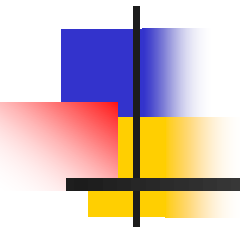
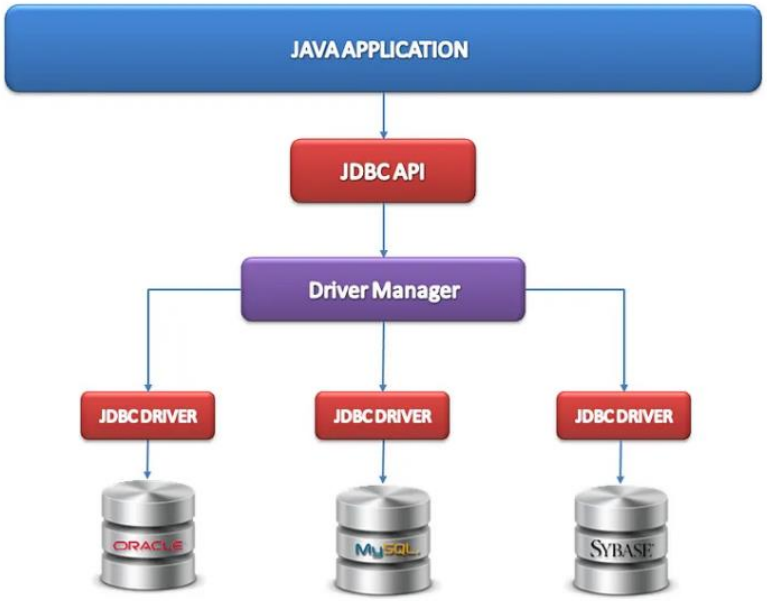
Advanced Application Development (CSE-214) week-8 :Hibernate / JPA



Dr. Alper ÖZCAN
Akdeniz University
alper.ozcan@gmail.com

What is JDBC (Java Database Connectivity)?

JDBC (Java Database Connectivity) is an API in Java that allows Java applications to connect to and interact with relational databases.



It provides methods to:

- Connect to a database
- Execute SQL queries and updates
- Retrieve and process the results

Key Components of JDBC:

- **DriverManager**: Manages the set of JDBC drivers.
- **Connection**: Interface to establish a session with the database.
- **Statement / PreparedStatement**: Used to execute SQL queries.
- **ResultSet**: Holds the result of a query.

JDBC is a lower-level API than **JPA**. Frameworks like **Hibernate** or Spring Data **JPA** internally use JDBC

JDBC Example

```java
import java.sql.*;

public class JdbcExample {
    public static void main(String[] args) {
        String url = "jdbc:mysql://localhost:3306/school";
        String user = "root";
        String password = "yourPassword";

        try {
            // 1. Load the MySQL JDBC driver (optional with modern JDBC)
            Class.forName("com.mysql.cj.jdbc.Driver");

            // 2. Create a connection
            Connection conn = DriverManager.getConnection(url, user, password);

            // 3. Create a statement
            Statement stmt = conn.createStatement();

            // 4. Execute a query
            ResultSet rs = stmt.executeQuery("SELECT * FROM students");

            // 5. Process the result set
            while (rs.next()) {
                int id = rs.getInt("id");
                String name = rs.getString("name");
                System.out.println("ID: " + id + ", Name: " + name);
            }

            // 6. Close resources
            rs.close();
            stmt.close();
            conn.close();

        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

What is Hibernate?

•A framework for persisting and saving Java objects in a database

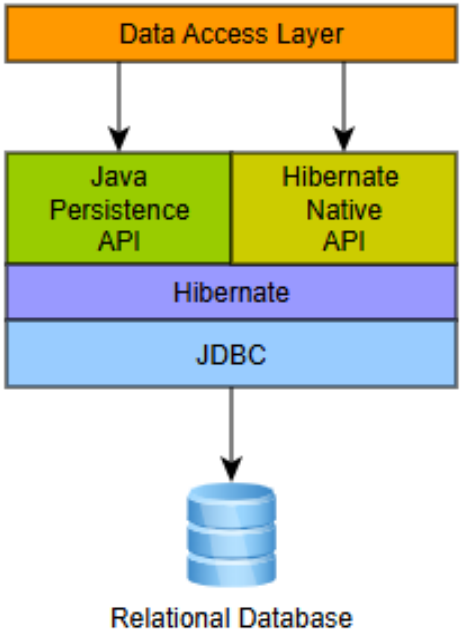
•Handles all low-level SQL operations

•Minimizes the amount of JDBC code developers need to write

•Provides Object-to-Relational Mapping (ORM)

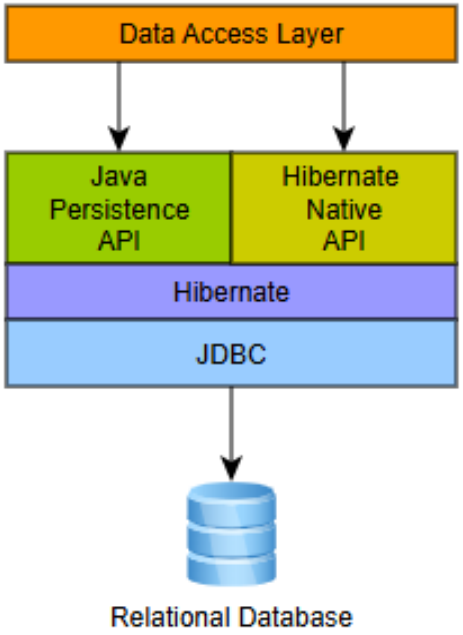
•Hibernate is an Object/Relational Mapping solution for Java environments.

•Object/Relational Mapping refers to the technique of mapping data from an object model representation to a relational data model representation



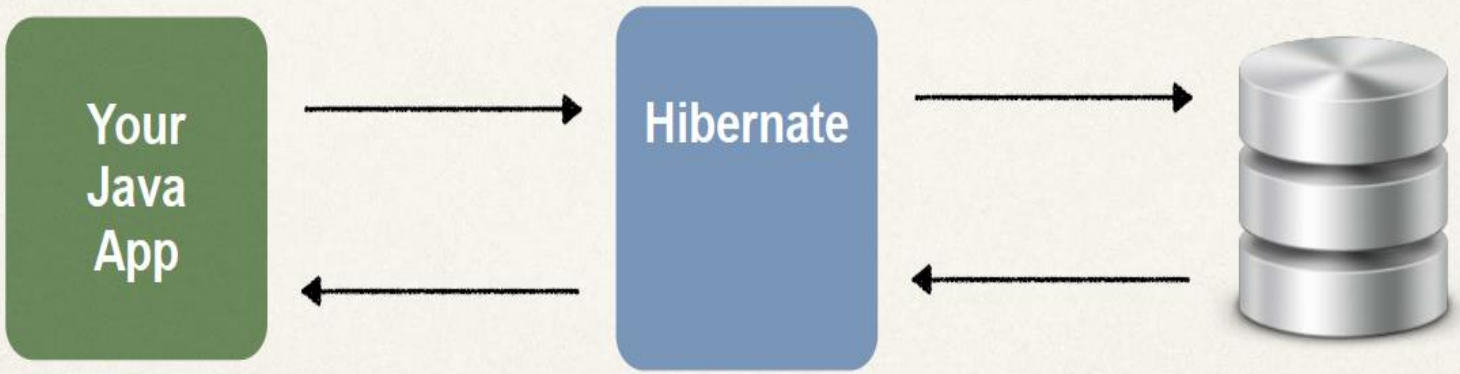
Java Persistence API (JPA)

- Standard API for ORM
- Defines a set of interfaces
- Requires an implementation
- JPA Vendor Implementations: Hibernate
- Write portable, flexible code



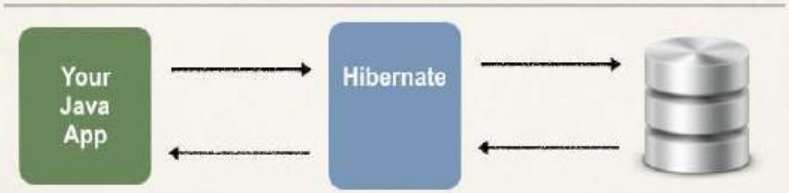
What is Hibernate?

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



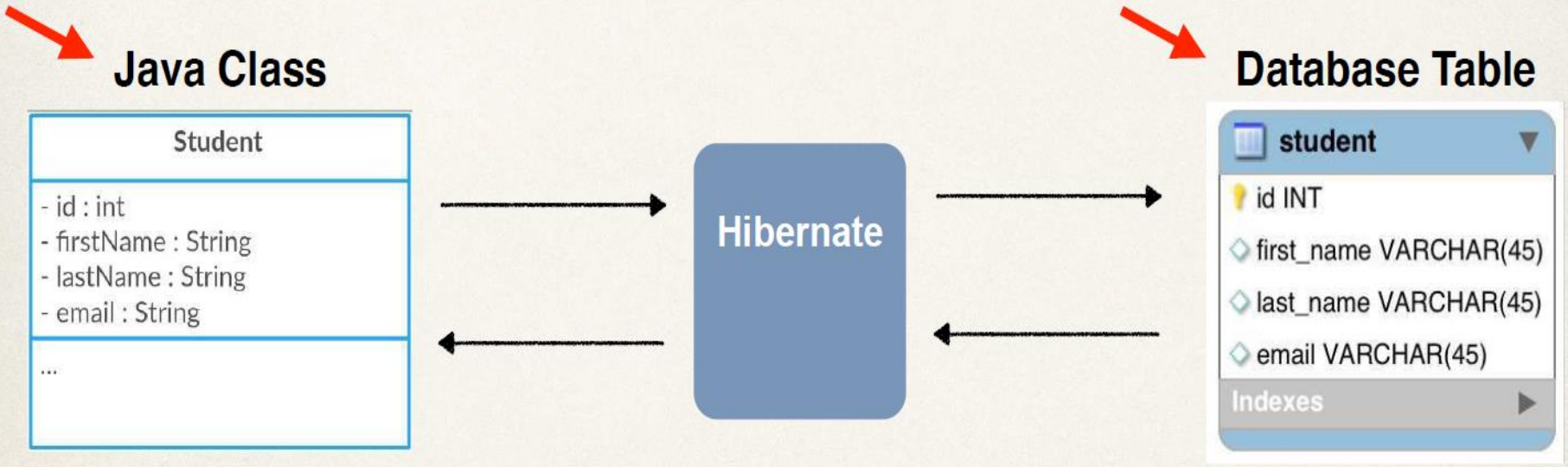
Benefits of Hibernate

- 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)

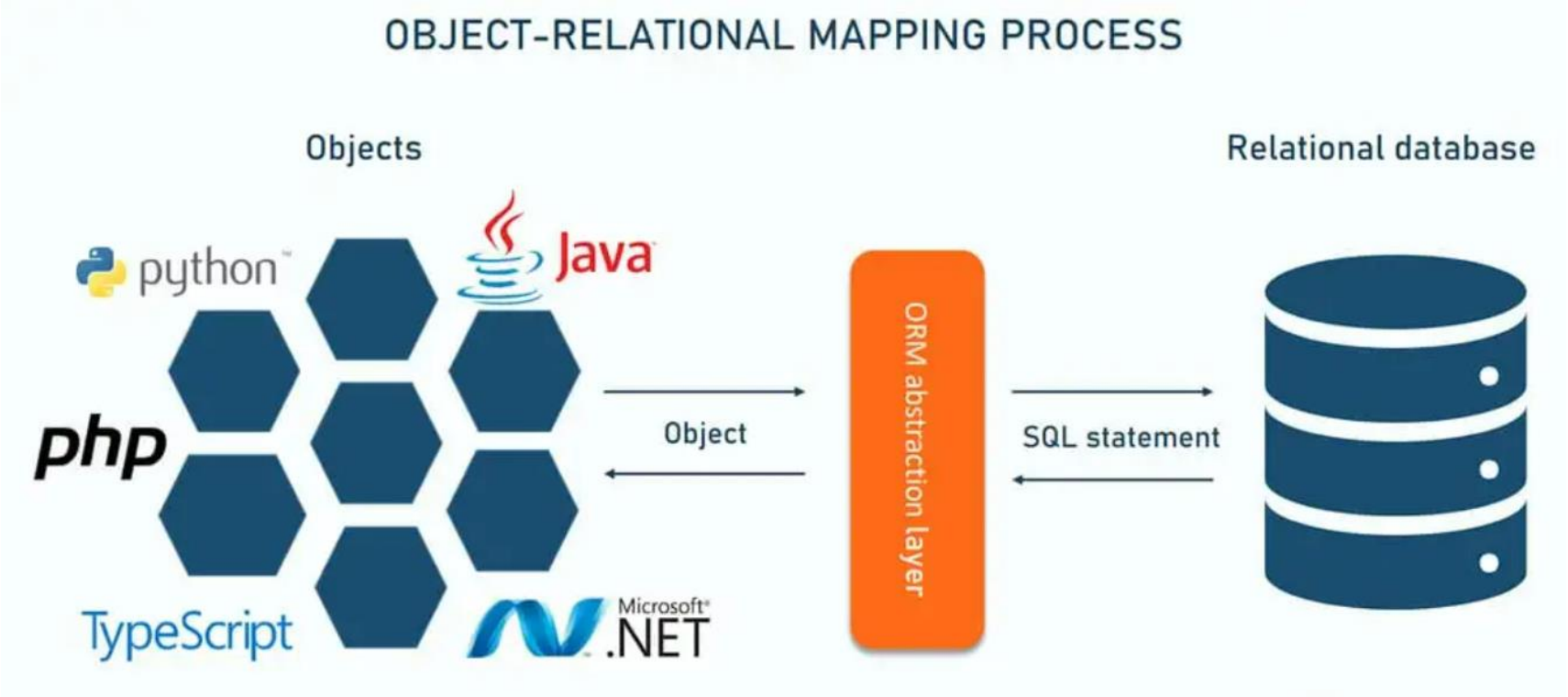


Object-To-Relational Mapping (ORM)

- The developer defines mapping between Java class and database table

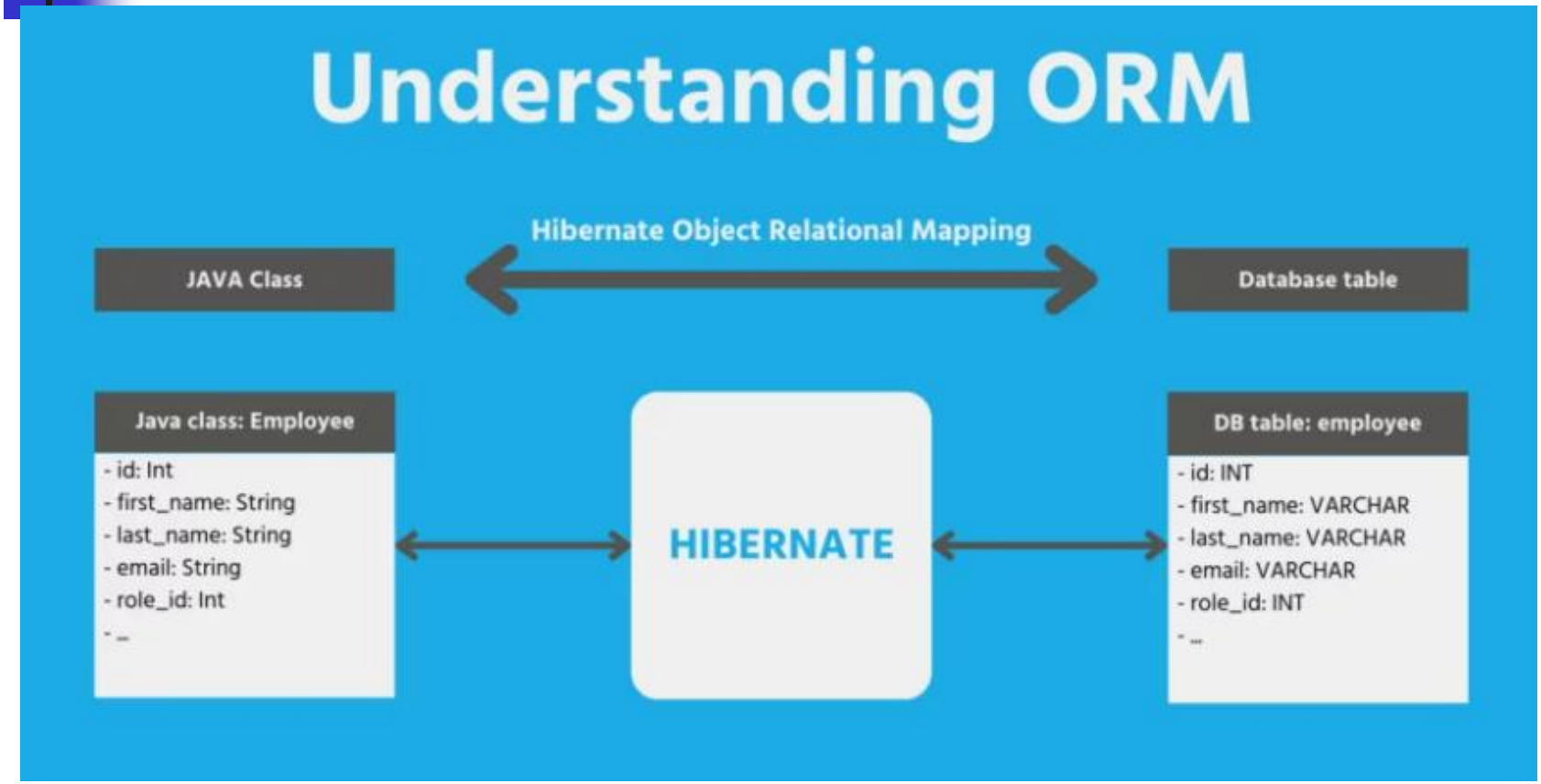


Object-to-Relational Mapping (ORM)



Object-Relational Mapping (ORM) is a programming technique used to convert data between incompatible systems—specifically between object-oriented programming languages (like Java) and relational databases (like MySQL).

Object-to-Relational Mapping (ORM)



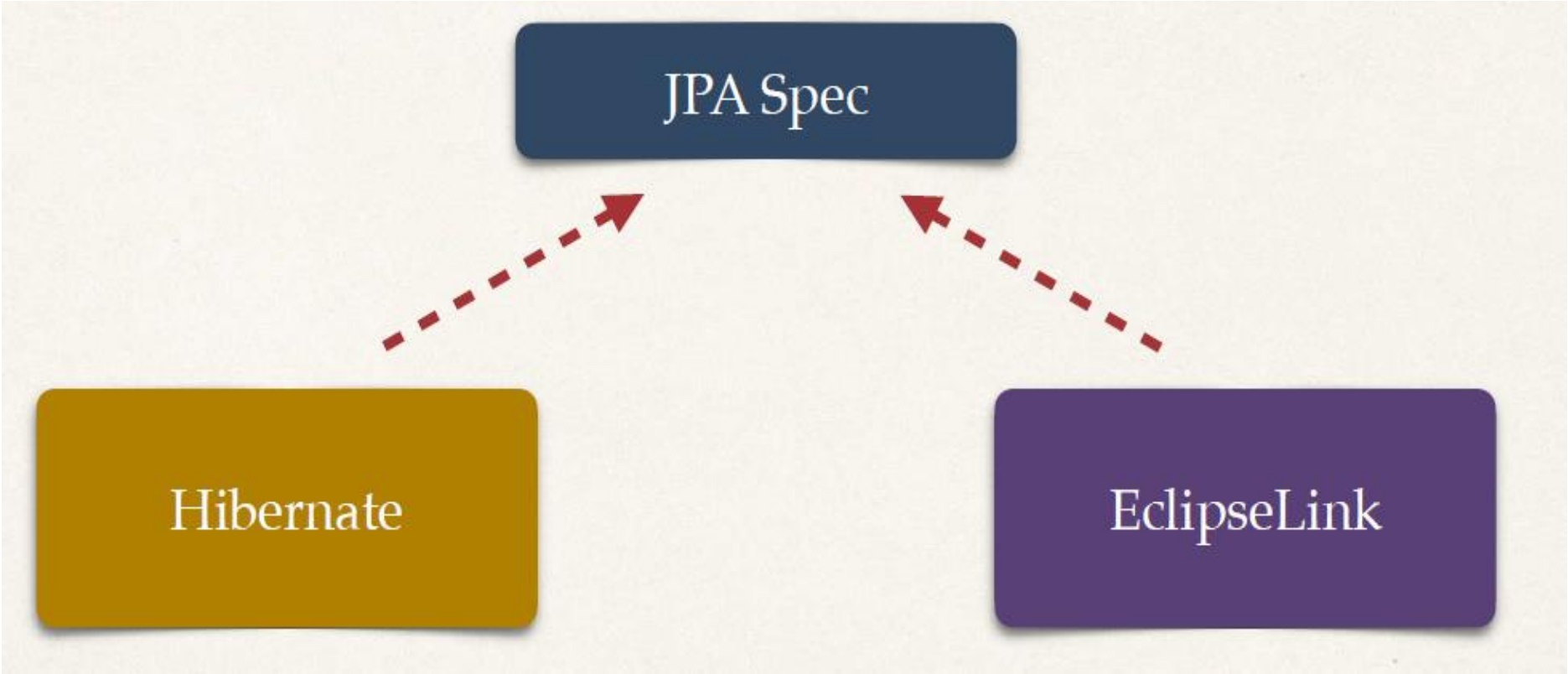
Java Hibernate is an object-relational mapping (ORM) framework that simplifies database interactions by mapping Java objects to database tables. Hibernate provides a solution to the impediment of programming with relational databases.

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

•Hibernate simplifies database interactions by using ORM.
•JPA is a standard specification; Hibernate is a popular implementation.
•Spring Boot configures Hibernate and JPA automatically.

JPA - Vendor Implementations



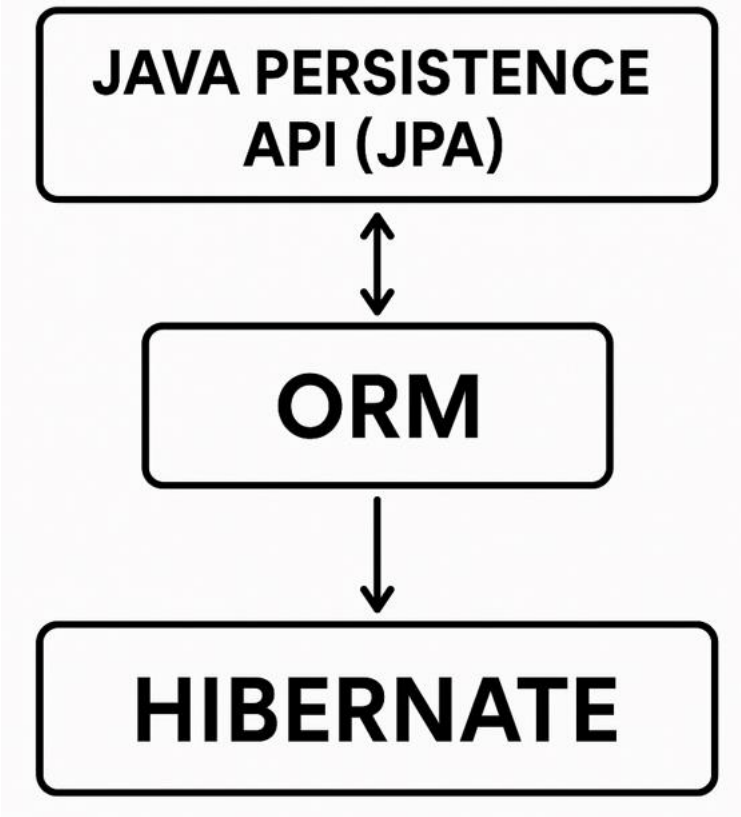
What are Benefits of JPA

- 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

JPA – ORM - Hibernate

Flow:

Application (Java code) → uses JPA interfaces → JPA relies on ORM concepts → Hibernate implements JPA + performs actual DB operations via JDBC.



Saving a Java Object with JPA

```
// 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

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

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

«from Student» is **JPQL** (Java Persistence Query Language), **not plain SQL**.
In JPQL:

- Student refers to the **entity class**, not the table name.
- "from Student" is shorthand for:

```
SELECT s FROM Student s
```

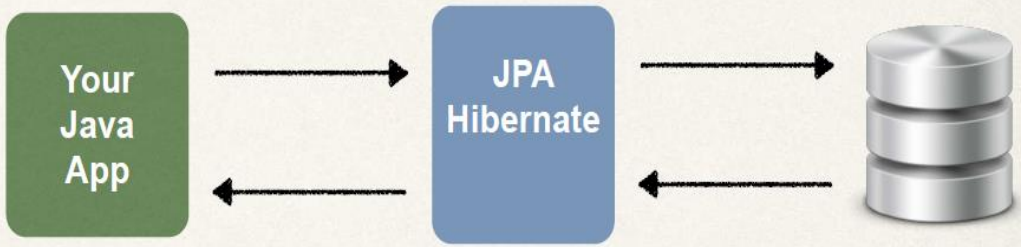
- JPQL → converted to SQL by the JPA provider (like Hibernate).

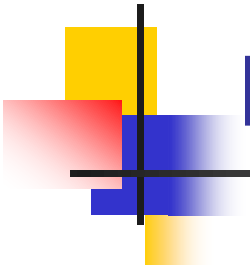
```
SELECT * FROM students;
```

Why no SELECT * ?
In **JPQL**, we deal with **Java objects (entities)**, not rows and columns. So:
- SELECT * (SQL) → not used.
- Instead, JPQL selects full objects, and your result is a **List of Student objects** (not rows of raw data).

JPA/Hibernate CRUD Apps

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



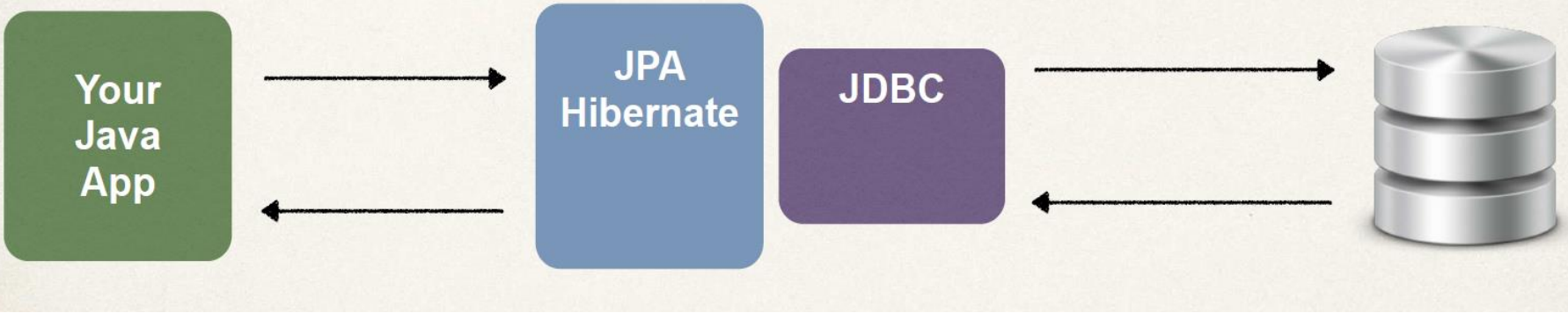


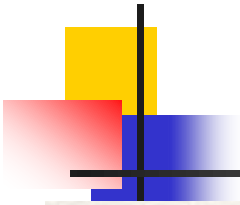
Hibernate / JPA and JDBC

How does Hibernate / JPA relate to JDBC?

Hibernate / JPA and JDBC

- Hibernate / JPA uses JDBC for all database communications

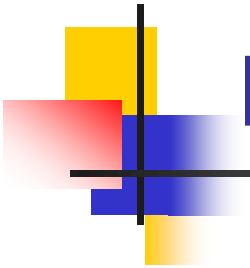




MySQL Database

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

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



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

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/`

Two Database Scripts

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

About: 01-create-user.sql

1. Create a new MySQL user for our application
   - user id: **springstudent**
   - password: **springstudent**

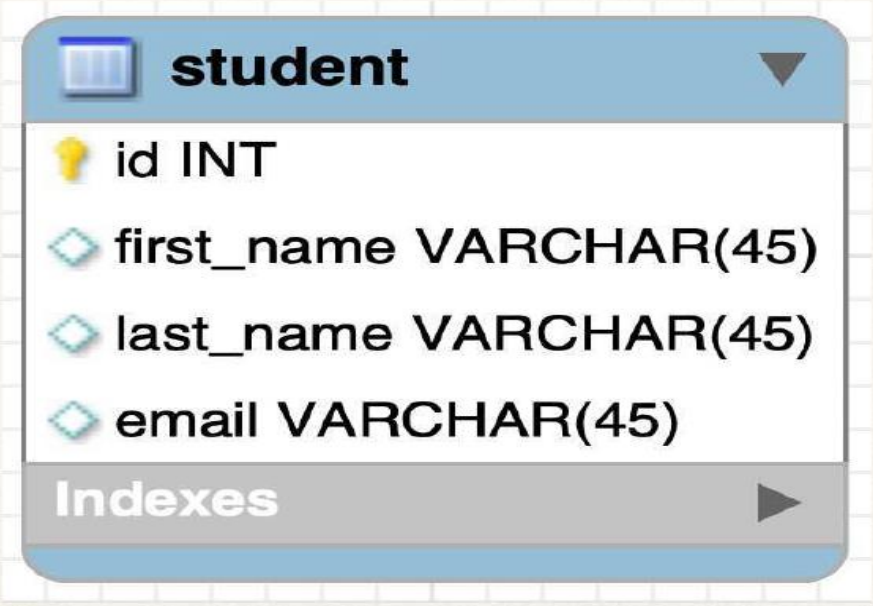
01-create-user.sql

```
-- Drop user first if they exist
DROP USER if exists 'springstudent'@'%' ;

-- Now create user with prop privileges
CREATE USER 'springstudent'@'%' IDENTIFIED BY 'springstudent';

GRANT ALL PRIVILEGES ON * . * TO 'springstudent'@'%';
```

Two Database Scripts

About: 02-student-tracker.sql

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



02-student-tracker.sql

```
CREATE DATABASE  IF NOT EXISTS `student_tracker`;
USE `student_tracker`;

--
-- Table structure for table `student`
--

DROP TABLE IF EXISTS `student`;

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`)
) ENGINE=InnoDB AUTO_INCREMENT=1 DEFAULT CHARSET=latin1;
```

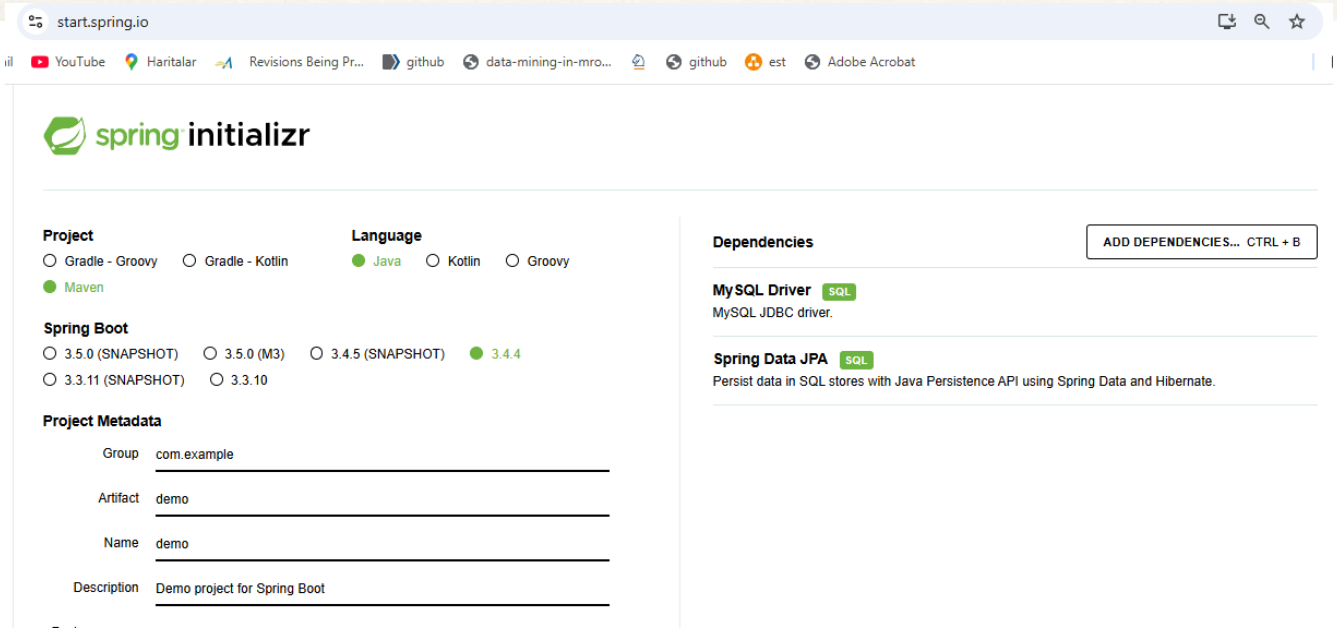
Setting Up Spring Boot Project
Automatic Data Source Configuration

- 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)

- Based on configs, Spring Boot will automatically create the beans:
  - **`DataSource, EntityManager,`** …
- You can then inject these into your app, for example your DAO

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`**



Spring Boot - Auto configuration

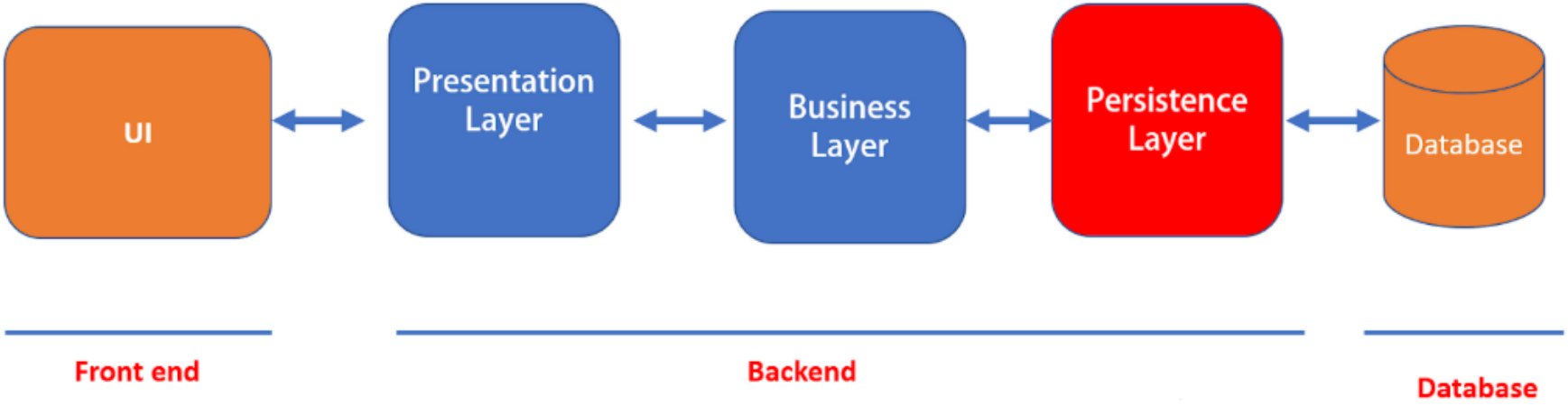
- Spring Boot will automatically configure your data source for you
- Based on entries from Maven pom file
  - JDBC Driver: **`mysql-connector-j`**
  - Spring Data (ORM): **`spring-boot-starter-data-jpa`**
- DB connection info from **`application.properties`**

application.properties

```
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

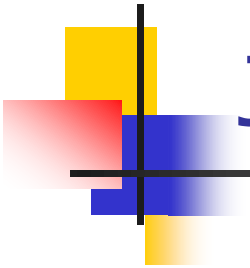
Spring MVC Three Layer Architecture



Presentation layer: This is the user interface of the application that presents the application's features and data to the user.

Business logic (or Application) layer: This layer contains the business logic that drives the application's core functionalities. Like making decisions, calculations, evaluations, and processing the data passing between the other two layers.

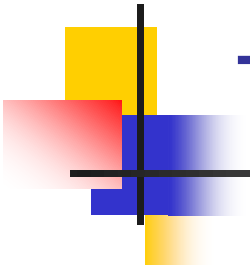
Data access layer (or Data) layer: This layer is responsible for interacting with databases to save and restore application data.



JPA Development Process

1. Annotate Java Class

2. Develop Java Code to perform database operations

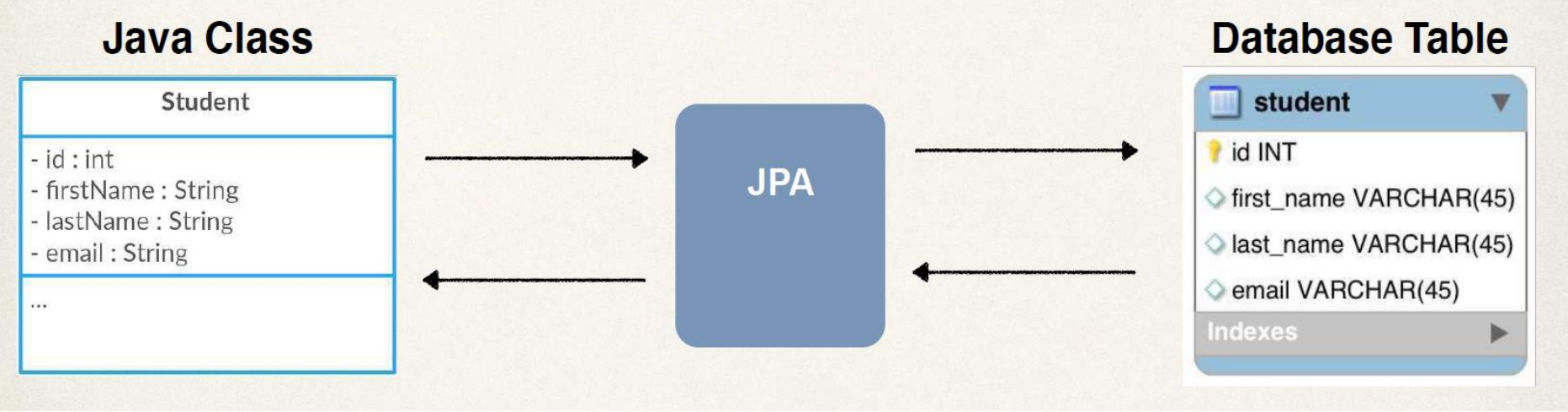


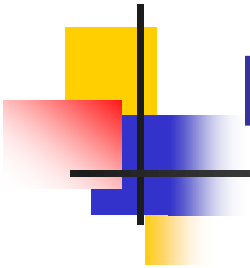
Terminology

Entity Class

Java class that is mapped to a database table

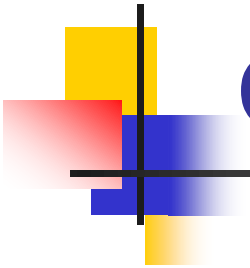
Object-to-Relational Mapping (ORM)





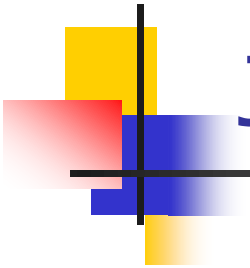
Entity Class

- 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



Constructors in Java - Refresher

- 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

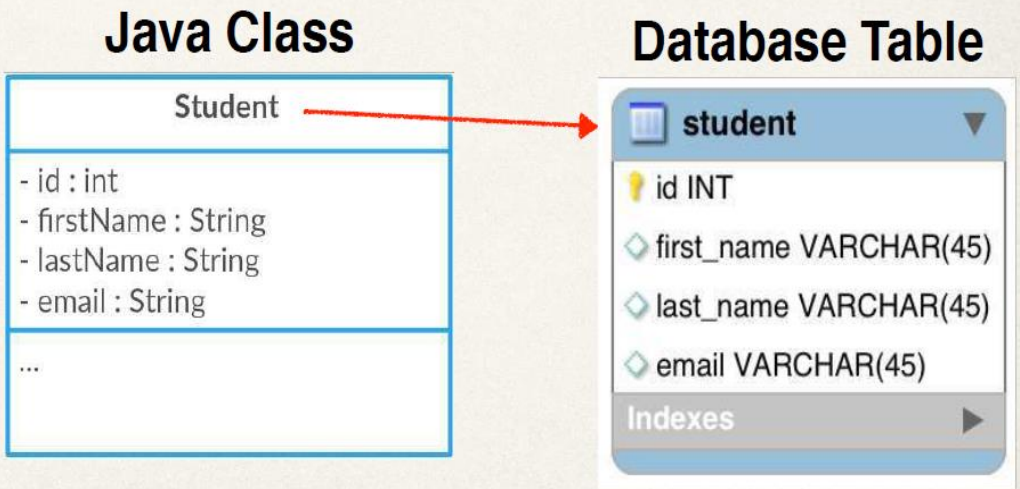


Java Annotations

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

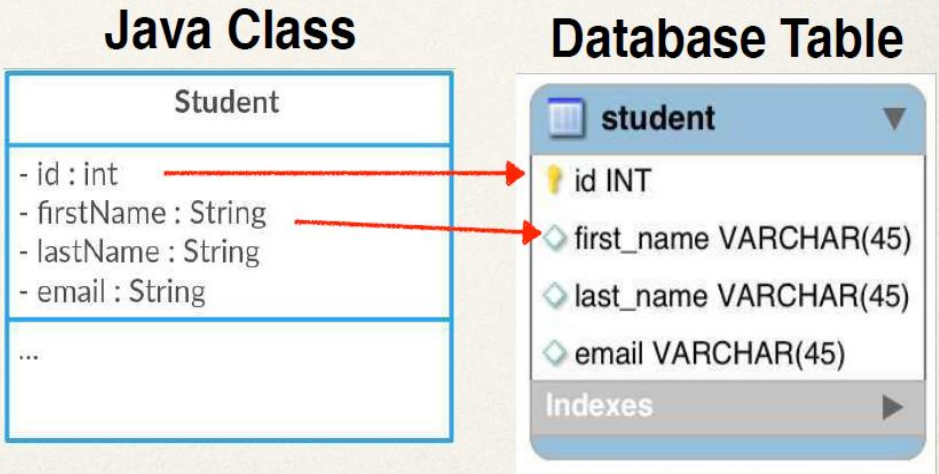
Step 1: Map class to database table

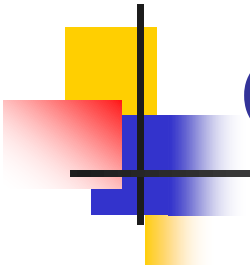
```
@Entity
@Table(name="student")
public class Student {

  ...

}
```



Step 2: Map fields to database columns

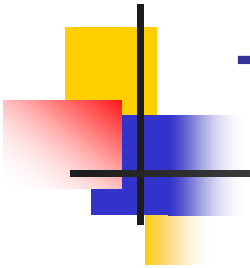
```
@Entity
@Table(name="student")
public class Student {

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

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





@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



Terminology

Primary Key

Uniquely identifies each row in a table

Must be a unique value

Cannot contain NULL values

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)

)
```

JPA Identity - Primary Key

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

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

 ...
}
```

ID Generation Strategies

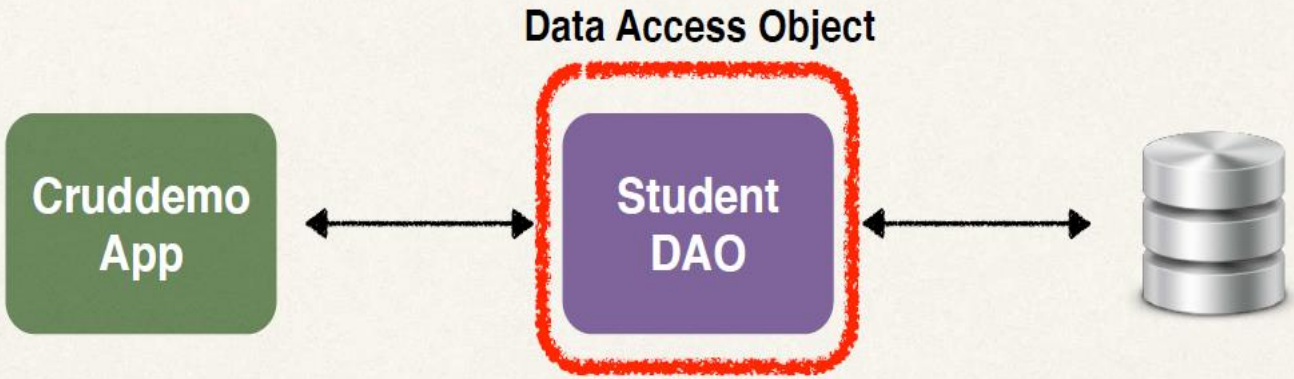
Name	Description
GenerationType.AUTO	Pick an appropriate strategy for the particular database
GenerationType.IDENTITY	Assign primary keys using database identity column
GenerationType.SEQUENCE	Assign primary keys using a database sequence
GenerationType.TABLE	Assign primary keys using an underlying database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

Save a Java Object - Sample App Features

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

Student Data Access Object

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

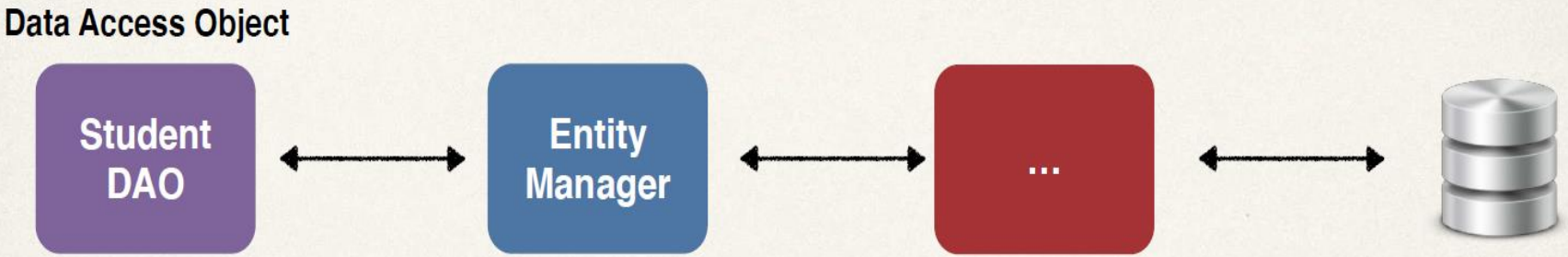


Student Data Access Object

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

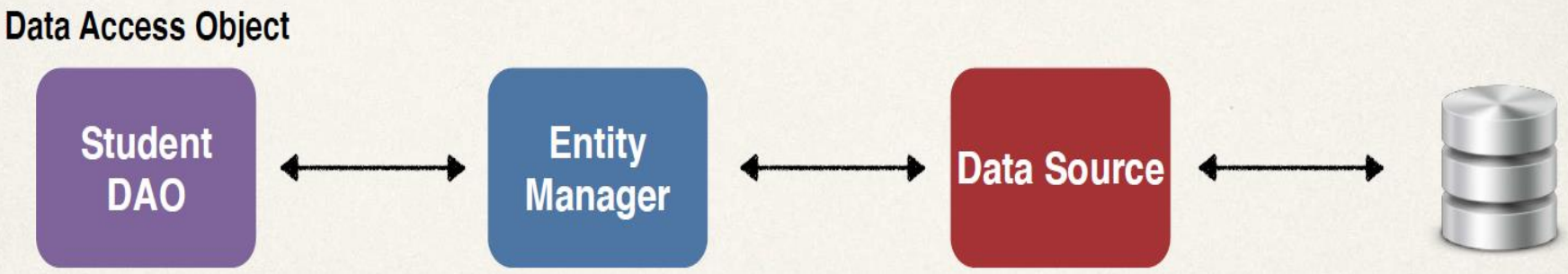
Student Data Access Object

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



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



Use Case

- If you need **low-level control and flexibility**, use **EntityManager**
- If you want **high-level of abstraction**, use **JpaRepository**

Entity Manager

- Need low-level control over the database operations and want to write custom queries
- Provides low-level access to JPA and work directly with JPA entities
- Complex queries that required advanced features such as native SQL queries or stored procedure calls
- When you have custom requirements that are not easily handled by higher-level abstractions

JpaRepository

- Provides commonly used CRUD operations out of the box, reducing the amount of code you need to write
- Additional features such as pagination, sorting
- Generate queries based on method names
- Can also create custom queries using @Query

JPA EntityManager Operations

We use EntityManager in Java (specifically in JPA – Java Persistence API) to interact directly with the persistence context and the underlying database. It's the core interface that manages the lifecycle of entities and performs operations like querying, inserting, updating, and deleting records.

What EntityManager Does:

1. Persist entities
   Saves new objects to the database.
   → entityManager.persist(entity);
2. Retrieve entities
   Finds existing entities by their primary key.
   → entityManager.find(Student.class, id);
3. Update entities
   Automatically tracks and synchronizes changes on managed entities during a transaction.
4. Delete entities
   Removes objects from the database.
   → entityManager.remove(entity);
5. Run JPQL or native SQL queries
   Executes custom queries for complex search/filter logic.
   → entityManager.createQuery("SELECT s FROM Student s");

JPA EntityManager Operations Examples

```java
@Service
@Transactional
public class StudentService {

    @PersistenceContext
    private EntityManager entityManager;

    // Get all students
    public List<Student> getAllStudents() {
        String jpql = "SELECT s FROM Student s";
        return entityManager.createQuery(jpql, Student.class).getResultList();
    }

    // Find student by ID
    public Student getStudentById(Long id) {
        Student student = entityManager.find(Student.class, id);
        if (student == null) {
            throw new EntityNotFoundException("Student not found with ID: " + id);
        }
        return student;
    }
```

```java
import com.example.demo.model.Student;
import jakarta.persistence.EntityManager;
import jakarta.persistence.EntityNotFoundException;
import jakarta.persistence.PersistenceContext;
import jakarta.persistence.TypedQuery;
import jakarta.transaction.Transactional;
import org.springframework.stereotype.Service;

import java.util.List;

@Service
@Transactional
public class StudentService {

    @PersistenceContext
    private EntityManager entityManager;

    // Get all students
    public List<Student> getAllStudents() {
        String jpql = "SELECT s FROM Student s";
        return entityManager.createQuery(jpql, Student.class).getResultList();
    }

    // Find student by ID
    public Student getStudentById(Long id) {
        Student student = entityManager.find(Student.class, id);
        if (student == null) {
            throw new EntityNotFoundException("Student not found with ID: " + id);
        }
        return student;
    }

    // Find student by email
    public Student getStudentByEmail(String email) {
        String jpql = "SELECT s FROM Student s WHERE s.email = :email";
        TypedQuery<Student> query = entityManager.createQuery(jpql, Student.class);
        query.setParameter("email", email);
        return query.getSingleResult();
    }

    // Delete student by ID
    public void deleteStudentById(Long id) {
        Student student = entityManager.find(Student.class, id);
        if (student != null) {
            entityManager.remove(student);
        }
    }
```

JpaRepository Examples

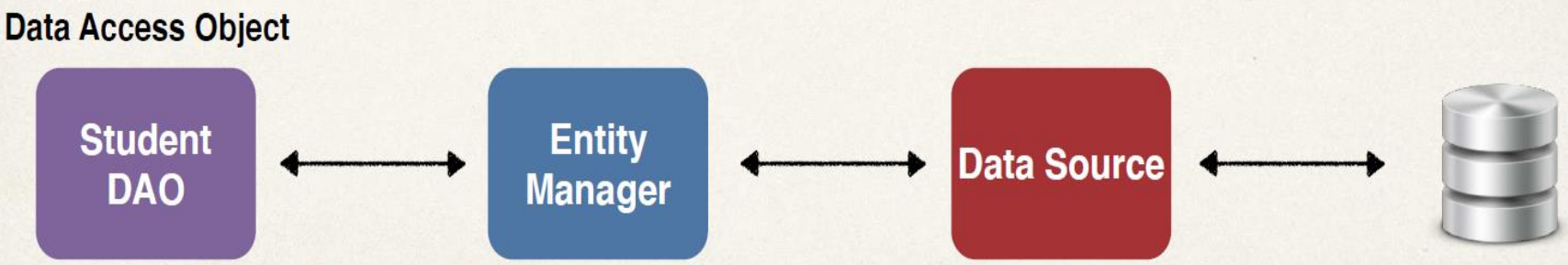
This interface uses JPA and is implemented automatically by Spring Data JPA. It relies on JPA interfaces and Hibernate

```java
package com.example.demo.repository;

import com.example.demo.model.Student;
import org.springframework.data.jpa.repository.JpaRepository;
import org.springframework.data.jpa.repository.Query;
import org.springframework.data.repository.query.Param;

import java.util.List;

public interface StudentRepository extends JpaRepository<Student, Long> {

    // 1. Find students by last name
    List<Student> findByLastName(String lastName);

    // 2. Find students by first name and last name
    List<Student> findByFirstNameAndLastName(String firstName, String lastName);

    // 3. Find students whose last name contains a substring
    List<Student> findByLastNameContaining(String substring);

    // 4. Custom query using JPQL - find by email
    @Query("SELECT s FROM Student s WHERE s.email = :email")
    Student findByEmail(@Param("email") String email);

    // 5. Custom query - find all students ordered by last name
    @Query("SELECT s FROM Student s ORDER BY s.lastName ASC")
    List<Student> findAllOrderByLastNameAsc();
}
```

Student DAO

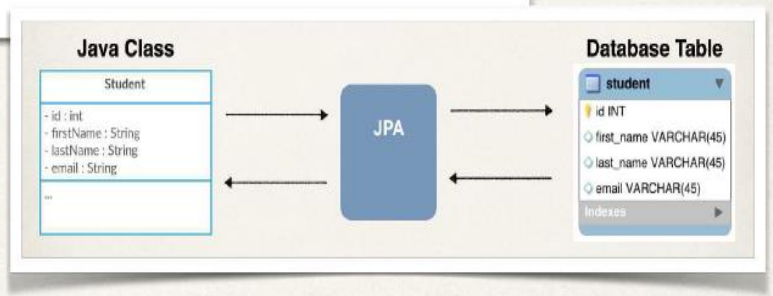
- Step 1: Define DAO interface
- Step 2: Define DAO implementation
  - Inject the entity manager
- Step 3: Update main app

Step-By-Step



Step 1: Define DAO interface

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

public interface StudentDAO {

    void save(Student theStudent);

}
```

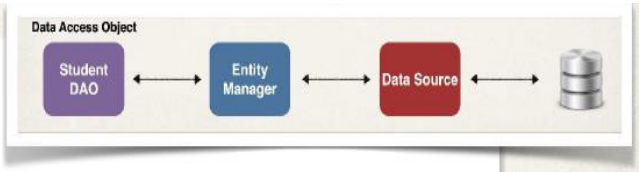


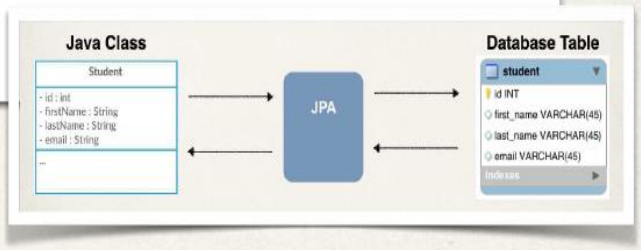
Step 2: Define DAO implementation

```java
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





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

•Spring provides @Transactional to manage transaction boundaries automatically.
•Apply to methods that perform save/update/delete.

@Transactional ensures that **all operations inside a method are executed as a single transaction**.
If one operation fails, **all others are rolled back** — like it never happened.

Scenario: Transfer Money Between Two Accounts
We'll create a method that:
1.Withdraws money from Account A
2.Deposits it into Account B
3.Rolls back if something fails (e.g., insufficient balance or system error)

Spring @Transactional

```java
@Repository
public class AccountDAOImpl implements AccountDAO {

    @PersistenceContext
    private EntityManager entityManager;

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

    @Override
    public void update(Account account) {
        entityManager.merge(account);
    }
}
```

In **JPA (Java Persistence API)**, merge() is used to **update an entity** in the database.

- Finding the matching row using the entity's ID
- Updating that row with the new field values

If **everything succeeds**, balances are updated.

If an **exception occurs**, changes to both accounts are **rolled back**.

You don't need to write SQL ROLLBACK; Spring handles it automatically via @Transactional.

```java
@Service
public class TransferService {

    private final AccountDAO accountDAO;

    @Autowired
    public TransferService(AccountDAO accountDAO) {
        this.accountDAO = accountDAO;
    }

    @Transactional
    public void transfer(Long fromId, Long toId, double amount) {
        Account fromAccount = accountDAO.findById(fromId);
        Account toAccount = accountDAO.findById(toId);

        if (fromAccount.getBalance() < amount) {
            throw new RuntimeException("Insufficient balance");
        }

        fromAccount.setBalance(fromAccount.getBalance() - amount);
        toAccount.setBalance(toAccount.getBalance() + amount);

        accountDAO.update(fromAccount);
        accountDAO.update(toAccount);

        // Simulate error after update
        // throw new RuntimeException("Something went wrong!");
    }
}
```

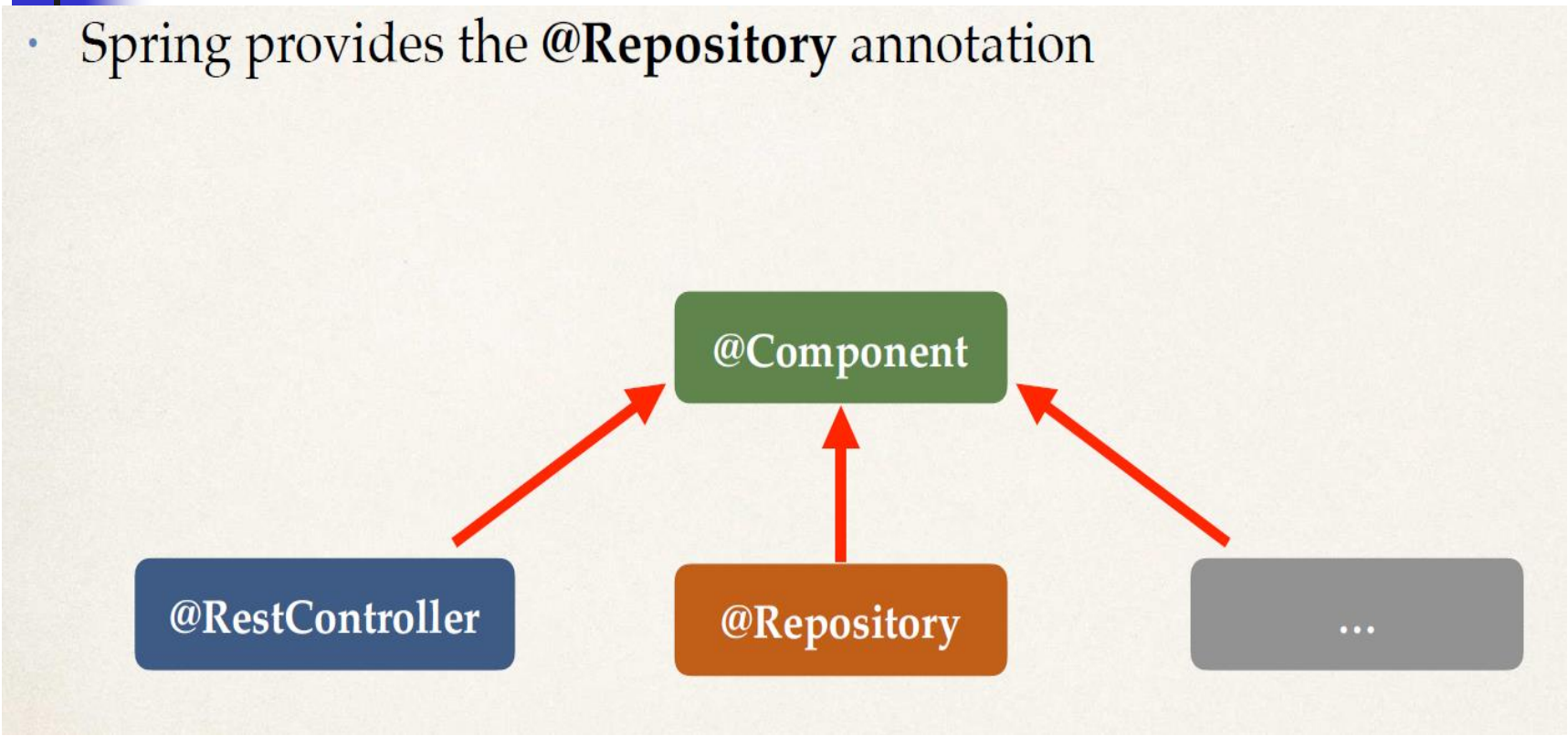
Step 2: Define DAO implementation

```
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

Specialized Annotation for DAOs

- Spring provides the **@Repository** annotation



@Component is a generic stereotype annotation for **any Spring-managed bean**. When Spring scans your classes and finds @Component, it registers the class as a **Spring Bean** in the application context.

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

Step 2: Define DAO implementation

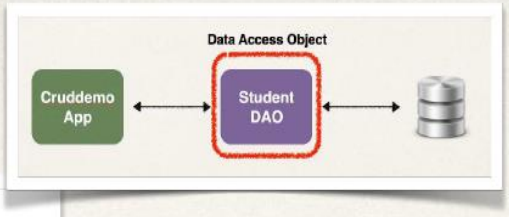
Specialized annotation for repositories

Supports component scanning

Translates JDBC exceptions

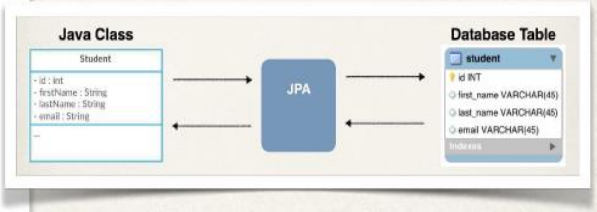
```java
import com.luv2code.cruddemo.entity.Student;
import jakarta.persistence.EntityManager;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Repository;
import org.springframework.transaction.annotation.Transactional;

@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);
    }

}
```

Step 3: Update main app

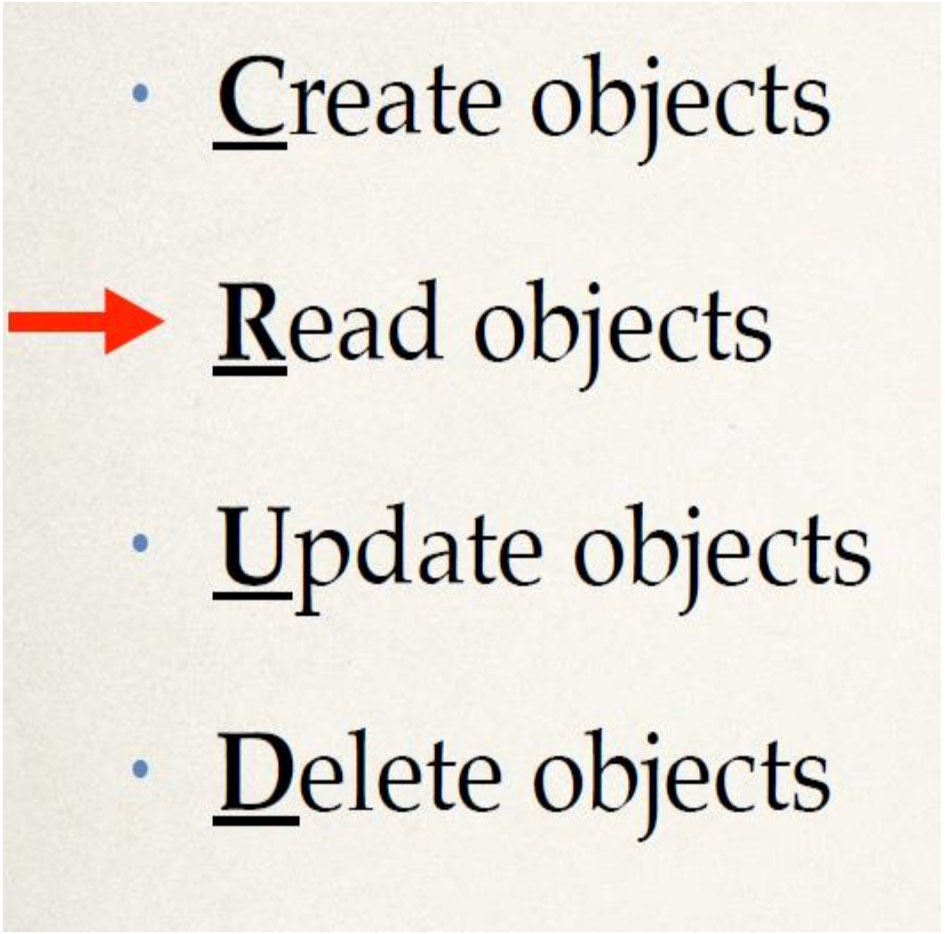



```
@SpringBootApplication
public class CruddemoApplication {

   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

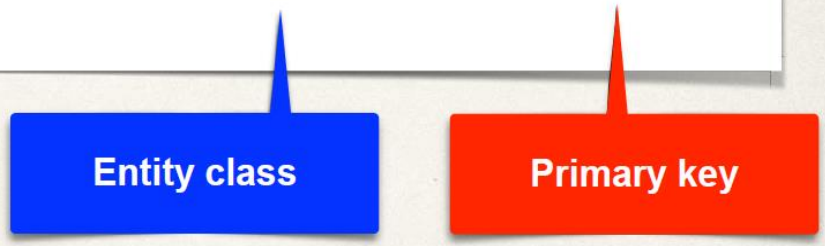


JPA CRUD Apps



Retrieving a Java Object with JPA

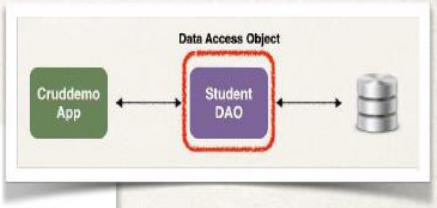
```
// retrieve/read from database using the primary key
// in this example, retrieve Student with primary key: 1

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



Development Process

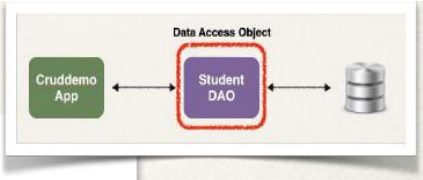
1. Add new method to DAO interface
2. Add new method to DAO implementation
3. Update main app

Step 1: Add new method to DAO interface




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

public interface StudentDAO {
    ...

    Student findById(Integer id);

}
```

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

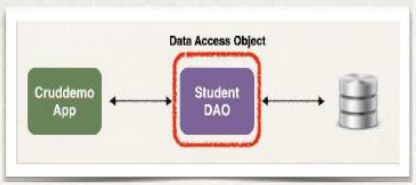
If not found, returns null

Entity class

Primary key

Step 3: Update main app

```
@SpringBootApplication
public class CruddemoApplication {
    …

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

            readStudent(studentDAO);
        };
    }

    …
```

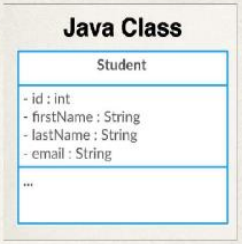



```
private void readStudent(StudentDAO studentDAO) {
    // create a student object
    System.out.println("Creating new student object...");
    Student tempStudent = new Student("Daffy", "Duck", "daffy@luv2code.com");

    // 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);
}
```

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**

Retrieving all Students



Name of JPA Entity ...
the class name

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

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

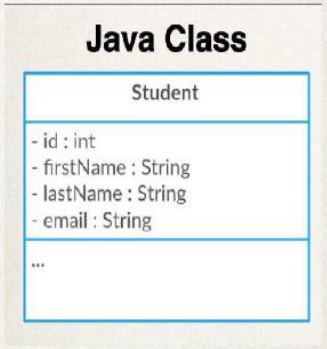
Note: this is NOT the name of the database table

All JPQL syntax is based on
entity name and entity fields

Retrieving Students: lastName = 'Doe'

Field of JPA Entity

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

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



Retrieving Students using OR predicate:

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

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

Field of JPA Entity

Field of JPA Entity

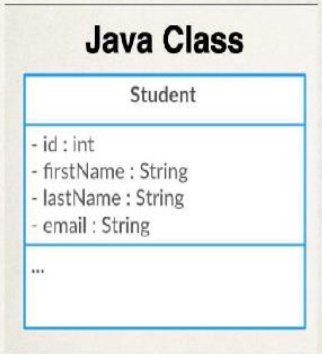
Java Class

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

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`



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

Java Class

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

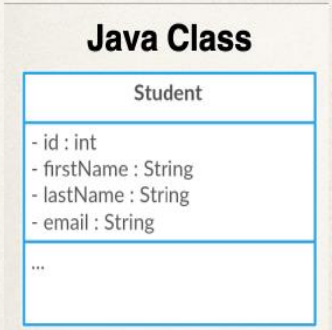
JPQL - select clause

- The query examples so far did not specify a "select" clause
- The Hibernate implementation is lenient and allows Hibernate Query Language (HQL)
- For strict JPQL, the "select" clause is required

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

s is an "identification variable" / alias
Provides a reference to the Student entity object

s - Can be any name
Useful for when you have complex queries



JPQL - select clause

- Other examples, for strict JPQL

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

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

SQL vs JPQL

SQL (Structured Query Language) is the standard language used to query and manipulate **relational databases** directly (e.g., MySQL, PostgreSQL, Oracle).

•Operates on **tables and columns**.
•Tightly coupled to the **actual database schema**.

Example SQL query:

```
SELECT * FROM students WHERE email = 'john@example.com';
```

SQL vs JPQL

JPQL (Java Persistence Query Language) is an **object-oriented query language** used with **JPA (Java Persistence API)**. It looks like SQL but operates on **Java entity objects** and their properties.

- Operates on **entity classes and fields**, not table names or columns.
- It's database-agnostic — you write it once and run it on any supported DB.
- Executed using EntityManager.createQuery(...) in **JPA** (Java Persistence API).

Example JPQL query:

```
SELECT s FROM Student s WHERE s.email = :email
```

Student is a Java class (not a table).
s.email is a Java field (not a column).
Uses Java-style syntax and types.

SQL vs JPQL

SQL	JPQL
`SELECT * FROM students WHERE email = 'john@exampple.com';`	`SELECT s FROM Student s WHERE s.email = :email`
Operation: works with tables and columns	works with **entities** and fields
Entity: uses table and column names	uses **entity** and field names

JPQL Examples

```
@PersistenceContext
private EntityManager entityManager;

    Find students by first name like a pattern (e.g., "Jo%")
public List<Student> findStudentsByFirstNameLike(String pattern) {
    String jpql = "SELECT s FROM Student s WHERE s.firstName LIKE :pattern";
    return entityManager.createQuery(jpql, Student.class)
            .setParameter("pattern", pattern)
            .getResultList();
}
```

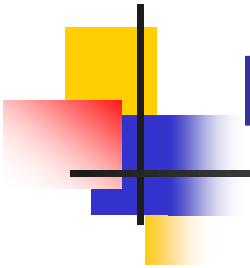
```
    Find students older than a specific age
public List<Student> findStudentsOlderThan(int age) {
    String jpql = "SELECT s FROM Student s WHERE s.age > :age";
    return entityManager.createQuery(jpql, Student.class)
            .setParameter("age", age)
            .getResultList();
}
```

JPQL Examples

```
    JPQL with JOIN - get students who have enrolled in a course with a specific title
public List<Student> findStudentsByCourseTitle(String courseTitle) {
    String jpql = "SELECT s FROM Student s JOIN s.courses c WHERE c.title = :title";
    return entityManager.createQuery(jpql, Student.class)
            .setParameter("title", courseTitle)
            .getResultList();
}
```

```
    JPQL with GROUP BY - count students grouped by age
public List<Object[]> countStudentsGroupedByAge() {
    String jpql = "SELECT s.age, COUNT(s) FROM Student s GROUP BY s.age";
    return entityManager.createQuery(jpql).getResultList();
}
```

```
    JPQL with UPDATE - increase age of all students by 1
@Transactional
public int increaseAllStudentAges() {
    String jpql = "UPDATE Student s SET s.age = s.age + 1";
    return entityManager.createQuery(jpql).executeUpdate();
}
```

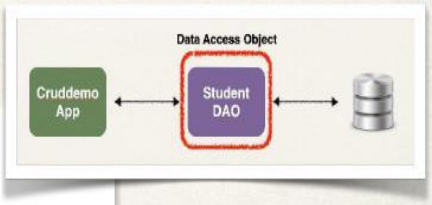


Development Process

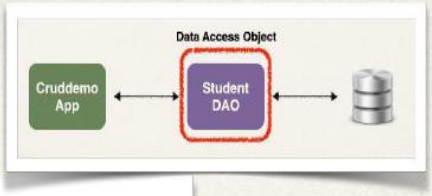
1. Add new method to DAO interface
2. Add new method to DAO implementation
3. Update main app

Step 1: Add new method to DAO interface

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

public interface StudentDAO {
    ...

    List<Student> findAll();

}
```



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();
    }

}
```

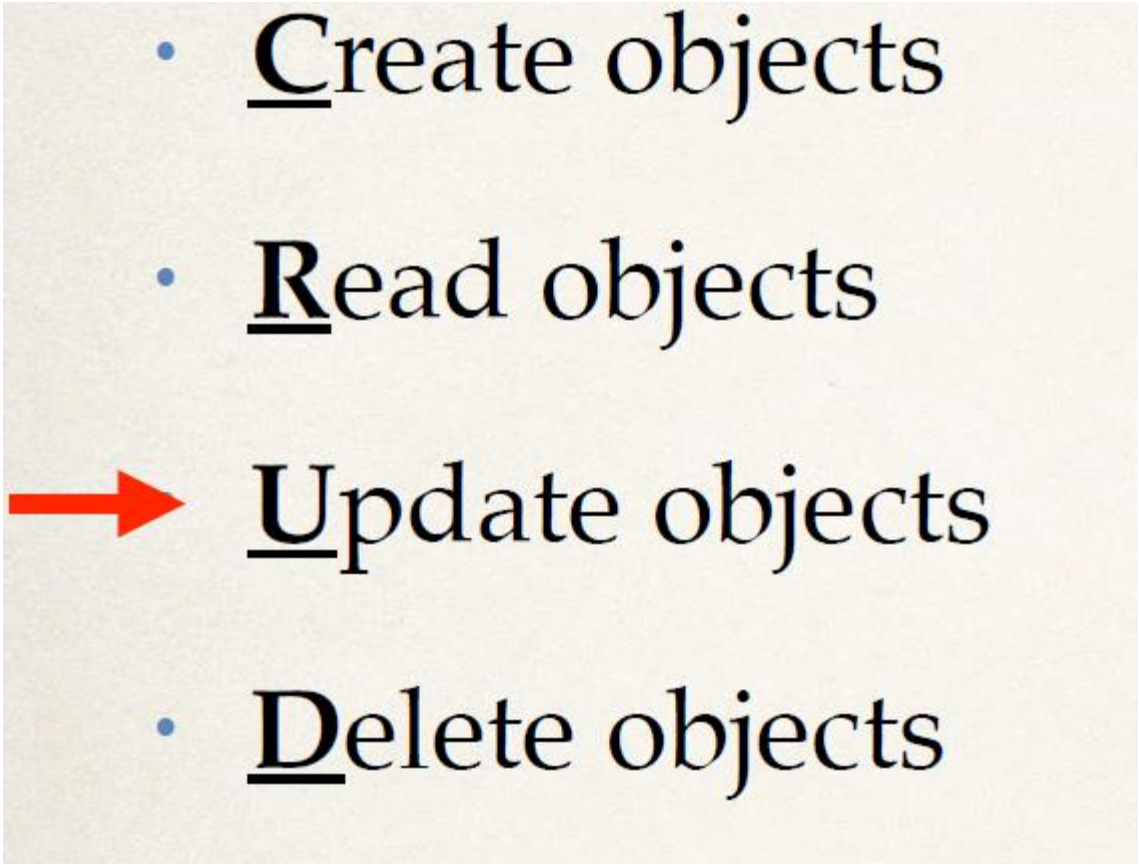
No need to add @Transactional since we are doing a query

Name of JPA Entity

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 -> {

            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);
        }
    }
}
```

JPA CRUD Apps

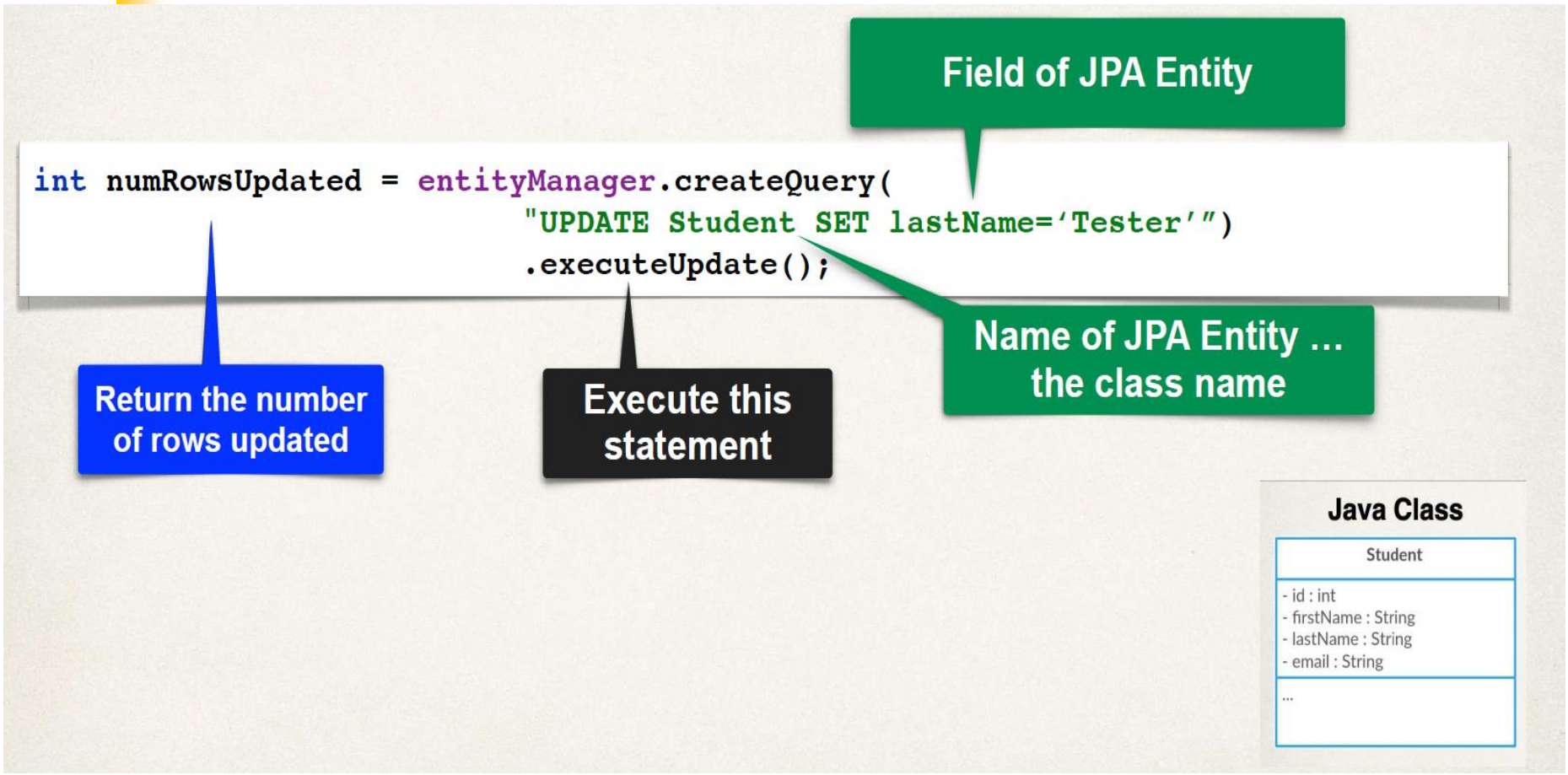


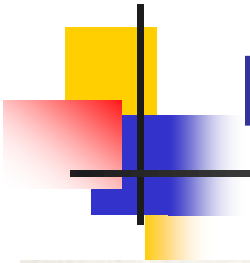
Update a Student

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

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

entityManager.merge(theStudent);
```

Update the entity

Update last name for all students



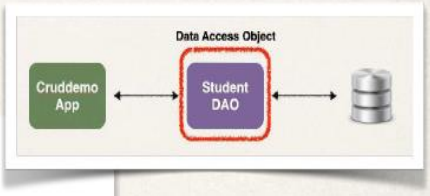


Development Process

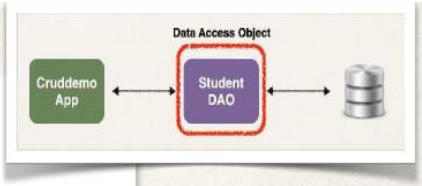
1. Add new method to DAO interface
2. Add new method to DAO implementation
3. Update main app

Step 1: Add new method to DAO interface

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

public interface StudentDAO {
    …

    void update(Student theStudent);

}
```



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);
    }

}
```

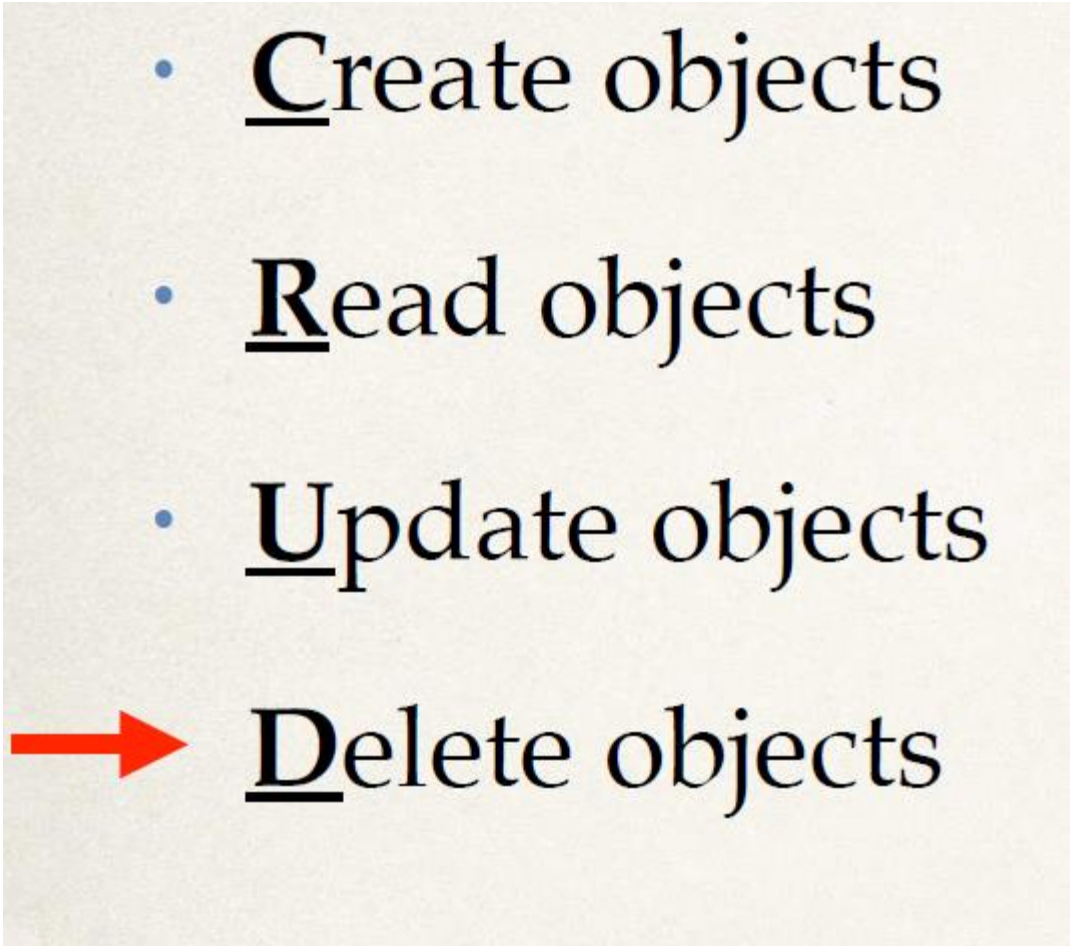
Add @Transactional since we are performing an update

Step 3: Update main app

```
@SpringBootApplication
public class CruddemoApplication {
    …

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

            updateStudent(studentDAO);
        };
    }

    …
```

```
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);
}
```

JPA CRUD Apps



Delete a Student

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

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

Delete based on a condition

```
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

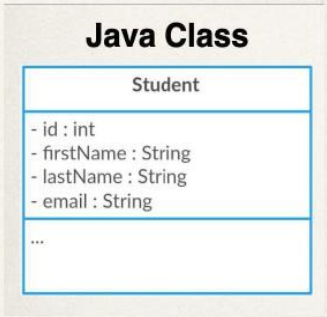
We are "modifying" the database

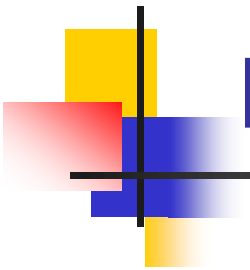
Java Class

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

Delete All Students

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



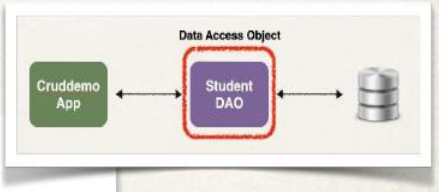


Development Process

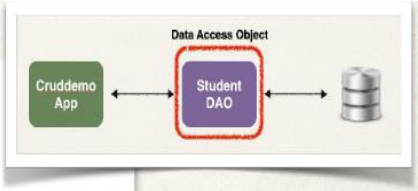
1. Add new method to DAO interface
2. Add new method to DAO implementation
3. Update main app

Step 1: Add new method to DAO interface

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

public interface StudentDAO {
    …

    void delete(Integer id);

}
```



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);
    }

}
```

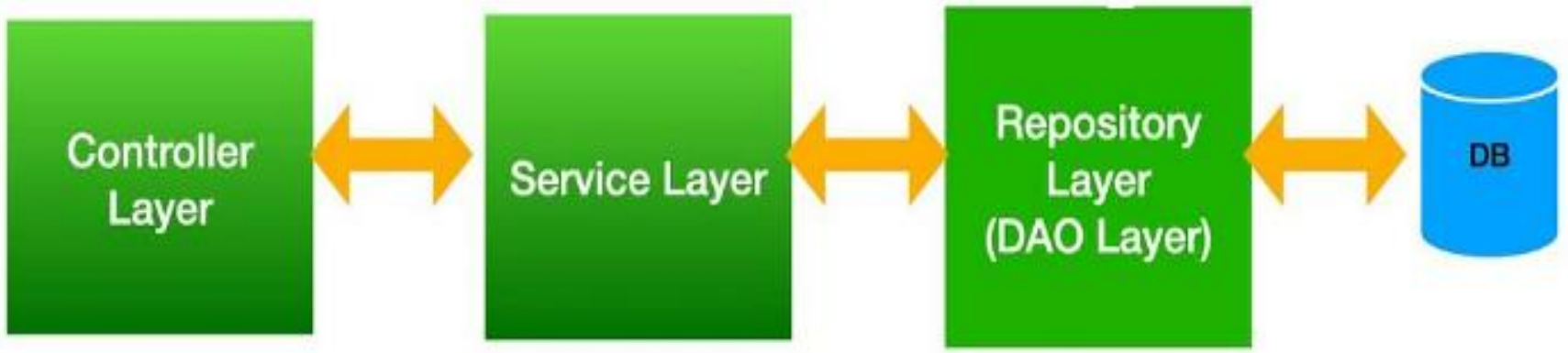
Add @Transactional since we are performing a delete

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);
}
```

Code Example



Code Example – Entity Class

```java
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;
import javax.persistence.Column;

@Entity
public class Student {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY) // Auto-increment ID
    private Long id;

    @Column(nullable = false)
    private String name;

    @Column(nullable = false)
    private int age;

    // Default constructor
    public Student() {}

    // Constructor with parameters
    public Student(String name, int age) {
        this.name = name;
        this.age = age;
    }

    // Getters and setters
    public Long getId() {
        return id;
    }

    public void setId(Long id) {
        this.id = id;
    }
```

Code Example - DAO Implementation

StudentDAOImpl class, which implements the **StudentDAO** interface and performs the actual database operations using JPA's EntityManager.

The class is annotated with **@Repository** to indicate that it is a DAO component managed by Spring.

```
public interface StudentDAO {

    Student findById(Long id);

    Student save(Student student);

    void deleteById(Long id);
}
```

```
import javax.persistence.EntityManager;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Repository;

@Repository
public class StudentDAOImpl implements StudentDAO {

    private final EntityManager entityManager;

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

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

    @Override
    public Student save(Student student) {
        if (student.getId() == null) {
            entityManager.persist(student);  // Insert new record
        } else {
            student = entityManager.merge(student);  // Update existing record
        }
        return student;
    }

    @Override
    public void deleteById(Long id) {
        Student student = findById(id);
        if (student != null) {
            entityManager.remove(student);  // Delete record
        }
    }
}
```

Code Example - Service

@Service is a **Spring stereotype annotation** used to mark a class as a **service layer component**. Treat this class as a service and automatically register it as a bean in the Spring application context.

```java
@Autowired
public StudentService(StudentDAO studentDAO) {
    this.studentDAO = studentDAO;
}
```

This is called **constructor-based dependency injection**.

```java
@Service
public class StudentService {

    private final StudentDAO studentDAO;

    @Autowired
    public StudentService(StudentDAO studentDAO) {
        this.studentDAO = studentDAO;
    }

    public Student getStudentById(Long id) {
        return studentDAO.findById(id);
    }

    public Student addStudent(Student student) {
        return studentDAO.save(student);
    }

    public Student updateStudent(Long id, Student student) {
        // Assuming the student already exists
        Student existingStudent = studentDAO.findById(id);
        existingStudent.setName(student.getName());
        existingStudent.setAge(student.getAge());
        return studentDAO.save(existingStudent);
    }

    public void deleteStudent(Long id) {
        studentDAO.deleteById(id);
    }
}
```

Code Example - RestController

```java
@RestController
@RequestMapping("/students")
public class StudentController {

    private final StudentService studentService;

    @Autowired
    public StudentController(StudentService studentService) {
        this.studentService = studentService;
    }

    @GetMapping("/{id}")
    public ResponseEntity<Student> getStudentById(@PathVariable Long id) {
        Student student = studentService.getStudentById(id);
        return ResponseEntity.ok(student);
    }

    @PostMapping
    public ResponseEntity<Student> addStudent(@RequestBody Student student) {
        Student createdStudent = studentService.addStudent(student);
        return ResponseEntity.status(HttpStatus.CREATED).body(createdStudent);
    }

    @PutMapping("/{id}")
    public ResponseEntity<Student> updateStudent(@PathVariable Long id, @RequestBody Student student) {
        Student updatedStudent = studentService.updateStudent(id, student);
        return ResponseEntity.ok(updatedStudent);
    }

    @DeleteMapping("/{id}")
    public ResponseEntity<Void> deleteStudent(@PathVariable Long id) {
        studentService.deleteStudent(id);
        return ResponseEntity.noContent().build();
    }
}
```

Code Example - RestController

@RequestMapping("/students")

•This annotation maps **HTTP requests** to a specific path or URL.

•In this case, @RequestMapping("/students") means **all requests starting with /students** will be handled by this controller.

•GET /students/1 → gets student with ID 1

•POST /students → creates a new student

```
@RequestMapping("/students")
public class StudentController {
    // ... endpoints like /students/{id}, /students (POST), etc.
}
```

Code Example - RestController

```java
@GetMapping("/{id}")
public ResponseEntity<Student> getStudent(@PathVariable Long id) {
    Student student = studentService.getStudentById(id);
    return ResponseEntity.ok(student);  // HTTP 200 + student JSON
}
```

org.springframework.http.ResponseEntity is a class provided by Spring in the package:

`ResponseEntity.ok(student)`:

- Means **return HTTP 200 OK** status with the `student` object as the **response body**.

Code Example - RestController

```java
@GetMapping("/{id}")
public ResponseEntity<Student> getStudentById(@PathVariable Long id) {
    // id will be extracted from the URL, like /students/5 → id = 5
}
```

@PathVariable Long id tells Spring to **bind a value from the URL** to the method parameter.

Code Example - RestController

```java
@PostMapping
public ResponseEntity<Student> addStudent(@RequestBody Student student) {
    Student createdStudent = studentService.addStudent(student);
    return ResponseEntity.status(HttpStatus.CREATED).body(createdStudent);
}
```

Return HTTP status **201 Created** (instead of the default 200 OK).

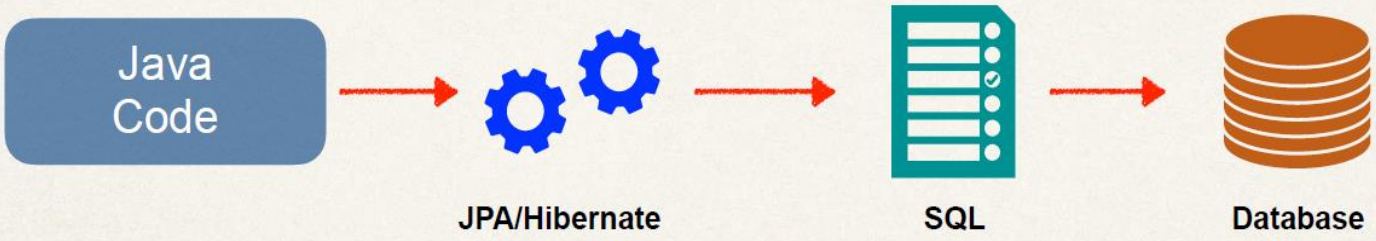
Send createdStudent object as the body of the response.

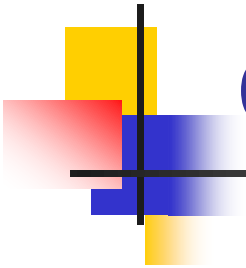
```
HTTP/1.1 201 Created
Content-Type: application/json

{
  "id": 10,
  "name": "John",
  "age": 22
}
```

Create Database Tables from Java Code

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





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

Creating Tables based on Java Code

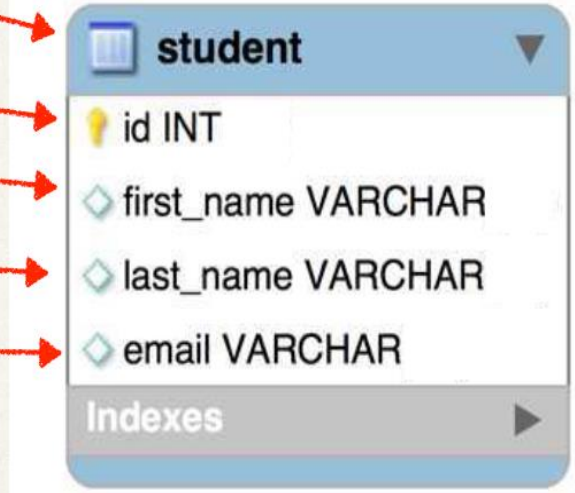
```java
@Entity
@Table(name="student")
public class Student {

  @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

Hibernate will generate and execute this

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



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

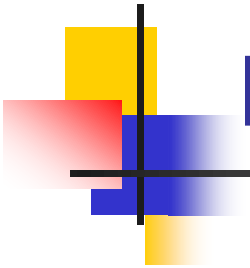
Basic Projects

- 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



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

spring.jpa.hibernate.ddl-auto=update

application.properties

```
spring.jpa.hibernate.ddl-auto=update
spring.datasource.url=jdbc:mysql://localhost:3306/testdb
spring.datasource.username=root
spring.datasource.password=yourpassword
spring.datasource.driver-class-name=com.mysql.cj.jdbc.Driver
```

Step 1: Initial Entity

```
@Entity
public class Student {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    private String name;
}
```

Now when you run your Spring Boot app, Hibernate will create a table like this:

```
CREATE TABLE student (
    id BIGINT AUTO_INCREMENT PRIMARY KEY,
    name VARCHAR(255)
);
```

spring.jpa.hibernate.ddl-auto=update

Step 2 : Now let's say you **add a new field** to your entity

```
@Entity
public class Student {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    private String name;

    private String email; // new field added
}
```

Then restart the application. Hibernate will execute something like:

```
ALTER TABLE student ADD COLUMN email VARCHAR(255);
```

It **adds the new column**, but doesn't delete any data or drop the table.

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

Drops existing tables (if any).
Creates tables based on your entity classes.

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