Hibernate / JPA Overview

© luv2code LLC

Topics

- What is Hibernate?
- Benefits of Hibernate
- What is JPA?
- Benefits of JPA
- Code Snippets

© luv2code LLC

What is Hibernate?

- A framework for persisting / saving Java objects in a database
  - www.hibernate.org/orm

© luv2code LLC

Benefits of Hibernate

- Hibernate handles all of the low-level SQL
- Minimizes the amount of JDBC code you have to develop
- Hibernate provides the Object-to-Relational Mapping (ORM)

© luv2code LLC

Object-To-Relational Mapping (ORM)

- The developer defines mapping between Java class and database table

© luv2code LLC

What is JPA?

- **J**akarta **P**ersistence **A**PI (JPA) … *previously known as Java Persistence API*
  - Standard API for Object-to-Relational-Mapping (ORM)

- Only a specification
  - Defines a set of interfaces
  - Requires an implementation to be usable

www.luv2code.com/jpa-spec

© luv2code LLC

JPA - Vendor Implementations

www.luv2code.com/jpa-vendors

© luv2code LLC

What are Benefits of JPA

- By having a standard API, you are not locked to vendor's implementation
- Maintain portable, flexible code by coding to JPA spec (interfaces)
- Can theoretically switch vendor implementations
  - For example, if Vendor ABC stops supporting their product
  - You could switch to Vendor XYZ without vendor lock in

© luv2code LLC

JPA - Vendor Implementations

Can theoretically switch vendor implementations

© luv2code LLC

Saving a Java Object with JPA

Old days of JDBC ...
you would have to manually
write the SQL code

```
// create Java object
Student theStudent = new Student("Paul", "Doe", "paul@luv2code.com");

// save it to database
entityManager.persist(theStudent);
```

Special JPA helper object

The data will be stored in the database

SQL insert

© luv2code LLC

Retrieving a Java Object with JPA

```
// create Java object
Student theStudent = new Student("Paul", "Doe", "paul@luv2code.com");

// save it to database
entityManager.persist(theStudent);

// now retrieve from database using the primary key
int theId = 1;
Student myStudent = entityManager.find(Student.class, theId);
```

Query the database for given id

© luv2code LLC

Querying for Java Objects

```
TypedQuery<Student> theQuery = entityManager.createQuery("from Student", Student.class);

List<Student> students= theQuery.getResultList();
```

Returns a list of Student objects from the database

© luv2code LLC

JPA/Hibernate CRUD Apps

- Create objects
- Read objects
- Update objects
- Delete objects

© luv2code LLC

Hibernate / JPA and JDBC

© luv2code LLC

How does Hibernate / JPA relate to JDBC?

© luv2code LLC

Hibernate / JPA and JDBC

- Hibernate / JPA uses JDBC for all database communications

© luv2code LLC

MySQL Database

© luv2code LLC

MySQL Database

- In this course, we will use the MySQL Database

- MySQL includes two components
  - MySQL Database Server
  - MySQL Workbench

© luv2code LLC

MySQL Database Server

- The MySQL Database Server is the main engine of the database
- Stores data for the database
- Supports CRUD features on the data

© luv2code LLC

MySQL Workbench

- MySQL Workbench is a client GUI for interacting with the database
- Create database schemas and tables
- Execute SQL queries to retrieve data
- Perform insert, updates and deletes on data
- Handle administrative functions such as creating users
- Others …

© luv2code LLC

Install the MySQL software

- Step 1: Install MySQL Database Server
  - **`https://dev.mysql.com/downloads/mysql/`**
- Step 2: Install MySQL Workbench
  - **`https://dev.mysql.com/downloads/workbench/`**

Please install the MySQL software now

© luv2code LLC

Setup Database Table

© luv2code LLC

Two Database Scripts

1. Folder: **00-starter-sql-scripts**

   - **01-create-user.sql**
   - **02-student-tracker.sql**

© luv2code LLC

About: 01-create-user.sql

1. Create a new MySQL user for our application

   - user id: **springstudent**
   - password: **springstudent**

© luv2code LLC

MySQL Workbench

Local instance 3306

Administration | Schemas | Query 4 | 01-create-user | Context Help | Snippets

MANAGEMENT
- Server Status
- Client Connections
- Users and Privileges
- Status and System Variables
- Data Export
- Data Import/Restore

INSTANCE
- Startup / Shutdown
- Server Logs
- Options File

PERFORMANCE
- Dashboard
- Performance Reports
- Performance Schema Setup

Object Info | Session

No object selected

Limit to 1000 rows

```
1   -- Drop user first if they exist
2   DROP USER if exists 'springstudent'@'localhost' ;
3
4   -- Now create user with prop privileges
5   CREATE USER 'springstudent'@'localhost' IDENTIFIED BY 'springstudent';
6
7   GRANT ALL PRIVILEGES ON *.* TO 'springstudent'@'localhost';
```

user id

password

100% 69:5

Action Output

	Time	Action	Response	Duration / Fetch Time

Automatic context help is disabled. Use the toolbar to manually get help for the current caret position or to toggle automatic help.

About: 02-student-tracker.sql

1. Create a new database table: **student**

© luv2code LLC

Setting Up Spring Boot Project

© luv2code LLC

Automatic Data Source Configuration

- In Spring Boot, Hibernate is the default implementation of JPA
- `EntityManager` is main component for creating queries etc …
- `EntityManager` is from Jakarta Persistence API (JPA)

© luv2code LLC

Automatic Data Source Configuration

- Based on configs, Spring Boot will automatically create the beans:
  - **`DataSource, EntityManager,`** …

- You can then inject these into your app, for example your DAO

© luv2code LLC

Setting up Project with Spring Initialzr

- At Spring Initializr website, start.spring.io
- Add dependencies
  - MySQL Driver: **`mysql-connector-j`**
  - Spring Data JPA: **`spring-boot-starter-data-jpa`**

© luv2code LLC

Spring Boot - Auto configuration

- Spring Boot will automatically configure your data source for you

- Based on entries from Maven pom file
  - JDBC Driver: **`mysql-connector-j`**
  - Spring Data (ORM): **`spring-boot-starter-data-jpa`**

- DB connection info from **`application.properties`**

© luv2code LLC

application.properties

DB connection info

```
spring.datasource.url=jdbc:mysql://localhost:3306/student_tracker
spring.datasource.username=springstudent
spring.datasource.password=springstudent
```

No need to give JDBC driver class name
Spring Boot will automatically detect it based on URL

© luv2code LLC

Creating Spring Boot - Command Line App

- We will create a Spring Boot - Command Line App
- This will allow us to focus on Hibernate / JPA
- Later in the course, we will apply this to a CRUD REST API

© luv2code LLC

Creating Spring Boot - Command Line App

```
import org.springframework.boot.CommandLineRunner;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.context.annotation.Bean

@SpringBootApplication
public class CruddemoApplication {

   public static void main(String[] args) {
      SpringApplication.run(CruddemoApplication.class, args);
   }

   @Bean
   public CommandLineRunner commandLineRunner(String[] args) {
      return runner -> {
         System.out.println("Hello world");
      };
   }
}
```

Executed after the Spring Beans have been loaded

Lambda expression

Add our custom code

Command Line Runner & Application Runner

Spring Boot iki tane interface sunar. Bunlar CommandLineRunner ve ApplicationRunner interface'idir. Görevleri, Boot uygulaması tamamen başladığı andan sonra bir takım kodların çalıştırılmasını sağlar.

© luv2code LLC

JPA Development Process

© luv2code LLC

JPA Dev Process - To Do List

1. Annotate Java Class

2. Develop Java Code to perform database operations

© luv2code LLC

Let's just say "JPA"

- As mentioned, Hibernate is the default JPA implementation in Spring Boot

- Going forward in this course, I will simply use the term: **JPA**
  - Instead of saying "JPA Hibernate"

- We know that by default, Hibernate is used behind the scenes

© luv2code LLC

Terminology

Entity Class

Java class that is mapped to a database table

© luv2code LLC

Object-to-Relational Mapping (ORM)

Java Class

Database Table

© luv2code LLC

Entity Class

- At a minimum, the Entity class
  - Must be annotated with @Entity
  - Must have a public or protected no-argument constructor
    - The class can have other constructors

© luv2code LLC

Because it often happens that the JPA provider has to instantiate your domain object dynamically. It cannot do so, unless there is a no-arg constructor - it can't guess what the arguments should be.

In summary, when it is mentioned that the JPA provider needs a no-arg constructor to instantiate domain objects dynamically, it means that the provider relies on the ability to create objects at runtime without knowing the specific constructor arguments in advance. The no-arg constructor allows the JPA provider to create an instance of a class and then populate its fields appropriately.

Constructors in Java - Refresher

!!! Important !!!

- Remember about constructors in Java
- If you don't declare any constructors
  - Java will provide a no-argument constructor for free
- If you declare constructors with arguments
  - then you do NOT get a no-argument constructor for free
  - In this case, you have to explicitly declare a no-argument constructor

© luv2code LLC

Java Annotations

- Step 1: Map class to database table
- Step 2: Map fields to database columns

© luv2code LLC

Step 1: Map class to database table

```
@Entity
@Table(name="student")
public class Student {

  ...

}
```

Java Class

Database Table

© luv2code LLC

Step 2: Map fields to database columns

```
@Entity
@Table(name="student")
public class Student {

    @Id
    @Column(name="id")
    private int id;

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

© luv2code LLC

@Column - Optional

- Actually, the use of @Column is optional
- If not specified, the column name is the same name as Java field
- In general, I don't recommend this approach
  - If you refactor the Java code, then it will not match existing database columns
  - This is a breaking change and you will need to update database column
- Same applies to @Table, database table name is same as the class

© luv2code LLC

Terminology

Primary Key

Uniquely identifies each row in a table

Must be a unique value

Cannot contain NULL values

© luv2code LLC

MySQL - Auto Increment

```
CREATE TABLE student (

  id int NOT NULL AUTO_INCREMENT,
  first_name varchar(45) DEFAULT NULL,
  last_name varchar(45) DEFAULT NULL,
  email varchar(45) DEFAULT NULL,
  PRIMARY KEY (id)

)
```


© luv2code LLC

JPA Identity - Primary Key

```
@Entity
@Table(name="student")
public class Student {

  @Id
  @GeneratedValue(strategy=GenerationType.IDENTITY)
  @Column(name="id")
  private int id;

 ...
}
```


© luv2code LLC

ID Generation Strategies JPA

Name	Description
GenerationType.AUTO	Pick an appropriate strategy for the particular database
GenerationType.IDENTITY	Assign primary keys using database identity column
GenerationType.SEQUENCE	Assign primary keys using a database sequence
GenerationType.TABLE	Assign primary keys using an underlying database table to ensure uniqueness

© luv2code LLC

JPA'da ID GenerationType Değerleri - Otomatik ID Artırma Tipleri

JPA'da bir id sütunun otomatik artma modeli @GeneratedValue enumun, strategy özelliği ile verilmektedir. Aşağıdaki değerleri alabilir:

- GenerationType.AUTO : Veritabanın varsayılan id artırma yöntemini kullanılır
- GenerationType.IDENTITY : Otomatik artan id sütunu kullanır
- GenerationType.SEQUENCE : Sequence (id değerlerinin ayrı bir sequence ile artırılması) kullanır
- GenerationType.TABLE : Id artırımı için bir tablo kullanılır ve id değerleri orada tutulur

Bonus Bonus

- You can define your own CUSTOM generation strategy :-)
- Create implementation of **`org.hibernate.id.IdentifierGenerator`**
- Override the method: **`public Serializable generate(…)`**

© luv2code LLC

Save a Java Object

© luv2code LLC

Sample App Features

- **Create** a new Student
- **Read** a Student
- **Update** a Student
- **Delete** a Student

CRUD

© luv2code LLC

Student Data Access Object

- Responsible for interfacing with the database
- This is a common design pattern: **Data Access Object (DAO)**

© luv2code LLC

Student Data Access Object

Methods
save(…)
findById(…)
findAll()
findByLastName(…)
update(…)
delete(…)
deleteAll()

© luv2code LLC

Student Data Access Object

- Our DAO needs a JPA Entity Manager
- JPA Entity Manager is the main component for saving/retrieving entities

© luv2code LLC

JPA Entity Manager

- Our JPA Entity Manager needs a Data Source
- The Data Source defines database connection info
- JPA Entity Manager and Data Source are automatically created by Spring Boot
  - Based on the file: application.properties (JDBC URL, user id, password, etc …)
- We can autowire/inject the JPA Entity Manager into our Student DAO

© luv2code LLC

Student DAO

Step-By-Step

- Step 1: Define DAO interface
- Step 2: Define DAO implementation
  - Inject the entity manager
- Step 3: Update main app

© luv2code LLC

Step 1: Define DAO interface

```
import com.luv2code.cruddemo.entity.Student;

public interface StudentDAO {

    void save(Student theStudent);

}
```

© luv2code LLC

Step 2: Define DAO implementation

```
import com.luv2code.cruddemo.entity.Student;
import jakarta.persistence.EntityManager;
import org.springframework.beans.factory.annotation.Autowired;

public class StudentDAOImpl implements StudentDAO {

    private EntityManager entityManager;

    @Autowired
    public StudentDAOImpl(EntityManager theEntityManager) {
        entityManager = theEntityManager;
    }

    @Override
    public void save(Student theStudent) {
        entityManager.persist(theStudent);
    }

}
```

Inject the Entity Manager

Save the Java object

© luv2code LLC

Spring @Transactional

- Spring provides an **@Transactional** annotation

- **Automagically** begin and end a transaction for your JPA code
  - No need for you to explicitly do this in your code

- This Spring **magic** happens behind the scenes

© luv2code LLC

Step 2: Define DAO implementation

```
import com.luv2code.cruddemo.entity.Student;
import jakarta.persistence.EntityManager;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.transaction.annotation.Transactional;

public class StudentDAOImpl implements StudentDAO {

    private EntityManager entityManager;

    @Autowired
    public StudentDAOImpl(EntityManager theEntityManager) {
        entityManager = theEntityManager;
    }

    @Override
    @Transactional
    public void save(Student theStudent) {
        entityManager.persist(theStudent);
    }

}
```

Handles transaction management

© luv2code LLC

Specialized Annotation for DAOs

- Spring provides the **@Repository** annotation

© luv2code LLC

The @Repository annotation is used in Spring Framework to indicate that a class is a repository component.

In the context of the Model-View-Controller (MVC) architectural pattern, a repository is responsible for handling data access and storage operations. It typically interacts with the underlying data source, such as a database, to perform CRUD (Create, Read, Update, Delete) operations on entities.

By applying the @Repository annotation to a class, you provide a hint to the Spring container that the class should be registered as a bean and be available for dependency injection. This allows other components, such as services or controllers, to depend on the repository and use its functionality

The @Repository annotation is a specialization of the more general @Component annotation in Spring. It adds an extra layer of semantic meaning to the class, indicating that it specifically deals with data access concerns. This can be helpful for code organization and clarity, as well as providing a clear separation of responsibilities within your application.

When you annotate a class with @Repository, you can then use it as a dependency in other components by using annotations like @Autowired or constructor injection to have the Spring container automatically inject an instance of the repository into the dependent class.

In summary, the @Repository annotation is used in Spring to mark a class as a repository component, indicating that it handles data access operations. It enables automatic registration and dependency injection of the repository class, making it easier to use and manage within your application.

Specialized Annotation for DAOs

- Applied to DAO implementations
- Spring will automatically register the DAO implementation
  - thanks to component-scanning
- Spring also provides translation of any JDBC related exceptions

© luv2code LLC

Step 2: Define DAO implementation

Specialized annotation for repositories

Supports component scanning

Translates JDBC exceptions

```
import com.luv2code.cruddemo.entity.Student;
import jakarta.persistence.EntityManager;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Repository;
import org.springframework.transaction.annotation.Transactional;

@Repository
public class StudentDAOImpl implements StudentDAO {

    private EntityManager entityManager;

    @Autowired
    public StudentDAOImpl(EntityManager theEntityManager) {
        entityManager = theEntityManager;
    }

    @Override
    @Transactional
    public void save(Student theStudent) {
        entityManager.persist(theStudent);
    }

}
```


© luv2code LLC

Step 3: Update main app

```
@SpringBootApplication
public class CruddemoApplication {

   public static void main(String[] args) {
      SpringApplication.run(CruddemoApplication.class, args);
   }

   @Bean
   public CommandLineRunner commandLineRunner(StudentDAO studentDAO) {
      return runner -> {

         createStudent(studentDAO);
   }

   private void createStudent(StudentDAO studentDAO) {

      // create the student object
      System.out.println("Creating new student object...");
      Student tempStudent = new Student("Paul", "Doe", "paul@luv2code.com");

      // save the student object
      System.out.println("Saving the student...");
      studentDAO.save(tempStudent);

      // display id of the saved student
      System.out.println("Saved student. Generated id: " + tempStudent.getId());
   }

}
```

Inject the StudentDAO

© luv2code LLC

FAQ: How do I CHANGE the AUTO_INCREMENT values?

ALTER TABLE student_tracker.student auto_increment=3000

Result Grid | Filter Rows: | Edit:

id	first_name	last_name	email
2	Paul	Doe	paul@luv2code.com
3	Paul	Doe	paul@luv2code.com
4	John	Doe	john@luv2code.com
5	Mary	Public	mary@luv2code.com
6	Bonita	Applebum	bonita@luv2code.com
3000	John	Doe	john@luv2code.com
3001	Mary	Public	mary@luv2code.com
3002	Bonita	Applebum	bonita@luv2code.com
NULL	NULL	NULL	NULL

FAQ: How do I RESET the AUTO_INCREMENT values to 1?

```
TRUNCATE student_tracker.student;
```

Result Grid | Filter Rows: | Edit:

id	first_name	last_name	email
1	John	Doe	john@luv2code.com
2	Mary	Public	mary@luv2code.com
3	Bonita	Applebum	bonita@luv2code.com
NULL	NULL	NULL	NULL

Retrieving an Object

© luv2code LLC

JPA CRUD Apps

- Create objects
- Read objects
- Update objects
- Delete objects

© luv2code LLC

Retrieving a Java Object with JPA

```
// retrieve/read from database using the primary key
// in this example, retrieve Student with primary key: 1

Student myStudent = entityManager.find(Student.class, 1);
```

If not found, returns null

Entity class

Primary key

© luv2code LLC

Development Process

Step-By-Step

1. Add new method to DAO interface

2. Add new method to DAO implementation

3. Update main app

© luv2code LLC

Step 1: Add new method to DAO interface

```
import com.luv2code.cruddemo.entity.Student;

public interface StudentDAO {
    …

    Student findById(Integer id);

}
```


© luv2code LLC

Step 2: Define DAO implementation

```
import com.luv2code.cruddemo.entity.Student;
import jakarta.persistence.EntityManager;
…

public class StudentDAOImpl implements StudentDAO {

    private EntityManager entityManager;
    …

    @Override
    public Student findById(Integer id) {
        return entityManager.find(Student.class, id);
    }


}
```

No need to add @Transactional since we are doing a query

If not found, returns null

Entity class

Primary key

© luv2code LLC

Step 3: Update main app

```java
@SpringBootApplication
public class CruddemoApplication {
   …

   @Bean
   public CommandLineRunner commandLineRunner(StudentDAO studentDAO) {
      return runner -> {

         readStudent(studentDAO);
      };
   }

   …
```

```java
private void readStudent(StudentDAO studentDAO) {
   // create a student object
   System.out.println("Creating new student object...");
   Student tempStudent = new Student("Daffy", "Duck", "daffy@luv2code.com");

   // save the student object
   System.out.println("Saving the student...");
   studentDAO.save(tempStudent);

   // display id of the saved student
   System.out.println("Saved student. Generated id: " + tempStudent.getId());

   // retrieve student based on the id: primary key
   System.out.println("\nRetrieving student with id: " + tempStudent.getId());

   Student myStudent = studentDAO.findById(tempStudent.getId());

   System.out.println("Found the student: " + myStudent);
}
```


© luv2code LLC

Query Objects

© luv2code LLC

JPA CRUD Apps

- Create objects
- Read objects
- Update objects
- Delete objects

© luv2code LLC

JPA Query Language (JPQL)

- Query language for retrieving objects

- Similar in concept to SQL
  - where, like, order by, join, in, etc…

- However, JPQL is based on **entity name** and **entity fields**

© luv2code LLC

Retrieving all Students

Name of JPA Entity … the class name

```
TypedQuery<Student> theQuery = entityManager.createQuery("FROM Student", Student.class);

List<Student> students = theQuery.getResultList();
```

Note: this is NOT the name of the database table

All JPQL syntax is based on entity name and entity fields

© luv2code LLC

Retrieving Students: `lastName = 'Doe'`

(not column name of db)

Field of JPA Entity

```
TypedQuery<Student> theQuery = entityManager.createQuery(
                                    "FROM Student WHERE lastName='Doe'", Student.class);

List<Student> students = theQuery.getResultList();
```

© luv2code LLC

Retrieving Students using OR predicate:

(not column name of db)

Field of JPA Entity

```
TypedQuery<Student> theQuery = entityManager.createQuery(
                        "FROM Student WHERE lastName='Doe' OR firstName='Daffy'", Student.class);

List<Student> students = theQuery.getResultList();
```

Field of JPA Entity

(not column name of db)

Java Class

Student
- id : int - firstName : String - lastName : String - email : String
...

© luv2code LLC

Retrieving Students using LIKE predicate:

```
TypedQuery<Student> theQuery = entityManager.createQuery(
                        "FROM Student WHERE email LIKE '%luv2code.com'", Student.class);

List<Student> students = theQuery.getResultList();
```

Match of email addresses that ends with `luv2code.com`

© luv2code LLC

JPQL - Named Parameters

```
public List<Student> findByLastName(String theLastName) {

    TypedQuery<Student> theQuery = entityManager.createQuery(
                                        "FROM Student WHERE lastName=:theData", Student.class);

    theQuery.setParameter("theData", theLastName);

    return theQuery.getResultList();
}
```

JPQL Named Parameters are prefixed with a colon :

Think of this as a placeholder that is filled in later

© luv2code LLC

Development Process

Step-By-Step

1. Add new method to DAO interface

2. Add new method to DAO implementation

3. Update main app

© luv2code LLC

Step 1: Add new method to DAO interface

```
import com.luv2code.cruddemo.entity.Student;
import java.util.List;

public interface StudentDAO {
    ...

    List<Student> findAll();

}
```


© luv2code LLC

Step 2: Define DAO implementation

```
import com.luv2code.cruddemo.entity.Student;
import jakarta.persistence.EntityManager;
import jakarta.persistence.TypedQuery;
import java.util.List;
…

public class StudentDAOImpl implements StudentDAO {

    private EntityManager entityManager;
    …

    @Override
    public List<Student> findAll() {
        TypedQuery<Student> theQuery = entityManager.createQuery("FROM Student", Student.class);
        return theQuery.getResultList();
    }

}
```

No need to add @Transactional since we are doing a query

Name of JPA Entity

© luv2code LLC

Step 3: Update main app

```java
@SpringBootApplication
public class CruddemoApplication {

   public static void main(String[] args) {
      SpringApplication.run(CruddemoApplication.class, args);
   }

   @Bean
   public CommandLineRunner commandLineRunner(StudentDAO studentDAO) {
      return runner -> {

         queryForStudents(studentDAO);
      };
   }

   private void queryForStudents(StudentDAO studentDAO) {

      // get list of students
      List<Student> theStudents = studentDAO.findAll();

      // display list of students
      for (Student tempStudent : theStudents) {
         System.out.println(tempStudent);
      }
   }
}
```


© luv2code LLC

Updating an Object

© luv2code LLC

JPA CRUD Apps

- Create objects
- Read objects
- Update objects
- Delete objects

© luv2code LLC

Update a Student

```
Student theStudent = entityManager.find(Student.class, 1);

// change first name to "Scooby"
theStudent.setFirstName("Scooby");

entityManager.merge(theStudent);
```

setter

Update the entity

© luv2code LLC

Update last name for all students

Field of JPA Entity

```
int numRowsUpdated = entityManager.createQuery(
                        "UPDATE Student SET lastName='Tester'")
                        .executeUpdate();
```

Name of JPA Entity … the class name

Return the number of rows updated

Execute this statement

© luv2code LLC

Development Process

Step-By-Step

1. Add new method to DAO interface

2. Add new method to DAO implementation

3. Update main app

© luv2code LLC

Step 1: Add new method to DAO interface

```
import com.luv2code.cruddemo.entity.Student;

public interface StudentDAO {
    …

    void update(Student theStudent);

}
```


© luv2code LLC

Step 2: Define DAO implementation

```
import com.luv2code.cruddemo.entity.Student;
import jakarta.persistence.EntityManager;
import org.springframework.transaction.annotation.Transactional;
…

public class StudentDAOImpl implements StudentDAO {

    private EntityManager entityManager;
    …

    @Override
    @Transactional
    public void update(Student theStudent) {
        entityManager.merge(theStudent);
    }

}
```

Add @Transactional since we are performing an update

© luv2code LLC

Step 3: Update main app

```java
@SpringBootApplication
public class CruddemoApplication {
   …

   @Bean
   public CommandLineRunner commandLineRunner(StudentDAO studentDAO) {
      return runner -> {

         updateStudent(studentDAO);
      };
   }

   …
```

```java
private void updateStudent(StudentDAO studentDAO) {

   // retrieve student based on the id: primary key
   int studentId = 1;
   System.out.println("Getting student with id: " + studentId);

   Student myStudent = studentDAO.findById(studentId);

   System.out.println("Updating student...");

   // change first name to "Scooby"
   myStudent.setFirstName("Scooby");
   studentDAO.update(myStudent);

   // display updated student
   System.out.println("Updated student: " + myStudent);
}
```


© luv2code LLC

Deleting an Object

© luv2code LLC

JPA CRUD Apps

- Create objects
- Read objects
- Update objects
- Delete objects

© luv2code LLC

Delete a Student

```
// retrieve the student
int id = 1;
Student theStudent = entityManager.find(Student.class, id);

// delete the student
entityManager.remove(theStudent);
```


© luv2code LLC

Delete based on a condition

Delete single student

```
int numRowsDeleted = entityManager.createQuery(
                "DELETE FROM Student WHERE lastName='Smith'")
                .executeUpdate();
```

Field of JPA Entity

Name of JPA Entity … the class name

Return the number of rows deleted

Execute this statement

Method name "Update" is a generic term

We are "modifying" the database

Java Class

Student
- id : int - firstName : String - lastName : String - email : String
...

© luv2code LLC

Delete All Students

```
int numRowsDeleted = entityManager
                        .createQuery("DELETE FROM Student")
                        .executeUpdate();
```

© luv2code LLC

Development Process

Step-By-Step

1. Add new method to DAO interface

2. Add new method to DAO implementation

3. Update main app

© luv2code LLC

Step 1: Add new method to DAO interface

```
import com.luv2code.cruddemo.entity.Student;

public interface StudentDAO {
    …

    void delete(Integer id);

}
```


© luv2code LLC

Step 2: Define DAO implementation

```
import com.luv2code.cruddemo.entity.Student;
import jakarta.persistence.EntityManager;
import org.springframework.transaction.annotation.Transactional;
…

public class StudentDAOImpl implements StudentDAO {

    private EntityManager entityManager;
    …

    @Override
    @Transactional
    public void delete(Integer id) {
        Student theStudent = entityManager.find(Student.class, id);
        entityManager.remove(theStudent);
    }

}
```

Add @Transactional since we are performing a delete

© luv2code LLC

Step 3: Update main app

```
@SpringBootApplication
public class CruddemoApplication {
  …

  @Bean
  public CommandLineRunner commandLineRunner(StudentDAO studentDAO) {
    return runner -> {

      deleteStudent(studentDAO);
    };
  }

  …
```

```
private void deleteStudent(StudentDAO studentDAO) {
  // delete the student
  int studentId = 3;

  System.out.println("Deleting student id: " + studentId);

  studentDAO.delete(studentId);
}
```


© luv2code LLC

Create Database Tables from Java Code

© luv2code LLC

Create database tables: `student`

- Previously, we created database tables by running a SQL script

© luv2code LLC

Create database tables: `student`

- JPA/Hibernate provides an option to automagically create database tables
- Creates tables based on Java code with JPA/Hibernate annotations
- Useful for development and testing

© luv2code LLC

Configuration

- In Spring Boot configuration file: **`application.properties`**

```
spring.jpa.hibernate.ddl-auto=create
```

- When you run your app, JPA/Hibernate will **drop** tables then **create** them
- Based on the JPA/Hibernate annotations in your Java code

ddl: Data Definition Language

© luv2code LLC

Creating Tables based on Java Code

Hibernate will generate and execute this

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

  @Id
  @GeneratedValue(strategy=GenerationType.IDENTITY)
  @Column(name="id")
  private int id;

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

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

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

  …
  // constructors, getters / setters
}
```

1

2

```java
create table student (id integer not null auto_increment,
                      email varchar(255), first_name varchar(255),
                      last_name varchar(255), primary key (id))
```

© luv2code LLC

Configuration - application.properties

```
spring.jpa.hibernate.ddl-auto=PROPERTY-VALUE
```

Property Value	Property Description
none	No action will be performed
create-only	Database tables are only created
drop	Database tables are dropped
create	Database tables are dropped followed by database tables creation
create-drop	Database tables are dropped followed by database tables creation. On application shutdown, drop the database tables
validate	Validate the database tables schema
update	Update the database tables schema

When database tables are dropped, all data is lost

© luv2code LLC

Basic Projects

for ease of development and testing

- For basic projects, can use auto configuration

```
spring.jpa.hibernate.ddl-auto=create
```

- Database tables are **dropped** first and then **created** from scratch

Note:
When database tables are dropped, all data is lost

© luv2code LLC

Basic Projects

- If you want to create tables once … and then keep data, use: `update`

```
spring.jpa.hibernate.ddl-auto=update
```

- However, will ALTER database schema based on latest code updates
- Be VERY careful here … only use for basic projects

© luv2code LLC

Warning

```
spring.jpa.hibernate.ddl-auto=create
```

- Don't do this on **Production** databases!!!

- You don't want to **drop** your Production data
  - **All data is deleted!!!**

- Instead for Production, you should have DBAs run SQL scripts

© luv2code LLC

Use Case

```
spring.jpa.hibernate.ddl-auto=create
```

- Automatic table generation is useful for
  - Database integration testing with in-memory databases
  - Basic, small hobby projects

auto-generating is not recommended for enterprise, real-time projects

© luv2code LLC

Recommendation

- In general, I don't recommend auto generation for enterprise, real-time projects
  - You can VERY easily drop PRODUCTION data if you are not careful

- I recommend SQL scripts
  - Corporate DBAs prefer SQL scripts for governance and code review
  - The SQL scripts can be customized and fine-tuned for complex database designs
  - The SQL scripts can be version-controlled
  - Can also work with schema migration tools such as Liquibase and Flyway

© luv2code LLC

```
# Turn off the Spring Boot banner
spring.main.banner-mode=off

# Reduce logging level. Set logging le
logging.level.root=warn

# Add logging configs to display SQL statements
logging.level.org.hibernate.SQL=debug
logging.level.org.hibernate.orm.jdbc.bind=trace
```

Log SQL statements

Log values for SQL statements