Contents

Composite Keys

A composite key is a combination of two or more columns in a table that uniquely identifies each row.

Unlike a single-column primary key, a composite key relies on **multiple attributes** to ensure data integrity.

When to Use Composite Keys:

- No single attribute can uniquely identify a record.
- There's a natural relationship between multiple attributes.
- Performance optimization (in some cases).

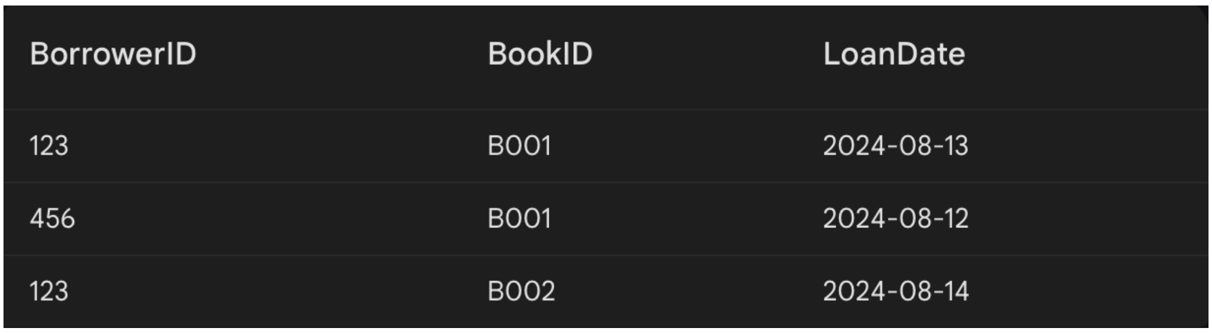
Key Points:

1. **Multiple columns:** Composed of two or more columns.
2. **Unique identifier:** Guarantees that every record is uniquely identifiable.
3. **Candidate key:** Can be chosen as a primary key from multiple candidate keys.
4. **Compound key:** A specific type of composite key where each component is a foreign key itself.

Example:

Consider a `Library` database with a `BookLoan` table.

- Columns: `BorrowerID`, `BookID`, `LoanDate`
- Composite primary key: `(BorrowerID, BookID, LoanDate)`
- This combination uniquely identifies a specific book being borrowed by a specific borrower on a particular date.



BorrowerID	BookID	LoanDate
123	B001	2024-08-13
456	B001	2024-08-12
123	B002	2024-08-14

In this example, `BorrowerID`, `BookID`, and `LoanDate` together form a composite primary key.

Using Composite Keys in CrudRepository in Spring Boot

While Spring Data JPA provides a convenient `CrudRepository` interface for basic CRUD operations, it primarily focuses on entities with a single-column primary key.

Handling composite keys requires additional considerations.

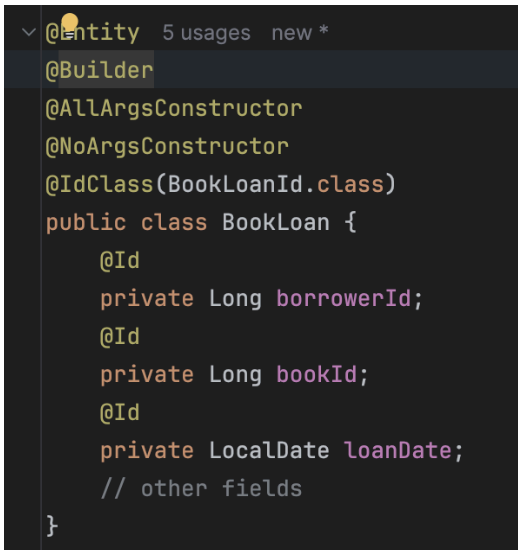
Key Considerations

- **Entity Annotation:**
    1. Use `@Entity` to mark your entity class.
    2. Use `@IdClass` or `@EmbeddedId` to define the composite key.
- **Repository Interface:**
    1. Extend `CrudRepository` and specify the Entity and Composite Key Type.
    2. Create custom query methods for specific operations.
- **Custom Query Methods:**
    1. Use Spring Data JPA's query for common operations.
    2. Create custom repositories for complex queries (JPQL or Native SQL)

Code Example:

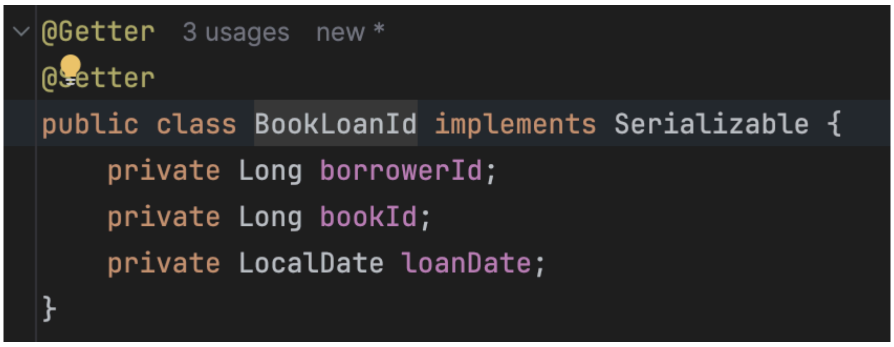
Let's consider a `BookLoan` entity with a composite key consisting of `borrowerId`, `bookId`, and `loanDate`.

1. The `BookLoan` entity uses `@IdClass` to define a custom identifier class `BookLoanId`.



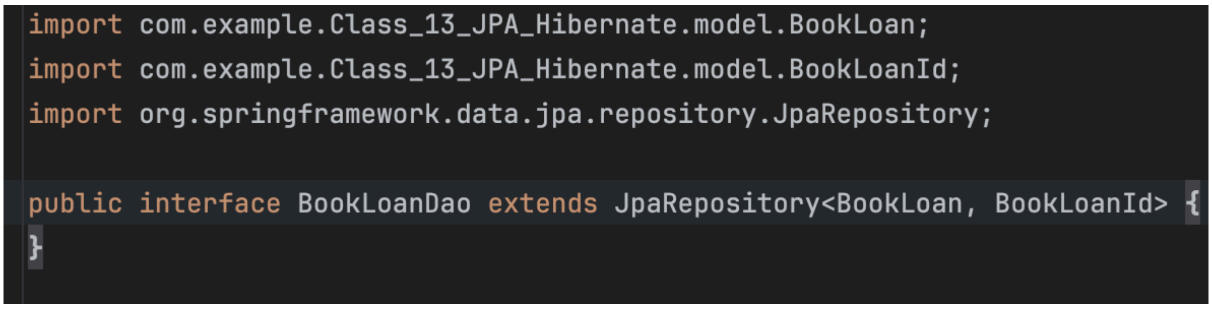
```java
@Entity  5 usages  new *
@Builder
@AllArgsConstructor
@NoArgsConstructor
@IdClass(BookLoanId.class)
public class BookLoan {
    @Id
    private Long borrowerId;
    @Id
    private Long bookId;
    @Id
    private LocalDate loanDate;
    // other fields
}
```

2. The `BookLoanId` class encapsulates the composite key fields.



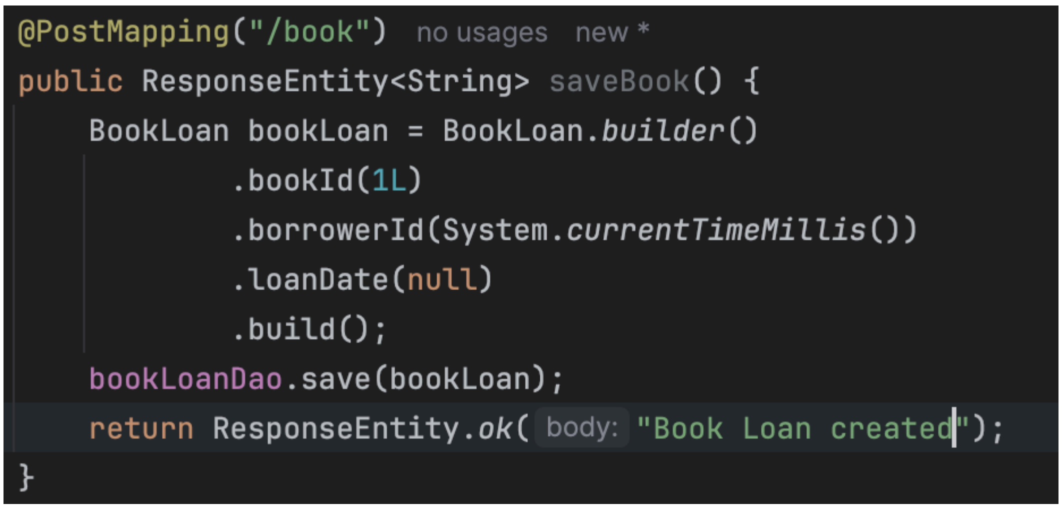
```java
@Getter  3 usages  new *
@Setter
public class BookLoanId implements Serializable {
    private Long borrowerId;
    private Long bookId;
    private LocalDate loanDate;
}
```

3. The `BookLoanRepository` extends `JpaRepository` with `BookLoan` as the entity and `BookLoanId` as the ID type.



```java
import com.example.Class_13_JPA_Hibernate.model.BookLoan;
import com.example.Class_13_JPA_Hibernate.model.BookLoanId;
import org.springframework.data.jpa.repository.JpaRepository;

public interface BookLoanDao extends JpaRepository<BookLoan, BookLoanId> {
}
```

Usage:



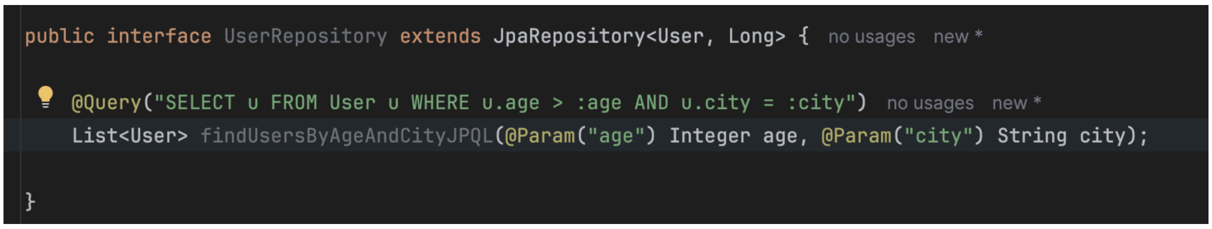
```
@PostMapping("/book")  no usages   new *
public ResponseEntity<String> saveBook() {
    BookLoan bookLoan = BookLoan.builder()
            .bookId(1L)
            .borrowerId(System.currentTimeMillis())
            .loanDate(null)
            .build();
    bookLoanDao.save(bookLoan);
    return ResponseEntity.ok( body: "Book Loan created");
}
```

Considerations:

1. java.sql.SQLIntegrityConstraintViolationException When any of the composite field is null
2. Consider indexing the composite key columns for improved query performance.
3. Evaluate if a composite key is the best approach or if normalisation can simplify the data structure.
4. For complex queries or specific operations, create custom query methods using Spring Data JPA's query derivation or native queries.

Custom Queries with JPQL (Java Persistence Query Language)

JPQL (Java Persistence Query Language) is a platform-independent object-oriented query language used to retrieve and manipulate data from a database using Java. It provides a powerful and flexible way to interact with your data without writing raw SQL.



```
public interface UserRepository extends JpaRepository<User, Long> {  no usages   new *

    @Query("SELECT u FROM User u WHERE u.age > :age AND u.city = :city")  no usages   new *
    List<User> findUsersByAgeAndCityJPQL(@Param("age") Integer age, @Param("city") String city);

}
```

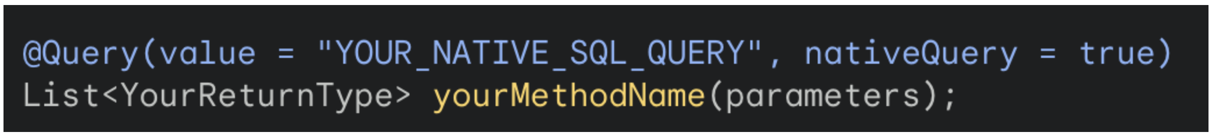
A JPQL query typically consists of:

- **SELECT clause:** Specifies the data to be retrieved.
- **FROM clause:** Specifies the entity or entities to query.
- **WHERE clause:** Specifies conditions for filtering data.

- **GROUP BY clause:** Groups rows based on specified columns.
- **HAVING clause:** Filters groups based on conditions.
- **ORDER BY clause:** Specifies the order of the result set.

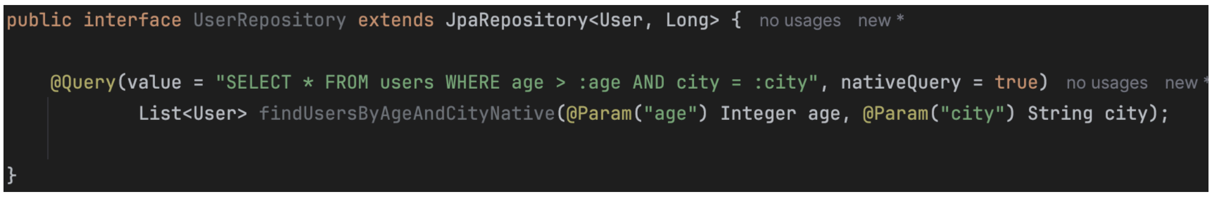
Custom Queries Using Native SQL

Native SQL queries provide a direct way to interact with your database, allowing you to leverage database-specific features and optimizations. Spring Data JPA provides a convenient mechanism to execute native SQL queries.



```
@Query(value = "YOUR_NATIVE_SQL_QUERY", nativeQuery = true)
List<YourReturnType> yourMethodName(parameters);
```

`@Query`: Specifies the query to be executed.
`value`: The actual SQL query.
`nativeQuery = true`: Indicates that the query is a native SQL query.
`YourReturnType`: The return type of the method, which can be a list of entities, a list of custom objects, or a primitive type.
`parameters`: Method parameters can be used to bind values to the query.



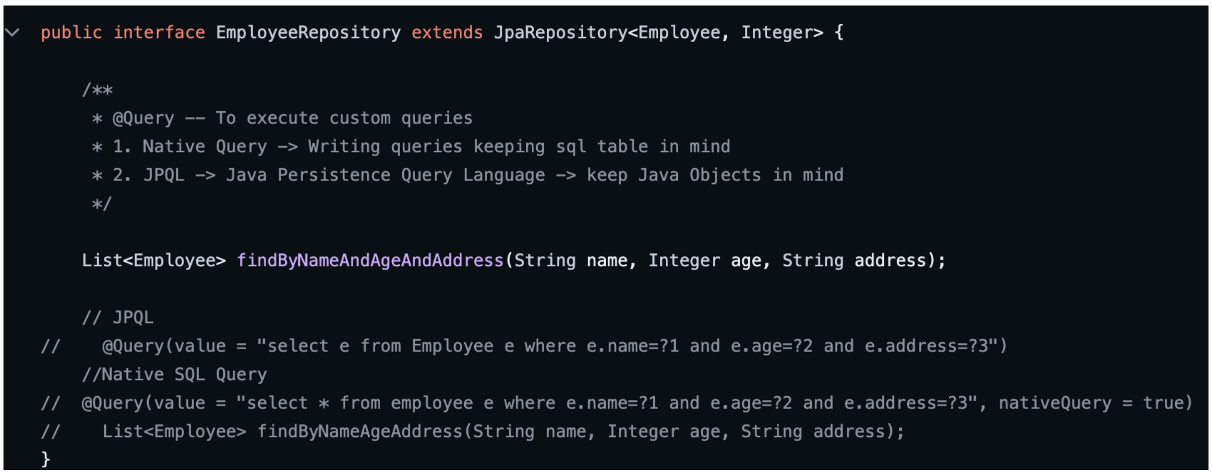
```
public interface UserRepository extends JpaRepository<User, Long> {

    @Query(value = "SELECT * FROM users WHERE age > :age AND city = :city", nativeQuery = true)
            List<User> findUsersByAgeAndCityNative(@Param("age") Integer age, @Param("city") String city);

}
```

How to use them?

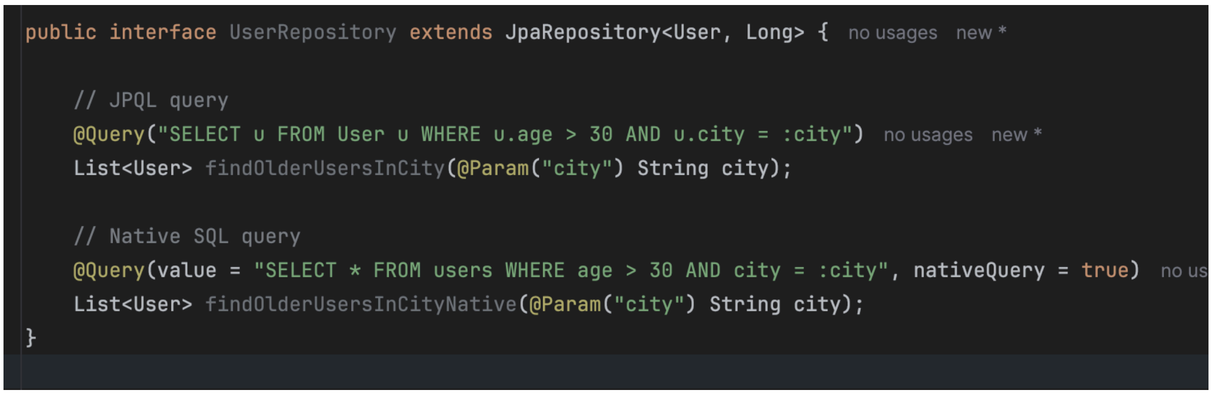
- Using `@Query` Annotation

This is the most common approach. You can annotate a method in your repository interface with `@Query` to specify the JPQL query or Native SQL query

Class - 14



```
public interface EmployeeRepository extends JpaRepository<Employee, Integer> {

    /**
     * @Query -- To execute custom queries
     * 1. Native Query -> Writing queries keeping sql table in mind
     * 2. JPQL -> Java Persistence Query Language -> keep Java Objects in mind
     */

    List<Employee> findByNameAndAgeAndAddress(String name, Integer age, String address);

    // JPQL
//    @Query(value = "select e from Employee e where e.name=?1 and e.age=?2 and e.address=?3")
    //Native SQL Query
//  @Query(value = "select * from employee e where e.name=?1 and e.age=?2 and e.address=?3", nativeQuery = true)
//    List<Employee> findByNameAgeAddress(String name, Integer age, String address);
}
```



```
public interface UserRepository extends JpaRepository<User, Long> {  no usages  new *

    // JPQL query
    @Query("SELECT u FROM User u WHERE u.age > 30 AND u.city = :city")  no usages  new *
    List<User> findOlderUsersInCity(@Param("city") String city);

    // Native SQL query
    @Query(value = "SELECT * FROM users WHERE age > 30 AND city = :city", nativeQuery = true)  no us
    List<User> findOlderUsersInCityNative(@Param("city") String city);
}
```

Minor Project (Digital Library)

0. Flowchart of the project

1. **Decide your entities:**
    1. Book
    2. Student
    3. Admin
    4. Author
    5. Request
    6. Transaction

2. Relationships between Entities : Relationships in SQL (Entity Mapping)

Multiple copies of the same book will have different ids.
One book can have only one author NOT multiple authors.
Students can not raise a request to return a book he does not have .
Student can not raise an issue request if he has the max limit of books already issued i.e. 3.
One Request can have only one book,one admin , one student
For every ADMIN APPROVED request there will be a transaction created after being processed by admin . Requests can be approved or denied but transactions will be created only if the Request is APPROVED by the admin. In the rejection case , we just update
the request status and add an admin comment in the request object .We will calculate the fine in Transaction .

3. **Functionalities for each Entity**

Let us now start coding in INTELLIJ
-dependencies
-application.properties
-create packages acc to mvc

--What is Enum ?
--What is @Enumerated ?
--What is foreign key ? Can it be null ?
https://www.geeksforgeeks.org/foreign-key-constraint-in-sql/
https://www.geeksforgeeks.org/how-to-create-a-table-with-a-foreign-key-in-sql/

--Let's discuss which table in the relationship should have the foreign key and which should have the back reference?
(IMP : class with back referencing is called referenced class ,class with foreign key is called a referencing class.)

--How to read relationship annotations in the Entity classes ? Many(Java class)ToOne(object property of the class)

- A Request and a Transaction are mapped OneToOne . Which one should be forign key ?
-What is an ER Model ?
Note : ER Model is used to model the logical view of the system from data perspective which consists of these components.
https://www.geeksforgeeks.org/er-diagram-of-a-company
https://www.geeksforgeeks.org/introduction-of-er-model/
-Post Data Modeling is completed , we can see the ER Model for our project in DBeaver. (Dot has foreign key of crystal.) , alternatively we can use dbdiagram.io
==
Data Modeling is complete , let's focus on developing API's now.

TASK : Let's start creating Controllers , Service and Repository class for functionality of students creating a request and assign an admin based on least request count .
NOTE : Students should not be able to return a book they have not issued .

Q. Now if I don't have the book with bookId as 40 and also don't have a student with studentId as 1 . What happens if I try to create a request from Postman using these bookId and studentId.What will happen ?

--Imp thing is to understand that first Validation checks (400) are happening and then foreign key validation will take place(500) .

TASK : Let's create controllers ,Service and Repository class for functionality of creating and fetching a Book and Student .
Note : We would not have a controller for Author as Author will be onboarded along with the book . So we will be creating an author at the time of creating the book .
Before creating an author , we have to check if it already exists in db.

NOTE :
1. If the author is not created, we need to create the author
2. Fetch the authorId and attach it in the book object
3. Save the book object

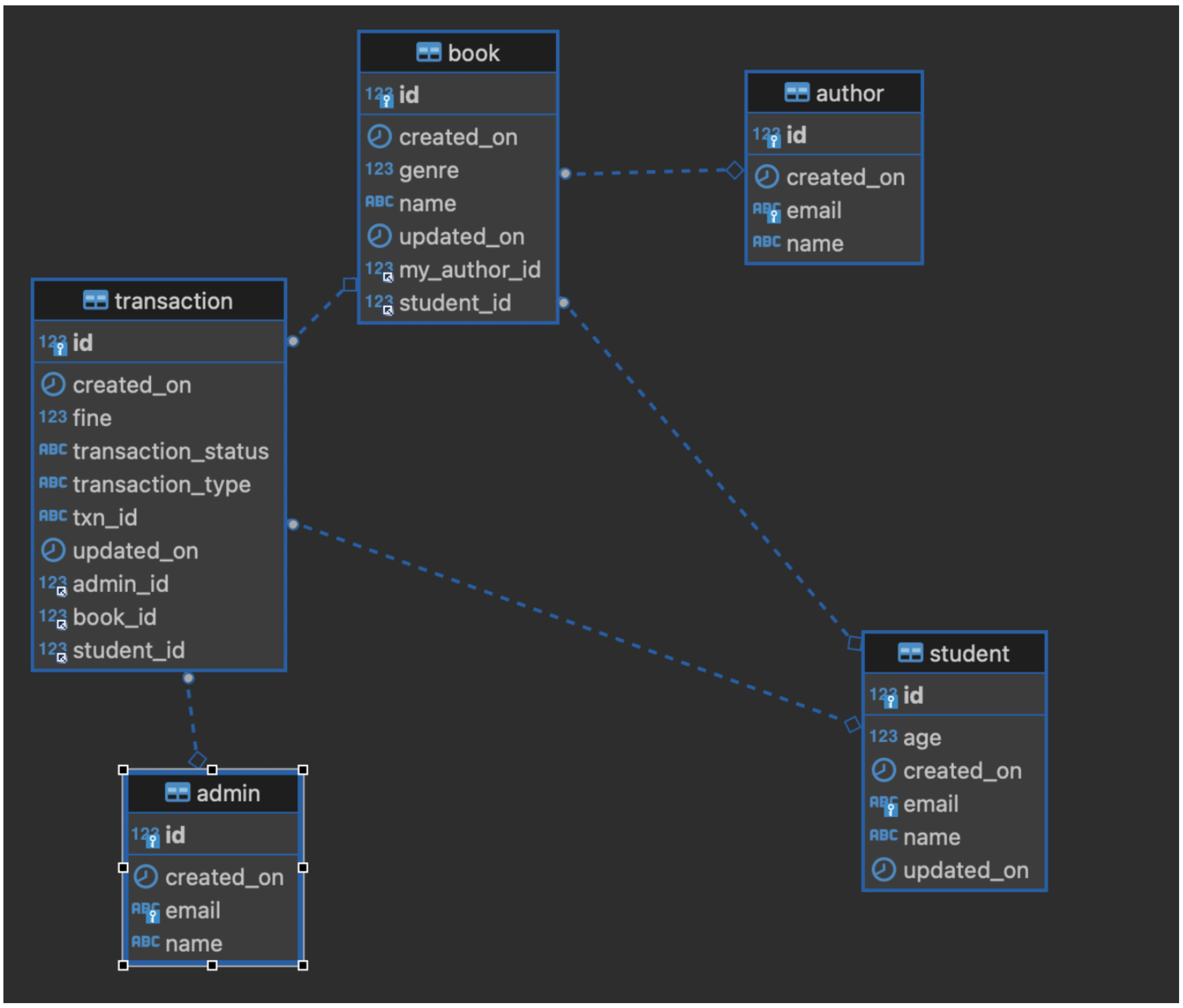
--Why do we need to (CRUD)Requests and Responses classes ?

--How to make an API request through terminal ?

-- When we fetch the details of Admin 1 , why does the json response go into an infinite loop and give a Null pointer exception in the logs ?
HINT : JsonIgnoreProperties("admin") put over requestList in the Admin class.

TASK : Now do JsonIgnoreProperties for all Entity classes to stop the echo.

Class - 14


book
id
created_on
genre
name
updated_on
my_author_id
student_id
author
id
created_on
email
name
transaction
id
created_on
fine
transaction_status
transaction_type
txn_id
updated_on
admin_id
book_id
student_id
student
id
age
created_on
email
name
updated_on
admin
id
created_on
email
name