookCountAndPublicationYear(minBookCount, publicationYear);

System.out.println("\nBooks published after 2010, written by authors with more than 3 books:");

printBooks(books);

} finally {

// Close the sessionFactory to release resources

if (sessionFactory != null) {

sessionFactory.close();

}

}

}

public static List<Book> getBooksByAuthorBookCountAndPublicationYear(int minBookCount, int publicationYear) {

Session session = sessionFactory.openSession();

CriteriaBuilder criteriaBuilder = session.getCriteriaBuilder();

CriteriaQuery<Book> criteriaQuery = criteriaBuilder.createQuery(Book.class);

Root<Book> root = criteriaQuery.from(Book.class);

// Join with the Author entity and add conditions for book count and publication year

Join<Book, Author> authorJoin = root.join("author", JoinType.INNER);

Predicate authorBookCountGreaterThan = criteriaBuilder.greaterThan(criteriaBuilder.size(authorJoin.get("books")), minBookCount);

Predicate publicationYearCondition = criteriaBuilder.greaterThan(root.get("publicationYear"), publicationYear);

criteriaQuery.select(root)

.where(

authorBookCountGreaterThan,

publicationYearCondition

);

List<Book> books = session.createQuery(criteriaQuery).getResultList();

session.close();

return books;

}

private static void printBooks(List<Book> books) {

for (Book book : books) {

System.out.println("Book ID: " + book.getId()

+ ", Title: " + book.getTitle()

+ ", Publication Year: " + book.getPublicationYear()

+ ", Author: " + book.getAuthor().getName());

}

}

}

2. Retrieve a list of authors who have written books with titles containing a specified keyword, and the total number of books they've written is greater than a specified threshold.

import org.hibernate.Session;

import org.hibernate.SessionFactory;

import org.hibernate.Transaction;

import org.hibernate.cfg.Configuration;

import javax.persistence.criteria.\*;

import java.util.List;

public class CriteriaApiExample {

private static SessionFactory sessionFactory;

static {

try {

// Initialize Hibernate SessionFactory

Configuration configuration = new Configuration().configure();

sessionFactory = configuration.buildSessionFactory();

} catch (Exception e) {

e.printStackTrace();

}

}

public static void main(String[] args) {

try {

// Task: Retrieve authors who have written books with titles containing a specified keyword,

// and the total number of books they've written is greater than a specified threshold.

String keyword = "Adventure";

int minBookCount = 5;

List<Author> authors = getAuthorsByBookTitleKeywordAndBookCount(keyword, minBookCount);

System.out.println("\nAuthors who have written books with titles containing 'Adventure' and have more than 5 books:");

printAuthors(authors);

} finally {

// Close the sessionFactory to release resources

if (sessionFactory != null) {

sessionFactory.close();

}

}

}

public static List<Author> getAuthorsByBookTitleKeywordAndBookCount(String keyword, int minBookCount) {

Session session = sessionFactory.openSession();

CriteriaBuilder criteriaBuilder = session.getCriteriaBuilder();

CriteriaQuery<Author> criteriaQuery = criteriaBuilder.createQuery(Author.class);

Root<Author> root = criteriaQuery.from(Author.class);

// Join with the Book entity and add conditions for title keyword and book count

Join<Author, Book> bookJoin = root.join("books", JoinType.INNER);

Predicate titleContainsKeyword = criteriaBuilder.like(bookJoin.get("title"), "%" + keyword + "%");

Predicate bookCountGreaterThan = criteriaBuilder.greaterThan(criteriaBuilder.size(root.get("books")), minBookCount);

criteriaQuery.select(root)

.distinct(true) // To avoid duplicate authors if they have multiple books with the specified keyword

.where(

titleContainsKeyword,

bookCountGreaterThan

);

List<Author> authors = session.createQuery(criteriaQuery).getResultList();

session.close();

return authors;

}

private static void printAuthors(List<Author> authors) {

for (Author author : authors) {

System.out.println("Author ID: " + author.getId()

+ ", Name: " + author.getName()

+ ", Total Books Written: " + author.getBooks().size());

}

}

}

1. Retrieve a list of books written by authors who have more than a specified number of books, and the average publication year of their books is older than a given year.

import org.hibernate.Session;

import org.hibernate.SessionFactory;

import org.hibernate.Transaction;

import org.hibernate.cfg.Configuration;

import javax.persistence.criteria.\*;

import java.util.List;

public class CriteriaApiExample {

private static SessionFactory sessionFactory;

static {

try {

// Initialize Hibernate SessionFactory

Configuration configuration = new Configuration().configure();

sessionFactory = configuration.buildSessionFactory();

} catch (Exception e) {

e.printStackTrace();

}

}

public static void main(String[] args) {

try {

// Task: Retrieve books written by authors who have more than 4 books,

// and the average publication year of their books is older than 2005.

int minBookCount = 4;

int minAveragePublicationYear = 2005;

List<Book> books = getBooksByAuthorBookCountAndAveragePublicationYear(minBookCount, minAveragePublicationYear);

System.out.println("\nBooks written by authors with more than 4 books and average publication year older than 2005:");

printBooks(books);

} finally {

// Close the sessionFactory to release resources

if (sessionFactory != null) {

sessionFactory.close();

}

}

}

public static List<Book> getBooksByAuthorBookCountAndAveragePublicationYear(int minBookCount, int minAveragePublicationYear) {

Session session = sessionFactory.openSession();

CriteriaBuilder criteriaBuilder = session.getCriteriaBuilder();

CriteriaQuery<Book> criteriaQuery = criteriaBuilder.createQuery(Book.class);

Root<Book> root = criteriaQuery.from(Book.class);

// Join with the Author entity and add conditions for book count and average publication year

Join<Book, Author> authorJoin = root.join("author", JoinType.INNER);

Predicate authorBookCountGreaterThan = criteriaBuilder.greaterThan(criteriaBuilder.size(authorJoin.get("books")), minBookCount);

Expression<Double> averagePublicationYear = criteriaBuilder.avg(root.get("publicationYear"));

Predicate averagePublicationYearGreaterThan = criteriaBuilder.greaterThan(averagePublicationYear, (double) minAveragePublicationYear);

criteriaQuery.select(root)

.groupBy(root.get("id"))

.having(

authorBookCountGreaterThan,

averagePublicationYearGreaterThan

);

List<Book> books = session.createQuery(criteriaQuery).getResultList();

session.close();

return books;

}

private static void printBooks(List<Book> books) {

for (Book book : books) {

System.out.println("Book ID: " + book.getId()

+ ", Title: " + book.getTitle()

+ ", Publication Year: " + book.getPublicationYear()

+ ", Author: " + book.getAuthor().getName());

}

}

}

1. Retrieve a list of authors who have written books with titles containing a specified keyword, and the average publication year of their books is within a given range.

import org.hibernate.Session;

import org.hibernate.SessionFactory;

import org.hibernate.Transaction;

import org.hibernate.cfg.Configuration;

import javax.persistence.criteria.\*;

import java.util.List;

public class CriteriaApiExample {

private static SessionFactory sessionFactory;

static {

try {

// Initialize Hibernate SessionFactory

Configuration configuration = new Configuration().configure();

sessionFactory = configuration.buildSessionFactory();

} catch (Exception e) {

e.printStackTrace();

}

}

public static void main(String[] args) {

try {

// Task: Retrieve authors who have written books with titles containing 'Mystery',

// and the average publication year of their books is between 2000 and 2010.

String keyword = "Mystery";

int minAveragePublicationYear = 2000;

int maxAveragePublicationYear = 2010;

List<Author> authors = getAuthorsByBookTitleKeywordAndAveragePublicationYearRange(keyword, minAveragePublicationYear, maxAveragePublicationYear);

System.out.println("\nAuthors who have written books with titles containing 'Mystery' and average publication year between 2000 and 2010:");

printAuthors(authors);

} finally {

// Close the sessionFactory to release resources

if (sessionFactory != null) {

sessionFactory.close();

}

}

}

public static List<Author> getAuthorsByBookTitleKeywordAndAveragePublicationYearRange(String keyword, int minAveragePublicationYear, int maxAveragePublicationYear) {

Session session = sessionFactory.openSession();

CriteriaBuilder criteriaBuilder = session.getCriteriaBuilder();

CriteriaQuery<Author> criteriaQuery = criteriaBuilder.createQuery(Author.class);

Root<Author> root = criteriaQuery.from(Author.class);

// Join with the Book entity and add conditions for title keyword and average publication year range

Join<Author, Book> bookJoin = root.join("books", JoinType.INNER);

Predicate titleContainsKeyword = criteriaBuilder.like(bookJoin.get("title"), "%" + keyword + "%");

Expression<Double> averagePublicationYear = criteriaBuilder.avg(bookJoin.get("publicationYear"));

Predicate averagePublicationYearInRange = criteriaBuilder.between(averagePublicationYear, (double) minAveragePublicationYear, (double) maxAveragePublicationYear);

criteriaQuery.select(root)

.distinct(true) // To avoid duplicate authors if they have multiple books with the specified keyword

.where(

titleContainsKeyword,

averagePublicationYearInRange

);

List<Author> authors = session.createQuery(criteriaQuery).getResultList();

session.close();

return authors;

}

private static void printAuthors(List<Author> authors) {

for (Author author : authors) {

System.out.println("Author ID: " + author.getId()

+ ", Name: " + author.getName()

+ ", Average Publication Year: " + calculateAveragePublicationYear(author));

}

}

private static double calculateAveragePublicationYear(Author author) {

double totalPublicationYears = 0;

for (Book book : author.getBooks()) {

totalPublicationYears += book.getPublicationYear();

}

return totalPublicationYears / author.getBooks().size();

}

}

1. Retrieve a list of books written by authors who have at least one book published before a specified year and at least one book published after another specified year.

import org.hibernate.Session;

import org.hibernate.SessionFactory;

import org.hibernate.Transaction;

import org.hibernate.cfg.Configuration;

import javax.persistence.criteria.\*;

import java.util.List;

public class CriteriaApiExample {

private static SessionFactory sessionFactory;

static {

try {

// Initialize Hibernate SessionFactory

Configuration configuration = new Configuration().configure();

sessionFactory = configuration.buildSessionFactory();

} catch (Exception e) {

e.printStackTrace();

}

}

public static void main(String[] args) {

try {

// Task: Retrieve books written by authors who have at least one book published before 2000

// and at least one book published after 2010.

int minBeforeYear = 2000;

int maxAfterYear = 2010;

List<Book> books = getBooksByAuthorsWithPublicationYearRange(minBeforeYear, maxAfterYear);

System.out.println("\nBooks written by authors with at least one book published before 2000 and at least one book published after 2010:");

printBooks(books);

} finally {

// Close the sessionFactory to release resources

if (sessionFactory != null) {

sessionFactory.close();

}

}

}

public static List<Book> getBooksByAuthorsWithPublicationYearRange(int minBeforeYear, int maxAfterYear) {

Session session = sessionFactory.openSession();

CriteriaBuilder criteriaBuilder = session.getCriteriaBuilder();

CriteriaQuery<Book> criteriaQuery = criteriaBuilder.createQuery(Book.class);

Root<Book> root = criteriaQuery.from(Book.class);

// Join with the Author entity and add conditions for publication year range

Join<Book, Author> authorJoin = root.join("author", JoinType.INNER);

Predicate beforeYearCondition = criteriaBuilder.lessThan(root.get("publicationYear"), minBeforeYear);

Predicate afterYearCondition = criteriaBuilder.greaterThan(root.get("publicationYear"), maxAfterYear);

criteriaQuery.select(root)

.distinct(true) // To avoid duplicate books if they have multiple authors meeting the criteria

.where(

beforeYearCondition,

afterYearCondition

);

List<Book> books = session.createQuery(criteriaQuery).getResultList();

session.close();

return books;

}

private static void printBooks(List<Book> books) {

for (Book book : books) {

System.out.println("Book ID: " + book.getId()

+ ", Title: " + book.getTitle()

+ ", Publication Year: " + book.getPublicationYear()

+ ", Author: " + book.getAuthor().getName());

}

}

}

1. Retrieve a list of authors who have written books with titles starting with a specified prefix, and the total number of books they've written is within a specified range.

import org.hibernate.Session;

import org.hibernate.SessionFactory;

import org.hibernate.Transaction;

import org.hibernate.cfg.Configuration;

import javax.persistence.criteria.\*;

import java.util.List;

public class CriteriaApiExample {

private static SessionFactory sessionFactory;

static {

try {

// Initialize Hibernate SessionFactory

Configuration configuration = new Configuration().configure();

sessionFactory = configuration.buildSessionFactory();

} catch (Exception e) {

e.printStackTrace();

}

}

public static void main(String[] args) {

try {

// Task: Retrieve authors who have written books with titles starting with 'The',

// and the total number of books they've written is between 3 and 5.

String titlePrefix = "The";

int minBookCount = 3;

int maxBookCount = 5;

List<Author> authors = getAuthorsByBookTitlePrefixAndBookCountRange(titlePrefix, minBookCount, maxBookCount);

System.out.println("\nAuthors who have written books with titles starting with 'The' and have written between 3 and 5 books:");

printAuthors(authors);

} finally {

// Close the sessionFactory to release resources

if (sessionFactory != null) {

sessionFactory.close();

}

}

}

public static List<Author> getAuthorsByBookTitlePrefixAndBookCountRange(String titlePrefix, int minBookCount, int maxBookCount) {

Session session = sessionFactory.openSession();

CriteriaBuilder criteriaBuilder = session.getCriteriaBuilder();

CriteriaQuery<Author> criteriaQuery = criteriaBuilder.createQuery(Author.class);

Root<Author> root = criteriaQuery.from(Author.class);

// Join with the Book entity and add conditions for title prefix and book count range

Join<Author, Book> bookJoin = root.join("books", JoinType.INNER);

Predicate titleStartsWithPrefix = criteriaBuilder.like(bookJoin.get("title"), titlePrefix + "%");

Predicate bookCountInRange = criteriaBuilder.between(criteriaBuilder.size(root.get("books")), minBookCount, maxBookCount);

criteriaQuery.select(root)

.distinct(true) // To avoid duplicate authors if they have multiple books with the specified title prefix

.where(

titleStartsWithPrefix,

bookCountInRange

);

List<Author> authors = session.createQuery(criteriaQuery).getResultList();

session.close();

return authors;

}

private static void printAuthors(List<Author> authors) {

for (Author author : authors) {

System.out.println("Author ID: " + author.getId()

+ ", Name: " + author.getName()

+ ", Total Books Written: " + author.getBooks().size());

}

}

}

1. Retrieve a list of authors who have written books with titles ending with a specified suffix, and the average publication year of their books is within a given range.

import org.hibernate.Session;

import org.hibernate.SessionFactory;

import org.hibernate.Transaction;

import org.hibernate.cfg.Configuration;

import javax.persistence.criteria.\*;

import java.util.List;

public class CriteriaApiExample {

private static SessionFactory sessionFactory;

static {

try {

// Initialize Hibernate SessionFactory

Configuration configuration = new Configuration().configure();

sessionFactory = configuration.buildSessionFactory();

} catch (Exception e) {

e.printStackTrace();

}

}

public static void main(String[] args) {

try {

// Task: Retrieve authors who have written books with titles ending with 'Chronicles',

// and the average publication year of their books is between 1990 and 2000.

String titleSuffix = "Chronicles";

int minAveragePublicationYear = 1990;

int maxAveragePublicationYear = 2000;

List<Author> authors = getAuthorsByBookTitleSuffixAndAveragePublicationYearRange(titleSuffix, minAveragePublicationYear, maxAveragePublicationYear);

System.out.println("\nAuthors who have written books with titles ending with 'Chronicles' and average publication year between 1990 and 2000:");

printAuthors(authors);

} finally {

// Close the sessionFactory to release resources

if (sessionFactory != null) {

sessionFactory.close();

}

}

}

public static List<Author> getAuthorsByBookTitleSuffixAndAveragePublicationYearRange(String titleSuffix, int minAveragePublicationYear, int maxAveragePublicationYear) {

Session session = sessionFactory.openSession();

CriteriaBuilder criteriaBuilder = session.getCriteriaBuilder();

CriteriaQuery<Author> criteriaQuery = criteriaBuilder.createQuery(Author.class);

Root<Author> root = criteriaQuery.from(Author.class);

// Join with the Book entity and add conditions for title suffix and average publication year range

Join<Author, Book> bookJoin = root.join("books", JoinType.INNER);

Predicate titleEndsWithSuffix = criteriaBuilder.like(bookJoin.get("title"), "%" + titleSuffix);

Expression<Double> averagePublicationYear = criteriaBuilder.avg(bookJoin.get("publicationYear"));

Predicate averagePublicationYearInRange = criteriaBuilder.between(averagePublicationYear, (double) minAveragePublicationYear, (double) maxAveragePublicationYear);

criteriaQuery.select(root)

.distinct(true) // To avoid duplicate authors if they have multiple books with the specified title suffix

.where(

titleEndsWithSuffix,

averagePublicationYearInRange

);

List<Author> authors = session.createQuery(criteriaQuery).getResultList();

session.close();

return authors;

}

private static void printAuthors(List<Author> authors) {

for (Author author : authors) {

System.out.println("Author ID: " + author.getId()

+ ", Name: " + author.getName()

+ ", Average Publication Year: " + calculateAveragePublicationYear(author));

}

}

private static double calculateAveragePublicationYear(Author author) {

double totalPublicationYears = 0;

for (Book book : author.getBooks()) {

totalPublicationYears += book.getPublicationYear();

}

return totalPublicationYears / author.getBooks().size();

}

}

1. Retrieve a list of books written by authors who have written books with titles containing a specified keyword, and the total number of books they've written is less than a specified threshold.

import org.hibernate.Session;

import org.hibernate.SessionFactory;

import org.hibernate.Transaction;

import org.hibernate.cfg.Configuration;

import javax.persistence.criteria.\*;

import java.util.List;

public class CriteriaApiExample {

private static SessionFactory sessionFactory;

static {

try {

// Initialize Hibernate SessionFactory

Configuration configuration = new Configuration().configure();

sessionFactory = configuration.buildSessionFactory();

} catch (Exception e) {

e.printStackTrace();

}

}

public static void main(String[] args) {

try {

// Task: Retrieve books written by authors who have written books with titles containing 'Science',

// and the total number of books they've written is less than 4.

String keyword = "Science";

int maxBookCount = 4;

List<Book> books = getBooksByAuthorsWithBookTitleKeywordAndMaxBookCount(keyword, maxBookCount);

System.out.println("\nBooks written by authors with books containing 'Science' and have written less than 4 books:");

printBooks(books);

} finally {

// Close the sessionFactory to release resources

if (sessionFactory != null) {

sessionFactory.close();

}

}

}

public static List<Book> getBooksByAuthorsWithBookTitleKeywordAndMaxBookCount(String keyword, int maxBookCount) {

Session session = sessionFactory.openSession();

CriteriaBuilder criteriaBuilder = session.getCriteriaBuilder();

CriteriaQuery<Book> criteriaQuery = criteriaBuilder.createQuery(Book.class);

Root<Book> root = criteriaQuery.from(Book.class);

// Join with the Author entity and add conditions for title keyword and maximum book count

Join<Book, Author> authorJoin = root.join("author", JoinType.INNER);

Predicate titleContainsKeyword = criteriaBuilder.like(root.get("title"), "%" + keyword + "%");

Predicate bookCountLessThan = criteriaBuilder.lessThan(criteriaBuilder.size(authorJoin.get("books")), maxBookCount);

criteriaQuery.select(root)

.where(

titleContainsKeyword,

bookCountLessThan

);

List<Book> books = session.createQuery(criteriaQuery).getResultList();

session.close();

return books;

}

private static void printBooks(List<Book> books) {

for (Book book : books) {

System.out.println("Book ID: " + book.getId()

+ ", Title: " + book.getTitle()

+ ", Publication Year: " + book.getPublicationYear()

+ ", Author: " + book.getAuthor().getName());

}

}

}

1. Retrieve a list of authors who have written books published within a specified range of years, and the average publication year of their books is greater than a given year.

import org.hibernate.Session;

import org.hibernate.SessionFactory;

import org.hibernate.Transaction;

import org.hibernate.cfg.Configuration;

import javax.persistence.criteria.\*;

import java.util.List;

public class CriteriaApiExample {

private static SessionFactory sessionFactory;

static {

try {

// Initialize Hibernate SessionFactory

Configuration configuration = new Configuration().configure();

sessionFactory = configuration.buildSessionFactory();

} catch (Exception e) {

e.printStackTrace();

}

}

public static void main(String[] args) {

try {

// Task: Retrieve authors who have written books published between 2005 and 2015,

// and the average publication year of their books is greater than 2010.

int minPublicationYear = 2005;

int maxPublicationYear = 2015;

int minAveragePublicationYear = 2010;

List<Author> authors = getAuthorsByBookPublicationYearAndAveragePublicationYear(minPublicationYear, maxPublicationYear, minAveragePublicationYear);

System.out.println("\nAuthors who have written books published between 2005 and 2015, and average publication year greater than 2010:");

printAuthors(authors);

} finally {

// Close the sessionFactory to release resources

if (sessionFactory != null) {

sessionFactory.close();

}

}

}

public static List<Author> getAuthorsByBookPublicationYearAndAveragePublicationYear(int minPublicationYear, int maxPublicationYear, int minAveragePublicationYear) {

Session session = sessionFactory.openSession();

CriteriaBuilder criteriaBuilder = session.getCriteriaBuilder();

CriteriaQuery<Author> criteriaQuery = criteriaBuilder.createQuery(Author.class);

Root<Author> root = criteriaQuery.from(Author.class);

// Join with the Book entity and add conditions for publication year range and average publication year

Join<Author, Book> bookJoin = root.join("books", JoinType.INNER);

Predicate publicationYearBetween = criteriaBuilder.between(bookJoin.get("publicationYear"), minPublicationYear, maxPublicationYear);

Expression<Double> averagePublicationYear = criteriaBuilder.avg(bookJoin.get("publicationYear"));

Predicate averagePublicationYearGreaterThan = criteriaBuilder.greaterThan(averagePublicationYear, (double) minAveragePublicationYear);

criteriaQuery.select(root)

.groupBy(root.get("id"))

.having(

publicationYearBetween,

averagePublicationYearGreaterThan

);

List<Author> authors = session.createQuery(criteriaQuery).getResultList();

session.close();

return authors;

}

private static void printAuthors(List<Author> authors) {

for (Author author : authors) {

System.out.println("Author ID: " + author.getId()

+ ", Name: " + author.getName()

+ ", Average Publication Year: " + calculateAveragePublicationYear(author));

}

}

private static double calculateAveragePublicationYear(Author author) {

double totalPublicationYears = 0;

for (Book book : author.getBooks()) {

totalPublicationYears += book.getPublicationYear();

}

return totalPublicationYears / author.getBooks().size();

}

}

1. Retrieve a list of authors who have written books with titles containing a specified keyword, and the total number of books they've written is equal to a specified number.

import org.hibernate.Session;

import org.hibernate.SessionFactory;

import org.hibernate.Transaction;

import org.hibernate.cfg.Configuration;

import javax.persistence.criteria.\*;

import java.util.List;

public class CriteriaApiExample {

private static SessionFactory sessionFactory;

static {

try {

// Initialize Hibernate SessionFactory

Configuration configuration = new Configuration().configure();

sessionFactory = configuration.buildSessionFactory();

} catch (Exception e) {

e.printStackTrace();

}

}

public static void main(String[] args) {

try {

// Task: Retrieve authors who have written books with titles containing 'Java',

// and the total number of books they've written is equal to 3.

String keyword = "Java";

int exactBookCount = 3;

List<Author> authors = getAuthorsByBookTitleKeywordAndExactBookCount(keyword, exactBookCount);

System.out.println("\nAuthors who have written books with titles containing 'Java' and have written exactly 3 books:");

printAuthors(authors);

} finally {

// Close the sessionFactory to release resources

if (sessionFactory != null) {

sessionFactory.close();

}

}

}

public static List<Author> getAuthorsByBookTitleKeywordAndExactBookCount(String keyword, int exactBookCount) {

Session session = sessionFactory.openSession();

CriteriaBuilder criteriaBuilder = session.getCriteriaBuilder();

CriteriaQuery<Author> criteriaQuery = criteriaBuilder.createQuery(Author.class);

Root<Author> root = criteriaQuery.from(Author.class);

// Join with the Book entity and add conditions for title keyword and exact book count

Join<Author, Book> bookJoin = root.join("books", JoinType.INNER);

Predicate titleContainsKeyword = criteriaBuilder.like(bookJoin.get("title"), "%" + keyword + "%");

Predicate bookCountEquals = criteriaBuilder.equal(criteriaBuilder.size(root.get("books")), exactBookCount);

criteriaQuery.select(root)

.distinct(true) // To avoid duplicate authors if they have multiple books with the specified title keyword

.where(

titleContainsKeyword,

bookCountEquals

);

List<Author> authors = session.createQuery(criteriaQuery).getResultList();

session.close();

return authors;

}

private static void printAuthors(List<Author> authors) {

for (Author author : authors) {

System.out.println("Author ID: " + author.getId()

+ ", Name: " + author.getName()

+ ", Total Books Written: " + author.getBooks().size());

}

}

}

1. Retrieve a list of books published within a specified range of years, and the authors of these books have written books with titles containing a specified keyword.

import org.hibernate.Session;

import org.hibernate.SessionFactory;

import org.hibernate.Transaction;

import org.hibernate.cfg.Configuration;

import javax.persistence.criteria.\*;

import java.util.List;

public class CriteriaApiExample {

private static SessionFactory sessionFactory;

static {

try {

// Initialize Hibernate SessionFactory

Configuration configuration = new Configuration().configure();

sessionFactory = configuration.buildSessionFactory();

} catch (Exception e) {

e.printStackTrace();

}

}

public static void main(String[] args) {

try {

// Task: Retrieve books published between 2010 and 2020,

// and the authors have written books with titles containing 'Programming'.

int minPublicationYear = 2010;

int maxPublicationYear = 2020;

String keyword = "Programming";

List<Book> books = getBooksByPublicationYearAndAuthorBookTitleKeyword(minPublicationYear, maxPublicationYear, keyword);

System.out.println("\nBooks published between 2010 and 2020, and the authors have written books with titles containing 'Programming':");

printBooks(books);

} finally {

// Close the sessionFactory to release resources

if (sessionFactory != null) {

sessionFactory.close();

}

}

}

public static List<Book> getBooksByPublicationYearAndAuthorBookTitleKeyword(int minPublicationYear, int maxPublicationYear, String keyword) {

Session session = sessionFactory.openSession();

CriteriaBuilder criteriaBuilder = session.getCriteriaBuilder();

CriteriaQuery<Book> criteriaQuery = criteriaBuilder.createQuery(Book.class);

Root<Book> root = criteriaQuery.from(Book.class);

// Join with the Author entity and add conditions for publication year range and title keyword

Join<Book, Author> authorJoin = root.join("author", JoinType.INNER);

Predicate publicationYearBetween = criteriaBuilder.between(root.get("publicationYear"), minPublicationYear, maxPublicationYear);

Predicate titleContainsKeyword = criteriaBuilder.like(root.get("title"), "%" + keyword + "%");

criteriaQuery.select(root)

.where(

publicationYearBetween,

titleContainsKeyword

);

List<Book> books = session.createQuery(criteriaQuery).getResultList();

session.close();

return books;

}

private static void printBooks(List<Book> books) {

for (Book book : books) {

System.out.println("Book ID: " + book.getId()

+ ", Title: " + book.getTitle()

+ ", Publication Year: " + book.getPublicationYear()

+ ", Author: " + book.getAuthor().getName());

}

}

}

1. Retrieve a list of authors who have written books with titles containing a specified keyword and were published within a given range of years. Additionally, include in the result the total number of books each author has written, the earliest and latest publication year of their books, and the average publication year of all their books. Sort the result based on the author's name in ascending order.

import org.hibernate.Session;

import org.hibernate.SessionFactory;

import org.hibernate.Transaction;

import org.hibernate.cfg.Configuration;

import javax.persistence.criteria.\*;

import java.util.List;

public class CriteriaApiExample {

private static SessionFactory sessionFactory;

static {

try {

// Initialize Hibernate SessionFactory

Configuration configuration = new Configuration().configure();

sessionFactory = configuration.buildSessionFactory();

} catch (Exception e) {

e.printStackTrace();

}

}

public static void main(String[] args) {

try {

// Task: Retrieve authors who have written books with titles containing 'Java',

// published between 2010 and 2020, and include additional information.

String keyword = "Java";

int minPublicationYear = 2010;

int maxPublicationYear = 2020;

List<Object[]> authorDetails = getAuthorsWithBookDetails(keyword, minPublicationYear, maxPublicationYear);

System.out.println("\nAuthors who have written books with titles containing 'Java', published between 2010 and 2020, with additional details:");

printAuthorsWithDetails(authorDetails);

} finally {

// Close the sessionFactory to release resources

if (sessionFactory != null) {

sessionFactory.close();

}

}

}

public static List<Object[]> getAuthorsWithBookDetails(String keyword, int minPublicationYear, int maxPublicationYear) {

Session session = sessionFactory.openSession();

CriteriaBuilder criteriaBuilder = session.getCriteriaBuilder();

CriteriaQuery<Object[]> criteriaQuery = criteriaBuilder.createQuery(Object[].class);

Root<Author> root = criteriaQuery.from(Author.class);

// Join with the Book entity and add conditions for title keyword and publication year range

Join<Author, Book> bookJoin = root.join("books", JoinType.INNER);

Predicate titleContainsKeyword = criteriaBuilder.like(bookJoin.get("title"), "%" + keyword + "%");

Predicate publicationYearBetween = criteriaBuilder.between(bookJoin.get("publicationYear"), minPublicationYear, maxPublicationYear);

// Select author details: [Author, Total Books, Earliest Publication Year, Latest Publication Year, Average Publication Year]

criteriaQuery.multiselect(

root,

criteriaBuilder.count(bookJoin),

criteriaBuilder.min(bookJoin.get("publicationYear")),

criteriaBuilder.max(bookJoin.get("publicationYear")),

criteriaBuilder.avg(bookJoin.get("publicationYear"))

)

.distinct(true) // To avoid duplicate authors if they have multiple books with the specified title keyword

.where(

titleContainsKeyword,

publicationYearBetween

)

.groupBy(root.get("id"))

.orderBy(criteriaBuilder.asc(root.get("name")));

List<Object[]> authorDetails = session.createQuery(criteriaQuery).getResultList();

session.close();

return authorDetails;

}

private static void printAuthorsWithDetails(List<Object[]> authors) {

for (Object[] authorData : authors) {

Author author = (Author) authorData[0];

Long totalBooks = (Long) authorData[1];

Integer earliestPublicationYear = (Integer) authorData[2];

Integer latestPublicationYear = (Integer) authorData[3];

Double averagePublicationYear = (Double) authorData[4];

System.out.println("Author ID: " + author.getId()

+ ", Name: " + author.getName()

+ ", Total Books Written: " + totalBooks

+ ", Earliest Publication Year: " + earliestPublicationYear

+ ", Latest Publication Year: " + latestPublicationYear

+ ", Average Publication Year: " + averagePublicationYear);

}

}

}

1. Retrieve a list of authors who have written books in a specified genre, and the books were published within a given range of years. Additionally, include in the result the total number of books each author has written, the total number of books in the specified genre, and the average publication year of their books in that genre. Sort the result based on the average publication year in ascending order.

import org.hibernate.Session;

import org.hibernate.SessionFactory;

import org.hibernate.Transaction;

import org.hibernate.cfg.Configuration;

import javax.persistence.criteria.\*;

import java.util.List;

public class CriteriaApiExample {

private static SessionFactory sessionFactory;

static {

try {

// Initialize Hibernate SessionFactory

Configuration configuration = new Configuration().configure();

sessionFactory = configuration.buildSessionFactory();

} catch (Exception e) {

e.printStackTrace();

}

}

public static void main(String[] args) {

try {

// Task: Retrieve authors who have written books in the 'Science Fiction' genre,

// published between 2010 and 2020, with additional details.

String genreName = "Science Fiction";

int minPublicationYear = 2010;

int maxPublicationYear = 2020;

List<Object[]> authorDetails = getAuthorsWithGenreDetails(genreName, minPublicationYear, maxPublicationYear);

System.out.println("\nAuthors who have written books in the 'Science Fiction' genre, published between 2010 and 2020, with additional details:");

printAuthorsWithGenreDetails(authorDetails);

} finally {

// Close the sessionFactory to release resources

if (sessionFactory != null) {

sessionFactory.close();

}

}

}

public static List<Object[]> getAuthorsWithGenreDetails(String genreName, int minPublicationYear, int maxPublicationYear) {

Session session = sessionFactory.openSession();

CriteriaBuilder criteriaBuilder = session.getCriteriaBuilder();

CriteriaQuery<Object[]> criteriaQuery = criteriaBuilder.createQuery(Object[].class);

Root<Author> authorRoot = criteriaQuery.from(Author.class);

Join<Author, Book> bookJoin = authorRoot.join("books", JoinType.INNER);

Join<Book, Genre> genreJoin = bookJoin.join("genre", JoinType.INNER);

// Add conditions for genre name and publication year range

Predicate genreEquals = criteriaBuilder.equal(genreJoin.get("name"), genreName);

Predicate publicationYearBetween = criteriaBuilder.between(bookJoin.get("publicationYear"), minPublicationYear, maxPublicationYear);

// Select author details: [Author, Total Books, Total Books in Genre, Average Publication Year]

criteriaQuery.multiselect(

authorRoot,

criteriaBuilder.count(bookJoin),

criteriaBuilder.sum(criteriaBuilder.selectCase().when(genreEquals, 1).otherwise(0)),

criteriaBuilder.avg(bookJoin.get("publicationYear"))

)

.groupBy(authorRoot.get("id"))

.having(

criteriaBuilder.greaterThan(criteriaBuilder.avg(bookJoin.get("publicationYear")), 0),

publicationYearBetween

)

.orderBy(criteriaBuilder.asc(criteriaBuilder.avg(bookJoin.get("publicationYear"))));

List<Object[]> authorDetails = session.createQuery(criteriaQuery).getResultList();

session.close();

return authorDetails;

}

private static void printAuthorsWithGenreDetails(List<Object[]> authors) {

for (Object[] authorData : authors) {

Author author = (Author) authorData[0];

Long totalBooks = (Long) authorData[1];

Long totalBooksInGenre = (Long) authorData[2];

Double averagePublicationYear = (Double) authorData[3];

System.out.println("Author ID: " + author.getId()

+ ", Name: " + author.getName()

+ ", Total Books Written: " + totalBooks

+ ", Total Books in Genre '" + genreName + "': " + totalBooksInGenre

+ ", Average Publication Year: " + averagePublicationYear);

}

}

}