

Hibernate

❑ Hibernate Introduction

❑ Hibernate Hello World:

- Entity class with annotation
- Configuration File
- SessionFactory and Session
- Create Table, insert values
- Fetch Data from MySQL database

❑ Hibernate entity mappings

- Hibernate **one to one** mapping in three ways
- Hibernate **one to many** mapping in two ways
- Hibernate **many to many** mapping with join table

❑ Hibernate Lazy loading

❑ Entity LifeCycle (TYI)

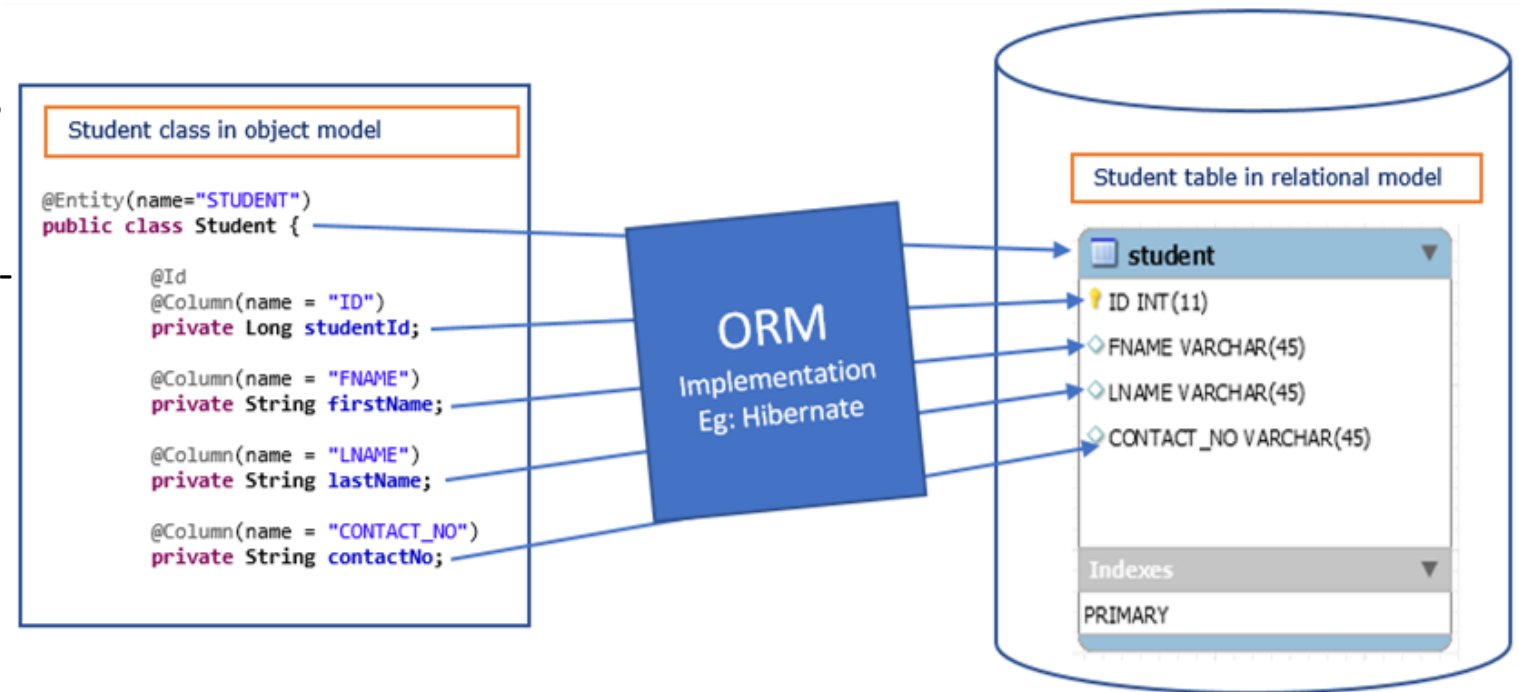
Hibernate Introduction

What is Hibernate and ORM?

- ❑ Hibernate is an **object-relational mapping (ORM)** library for the Java language, providing a framework for mapping an object-oriented domain model to a traditional relational database. This means you are not required to build and execute SQL queries for interaction with database. You just have to instruct hibernate for that by calling hibernate APIs and hibernate will create and execute those SQL queries on behalf of you.

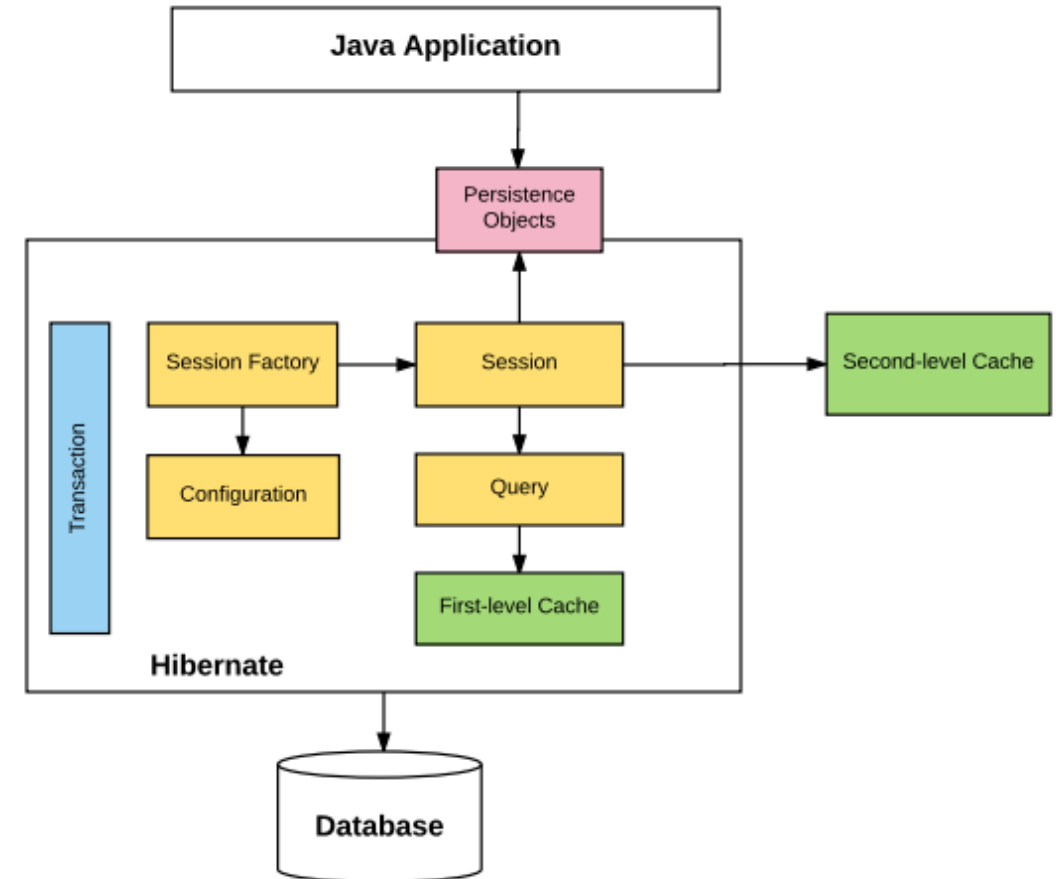
ORM:

- Mapping application objects to relational database
- solution for infamous object-relational impedance mismatch
- Finally application can focus on objects



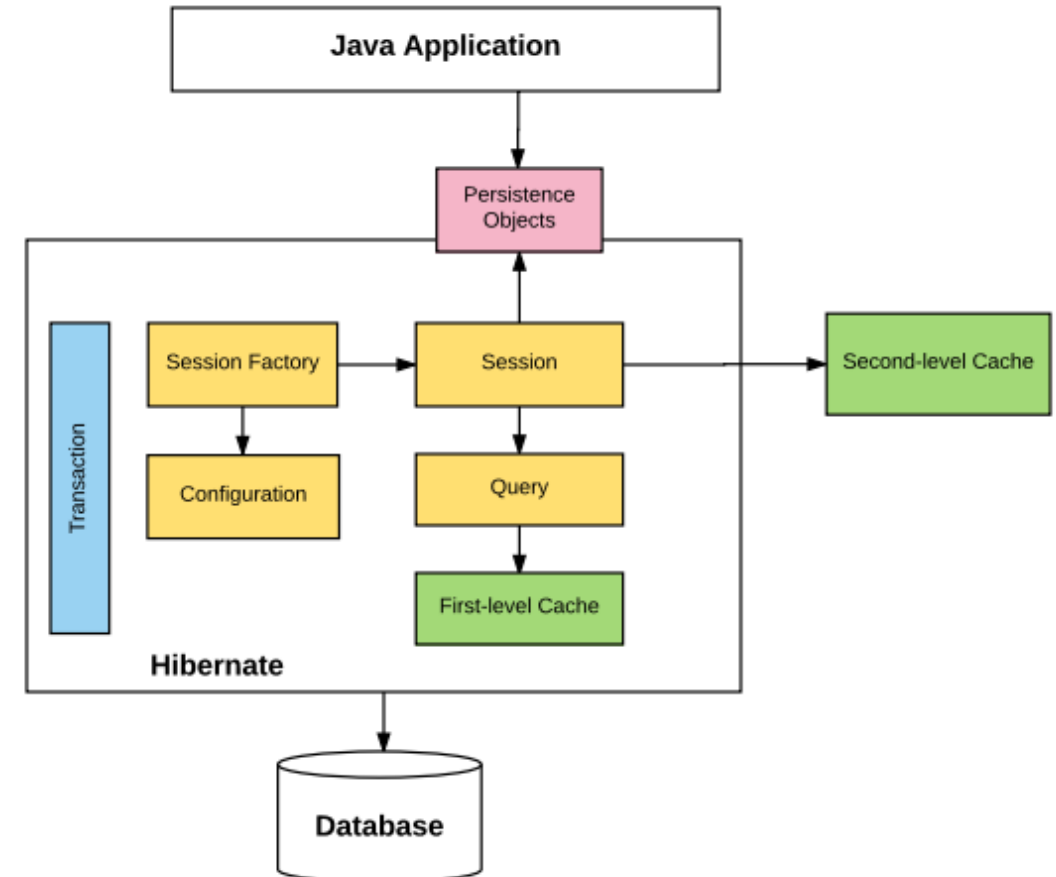
Hibernate Architecture

1. **Configuration** : Generally written in hibernate.properties or hibernate.cfg.xml files. For Java configuration, you may find class annotated with @Configuration. It is used by Session Factory to work with Java Application and the Database. It represents an entire set of mappings of an application Java Types to an SQL database.
2. **Session Factory** : Any user application requests Session Factory for a session object. Session Factory uses configuration information from above listed files, to instantiates the session object appropriately.
3. **Session** : This represents the interaction between the application and the database at any point of time. This is represented by the org.hibernate.Session class. The instance of a session can be retrieved from the SessionFactory bean.
4. **Query** : It allows applications to query the database for one or more stored objects. Hibernate provides different techniques to query database, including NamedQuery and Criteria API.



Hibernate Architecture -- continued

- 5. **First-level cache** : It represents the default cache used by Hibernate Session object while interacting with the database. It is also called as session cache and caches objects within the current session. All requests from the Session object to the database must pass through the first-level cache or session cache. One must note that the first-level cache is available with the session object until the Session object is live.
- 6. **Transaction** : enables you to achieve data consistency, and rollback incase something goes unexpected.
- 7. **Persistent objects** : These are plain old Java objects (POJOs), which get persisted as one of the rows in the related table in the database by hibernate. They can be configured in configurations files (hibernate.cfg.xml or hibernate.properties) or annotated with @Entity annotation.
- 8. **Second-level cache** : It is used to store objects across sessions. This needs to be explicitly enabled and one would be required to provide the cache provider for a second-level cache. One of the common second-level cache providers is *EhCache*.



Features of Hibernate

❑ **Object/Relational Mapping (ORM)**

Hibernate, as an ORM framework, allows the mapping of the Java domain object with database tables and vice versa. As a result, business logic is able to access and manipulate database entities via Java objects. It helps to speed up the overall development process by taking care of aspects such as transaction management, automatic primary key generation, managing database connections and related implementations, and so on.

❑ **JPA provider**

Hibernate does support the Java Persistence API (JPA) specification. JPA is a set of specifications for accessing, persisting, and managing data between Java objects and relational database entities.

❑ **Idiomatic persistence**

Any class that follows object-oriented principles such as inheritance, polymorphism, and so on, can be used as a persistent class.

❑ **High performance and scalability**

Hibernate supports techniques such as different fetching strategies, lazy initialization, optimistic locking, and so on, to achieve high performance, and it scales well in any environment.

❑ **Easy to maintain**

Hibernate is easier to maintain as it requires no special database tables or fields. It generates SQL at system initialization time. It is much quicker and easier to maintain compared to JDBC.

Basic Concepts and Code demo

-- Hello World Hibernate Project

Demo Project Setup

- ❑ Lets create our step by step hibernate demo project together. In this example, We will create a Student class and declared three properties: id, name, and email. Then with the help of Hibernate, we will create a Student table in Training database, and update it. Also, we will be fetching the data from database to App as well.
- ❑ Student entity/table in my MySQL Training database:
 - Entity name: **Student**
 - Columns:
 - **id**: int, primary key; auto generated by default as identity
 - **name**: varchar
 - **email**: varchar (unique constraint)
- ❑ The basic core of this demo consists with three parts:
 1. **hibernate.cfg.xml** -This configuration file will be used to store database connection information and schema level settings.
 2. **Student.java** – This class will refer Java POJOs having hibernate annotations.
 3. **Hibernate Util**: create session factory and session objects, transfer and commit to database.

Create a Maven project

- ❑ Suppose this is your first time creating a hibernate project, then just create a maven quickstart project:

The image displays two sequential screenshots of the Eclipse IDE's 'New Maven Project' wizard.

Left Screenshot: Select an Archetype

The 'New Maven Project' dialog is shown with the 'Select an Archetype' step. The 'Catalog' is set to 'All Catalogs'. A table lists available archetypes:

Group Id	Artifact Id	Version
org.apache.maven.archetypes	maven-archetype-archetype	1.0
org.apache.maven.archetypes	maven-archetype-j2ee-simple	1.0
org.apache.maven.archetypes	maven-archetype-plugin	1.2
org.apache.maven.archetypes	maven-archetype-plugin-site	1.1
org.apache.maven.archetypes	maven-archetype-portlet	1.0.1
org.apache.maven.archetypes	maven-archetype-profiles	1.0-alpha-4
org.apache.maven.archetypes	maven-archetype-quickstart	1.1
org.apache.maven.archetypes	maven-archetype-site	1.1
org.apache.maven.archetypes	maven-archetype-site-simple	1.1
org.apache.maven.archetypes	maven-archetype-webapp	1.0

The 'maven-archetype-quickstart' archetype is selected. Below the table, there is a description: 'An archetype which contains a sample Maven project.' At the bottom, there are checkboxes for 'Show the last version of Archetype only' (checked) and 'Include snapshot archetypes' (unchecked). The 'Next >' button is highlighted.

Right Screenshot: Specify Archetype parameters

The 'New Maven Project' dialog is shown with the 'Specify Archetype parameters' step. The parameters are:

- Group Id: com.dl
- Artifact Id: DemoHibernate
- Version: 0.0.1-SNAPSHOT
- Package: com.dl.DemoHibernate

Below these fields, there is a section for 'Properties available from archetype:' with a table:

Name	Value

At the bottom, there are buttons for '< Back', 'Next >', 'Cancel', and 'Finish'. The 'Finish' button is highlighted.

Add Dependencies

- ❑ To include Hibernate and MySQL, we need to add the following dependencies to pom.xml:

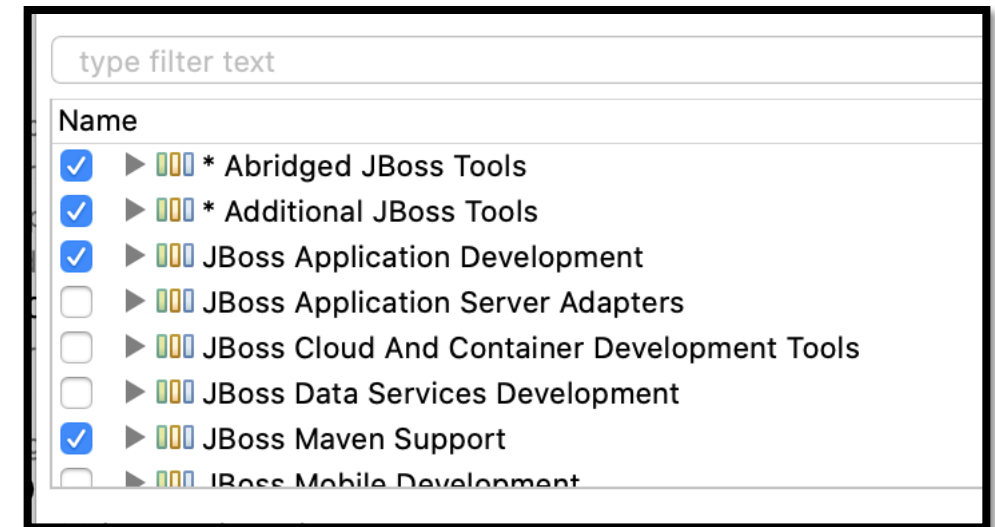
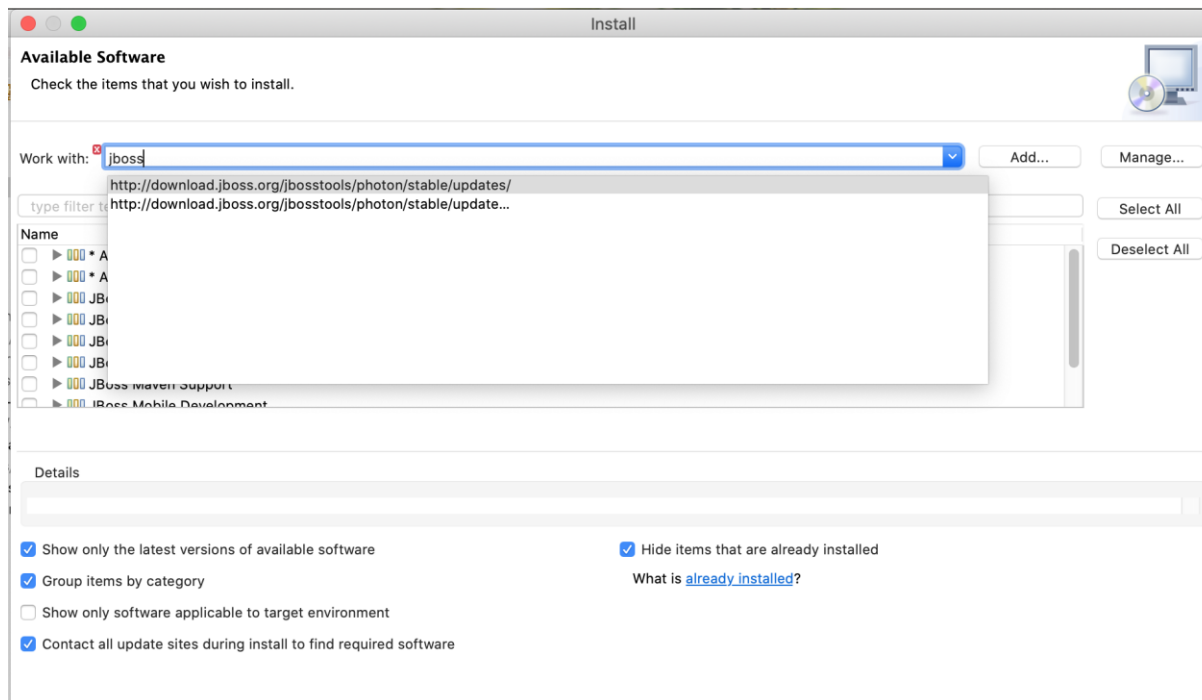
```
<!-- https://mvnrepository.com/artifact/org.hibernate.orm/hibernate-core -->  
<dependency>  
  <groupId>org.hibernate</groupId>  
  <artifactId>hibernate-core</artifactId>  
  <version>4.1.6.Final</version>  
</dependency>
```

```
<!-- https://mvnrepository.com/artifact/mysql/mysql-connector-java -->  
<dependency>  
  <groupId>mysql</groupId>  
  <artifactId>mysql-connector-java</artifactId>  
  <version>8.0.20</version>  
</dependency>
```

The version number here need to match your MySQL version to avoid connection error later.

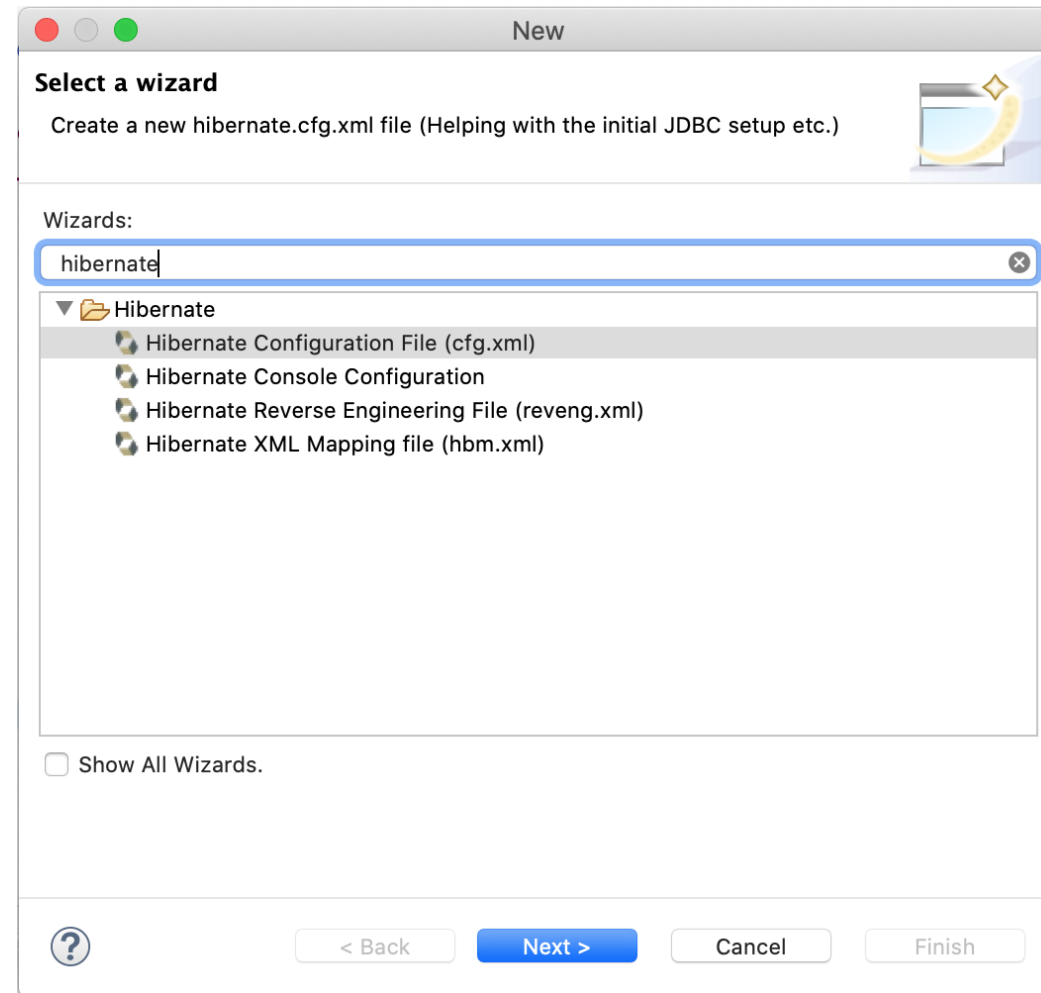
Install JBoss Hibernate Tools

- ❑ Another thing we need is the JBoss Hibernate Tools. If you are using Windows, you can go to Eclipse Marketplace and install the JBoss Tools. No need to install all of the tools, just pick hibernate during installation.
- ❑ If you are using Mac, and don't see JBoss in marketplace. Then go to help -> install new software, and type jboss to get the path. Check the needed ones for Hibernate project as shown:



1. Configuration File

- ❑ To create a hibernate configuration file. Right click on the project name and choose new file → others. Search for Hibernate, choose the highlighted one:



1. Configuration File

- Fill in the database connection information and click on Finish:

Hibernate Configuration File (cfg.xml)
This wizard creates a new configuration file to use with Hibernate.

Container: /DemoHibernate/src/main/java
File name: hibernate.cfg.xml
Hibernate version: 5.4
Session factory name:

[Get values from Connection](#)

Database dialect: MySQL
Driver class: com.mysql.jdbc.Driver
Connection URL: jdbc:mysql://localhost:3307/Training
Default Schema:
Default Catalog:
Username: root
Password:

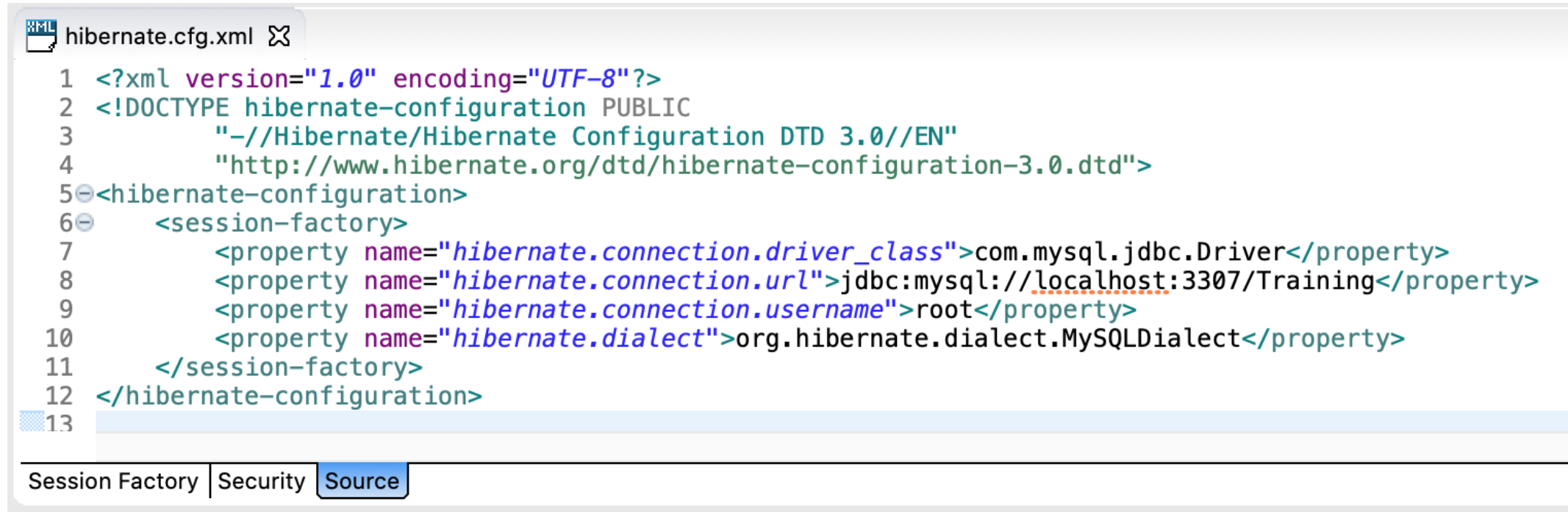
☐ Create a console configuration

? < Back Next > Cancel Finish

Training is the database name.

1. Configuration File

After you click 'Finished', the Configuration File will be auto generated as showing:



```
hibernate.cfg.xml ✕
1 <?xml version="1.0" encoding="UTF-8"?>
2 <!DOCTYPE hibernate-configuration PUBLIC
3     "-//Hibernate/Hibernate Configuration DTD 3.0//EN"
4     "http://www.hibernate.org/dtd/hibernate-configuration-3.0.dtd">
5 <hibernate-configuration>
6     <session-factory>
7         <property name="hibernate.connection.driver_class">com.mysql.jdbc.Driver</property>
8         <property name="hibernate.connection.url">jdbc:mysql://localhost:3307/Training</property>
9         <property name="hibernate.connection.username">root</property>
10        <property name="hibernate.dialect">org.hibernate.dialect.MySQLDialect</property>
11    </session-factory>
12 </hibernate-configuration>
13
```

Session Factory | Security | Source

We will make changes to this file later. But for now, let's leave it as default

2. Entity Class: Student.java

First create a normal POJO class, then add annotations. You can tell the annotations are mapping the POJO properties to MySQL table columns.

```
// Student POJO
public class Student {

    private int id;
    private String name;
    private String email;

    public int getId() {...}
    public void setId(int id) {...}
    public String getName() {...}
    public void setName(String name) {...}
    public String getEmail() {...}
    public void setEmail(String email) {...}

}
```



```
import javax.persistence.Column;
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;
import javax.persistence.Table;
import javax.persistence.UniqueConstraint;

@Entity
@Table(name="Student", uniqueConstraints = {
    @UniqueConstraint(columnNames = "ID"),
    @UniqueConstraint(columnNames = "EMAIL")})
public class Student {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    @Column(name = "ID", unique = true, nullable = false)
    private int id;

    @Column(name = "NAME", unique = false, nullable = false, length = 100)
    private String name;

    @Column(name = "EMAIL", unique = true, nullable = false, length = 100)

    private String email;
    public int getId() {...}
    public void setId(int id) {...}
    public String getName() {...}
    public void setName(String name) {...}
    public String getEmail() {...}
    public void setEmail(String email) {...}

}
```

3. Hibernate Utilities

```
public class App {  
    public static void main( String[] args ) {  
  
        // Create a student object and set values to properties  
        Student studentObj = new Student();  
  
        // Connect to the Hibernate configuration file  
        Configuration con = new Configuration().configure().addAnnotatedClass(Student.class);  
  
        // Use ServiceRegistry to build a SessionFactory  
        ServiceRegistry serviceRegistry = new ServiceRegistryBuilder().applySettings(con.getProperties()).buildServiceRegistry();  
  
        SessionFactory sf = con.buildSessionFactory(serviceRegistry);  
  
        // Open a session from the session factory  
        Session session = sf.openSession();  
  
        // Use transaction as a path to connect between Java App to database  
        Transaction tx = session.beginTransaction();  
  
        // In this example, we are going to save the object to database  
        session.save(studentObj);  
  
        // The change won't take place in database until the transaction is committed.  
        tx.commit();  
    }  
}
```

```
import org.hibernate.Session;  
import org.hibernate.SessionFactory;  
import org.hibernate.Transaction;  
import org.hibernate.cfg.Configuration;  
import org.hibernate.service.ServiceRegistry;  
import org.hibernate.service.ServiceRegistryBuilder;
```

Inserting Chris to Student table in Training database

3. Hibernate Utilities

- ❑ SessionFactory is an interface available in org.hibernate package which extends Referenceable and Serializable interface and provides factory methods to get session object. Let's see some points about SessionFactory.
 - SessionFactory is thread safe so multiple threads can access the SessionFactory at the same time.
 - SessionFactory is Immutable. Once we create SessionFactory we can not modify it(If you look SessionFactory methods we don't have any setter kind of method)
 - SessionFactory is created at the time of application startup, it reads all information from the configuration file(hibernate.cfg.xml file). We will see later in details.
 - We should have one SessionFactory for one database configuration.

- ❑ Session is an interface available in org.hibernate package which provides different API to communicate java application to hibernate. Let's see some points related to Session.
 - Session is not thread safe.
 - The main use of the Session object to perform create, get and delete operations for java classes(entity).
 - We can have multiple sessions for a SessionFactory.

Last step

- Lastly, we need to update the auto generated configuration file the high lighted:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE hibernate-configuration PUBLIC
"-//Hibernate/Hibernate Configuration DTD 3.0//EN"
"http://www.hibernate.org/dtd/hibernate-configuration-3.0.dtd">
<hibernate-configuration>
  <session-factory>
    <property name="hibernate.connection.driver_class">com.mysql.jdbc.Driver</property>
    <property name="hibernate.connection.url">
      jdbc:mysql://localhost:3306/Training?useJDBCCompliantTimezoneShift=true&useLegacyDatetimeCode=false&serverTimezone=UTC
    </property>
    <property name="hibernate.connection.username">root</property>
    <property name="hibernate.connection.password">your Password Here If you created one during installation</property>
    <property name="hibernate.dialect">org.hibernate.dialect.MySQLDialect</property>
    <property name="hbm2ddl.auto">create</property>
    <property name="show_sql">true</property>
  </session-factory>
</hibernate-configuration>
```

For some MySQL version, you might get an error when connecting to the server regarding Timezone. If that's the case, add this to the original url:

If we want to update existing table, replace 'create' with 'update'

Add this to show the SQL Query performed for you in console.

Run the Application

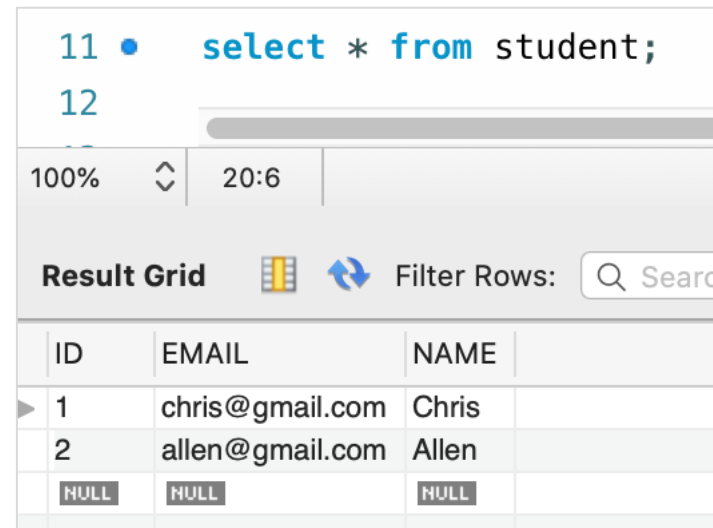
In the console, you will see these queries performed by hibernate for you:

```
Hibernate: drop table if exists Student
Hibernate: create table Student (ID integer not null auto_increment unique, EMAIL varchar(100) not null unique, NAME varchar(100) not null, primary key (ID), unique (EMAIL))
Hibernate: insert into Student (EMAIL, NAME) values (?, ?)
```

If we change ‘create’ to ‘update’ in configuration file, and create a new object to insert and run again, we will only see the insert query in console now:

```
Hibernate: insert into Student (EMAIL, NAME) values (?, ?)
```

Check in your MySQL to see if the table is created and values are inserted as expected:



The screenshot shows a MySQL query client interface. At the top, a SQL query is entered: `select * from student;`. Below the query editor, the 'Result Grid' is displayed, showing the results of the query. The grid has four columns: 'ID', 'EMAIL', 'NAME', and an empty column. There are three rows of data: the first row has ID 1, EMAIL chris@gmail.com, and NAME Chris; the second row has ID 2, EMAIL allen@gmail.com, and NAME Allen; the third row has NULL for all three fields. The interface also shows a zoom level of 100%, a duration of 20:6, and a search bar for filtering rows.

ID	EMAIL	NAME	
1	chris@gmail.com	Chris	
2	allen@gmail.com	Allen	
NULL	NULL	NULL	

Fetching Data From Database

- ❑ To change code from inserting to fetching, we still use 'update' in Configuration file, and adjust the code in main:

```
Student studentObj = new Student();

// Connect to the Hibernate configuration file
Configuration con = new Configuration().configure().addAnnotatedClass(Student.class);

// Use ServiceRegistry to build a SessionFactory
ServiceRegistry serviceRegistry = new ServiceRegistryBuilder().applySettings(con.getProperties()).buildServiceRegistry();

SessionFactory sf = con.buildSessionFactory(serviceRegistry);

// Open a session from the session factory
Session session = sf.openSession();

// Use transaction as a path to connect between Java App to database
Transaction tx = session.beginTransaction();

// This time, we are fetching the entry from table and cast it into Student object
studentObj = (Student) session.get(Student.class, 1);

// The change won't take place in database until the transaction is committed.
tx.commit();
System.out.println("student fetched from database: " + studentObj);
```

→ Instead of save(), use get(). Provide the id to fetch the corresponding object

```
student fetched from database: Student [id=1, name=Chris, email=chris@gmail.com]
```

Hibernate Mapping Relations

Hibernate Mappings

- ❑ Hibernate mappings are one of the key features of Hibernate. They establish the relationship between two database tables as attributes in your model. That allows you to easily navigate the associations in your model and Criteria queries.
- ❑ You can establish either unidirectional or bidirectional i.e you can either model them as an attribute on only one of the associated entities or on both. It will not impact your database mapping tables, but it defines in which direction you can use the relationship in your model and Criteria queries.
- ❑ The relationship that can be established between entities are-
 - One to One
 - One to Many/Many to One
 - Many to Many

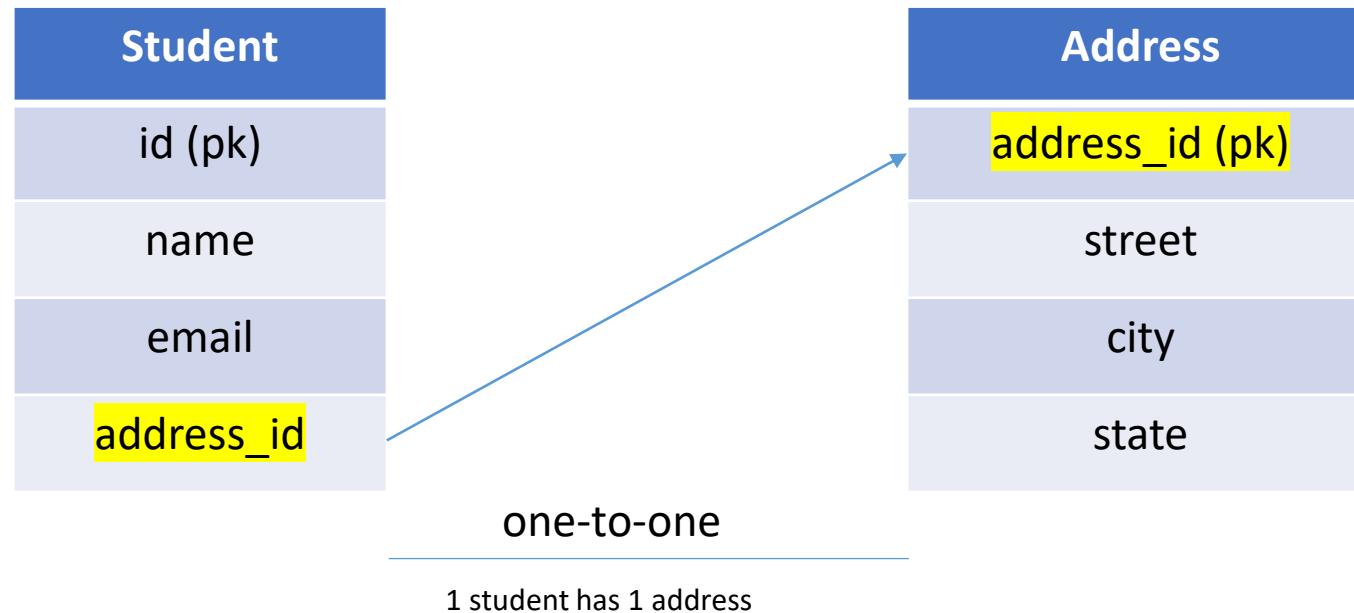
Mapping Relations

1. We will introduce 3 ways of doing **one-to-one** mappings supported in hibernate:
 - 1) Using foreign key association
 - 2) Using a common join table
 - 3) Using @MapsId for tables sharing same primary key values
2. We will introduce 2 ways of **one-to-many** mappings supported in hibernate:
 - 1) Using foreign key association
 - 2) Using a join table
3. We will also give a demo on how to do **many-to-many** mappings with join table.

1.1 One-to-One with foreign key association

1.1 One-to-One with foreign key association

- Suppose each student has one address, and every address is associated with one student. In this kind of association, a foreign key column is created in **owner entity**.



1.1 One-to-One with foreign key association

- ❑ To make such association, refer the Address entity in Student class as follow:

```
Student.java
```

```
@OneToOne  
@JoinColumn(name="address_id")  
private Address address;
```

- ❑ In a bidirectional relationship, one of the sides (and only one) has to be the owner. The owner is responsible for the association column(s) update. To declare a side as not responsible for the relationship, the attribute mappedBy is used. 'mappedBy' refers to the property name of the association on the owner side:

```
Address.java
```

```
@OneToOne(mappedBy="address")  
private Student student;
```

1.1 One-to-One with foreign key association

```
@Entity
@Table(name="Student", uniqueConstraints = {
    @UniqueConstraint(columnNames = "ID"),
    @UniqueConstraint(columnNames = "EMAIL")})
public class Student {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    @Column(name = "ID", unique = true, nullable = false)
    private int id;

    @Column(name = "NAME", unique = false, nullable = false, length = 100)
    private String name;

    @Column(name = "EMAIL", unique = true, nullable = false, length = 100)
    private String email;

    @OneToOne
    @JoinColumn(name="address_id")
    private Address address;

    ...Getters and Setters ...

    @Override
    public String toString() {...}
}
```

1.1 One-to-One with foreign key association

@Entity

public class Address {

 @Id

 @GeneratedValue(strategy = GenerationType.*IDENTITY*)

 @Column(name = "address_id", unique = **true**, nullable = **false**)

private int address_id;

 @Column(name = "STREET", unique = **false**, nullable = **false**, length = 100)

private String street;

 @Column(name = "CITY", unique = **false**, nullable = **false**, length = 20)

private String city;

 @Column(name = "STATE", unique = **false**, nullable = **false**, length = 20)

private String state;

 @OneToOne(mappedBy="address")

private Student student;

...Getters and Setters ...

 @Override

public String toString() {...}

}

1.1 One-to-One with foreign key association

```
public class App {  
    public static void main( String[] args ) {  
  
        Configuration con = new Configuration().configure().addAnnotatedClass(Student.class).addAnnotatedClass(Address.class);  
  
        ServiceRegistry serviceRegistry = new ServiceRegistryBuilder().applySettings(con.getProperties()).buildServiceRegistry();  
  
        SessionFactory sf = con.buildSessionFactory(serviceRegistry);  
        Session session = sf.openSession();  
        Transaction tx = session.beginTransaction();  
  
        // ***** create one student and the address he has *****  
        // create address object  
        Address address = new Address();  
        address.setStreet("1st Main Street");  
        address.setCity("Atlanta");  
        address.setState("GA");  
  
        // Save address  
        session.saveOrUpdate(address);  
  
        // Add new Student object  
        Student student = new Student();  
        student.setEmail("demo-user@mail.com");  
        student.setName("demo");  
  
        // Save Student  
        student.setAddress(address);  
        session.saveOrUpdate(student);  
        // *****  
  
        tx.commit();  
    }  
}
```

1.1 One-to-One with foreign key association

Console

Hibernate: create table Address (address_id integer not null auto_increment unique, CITY varchar(20) not null, STATE varchar(20) not null, STREET varchar(100) not null, primary key (address_id))

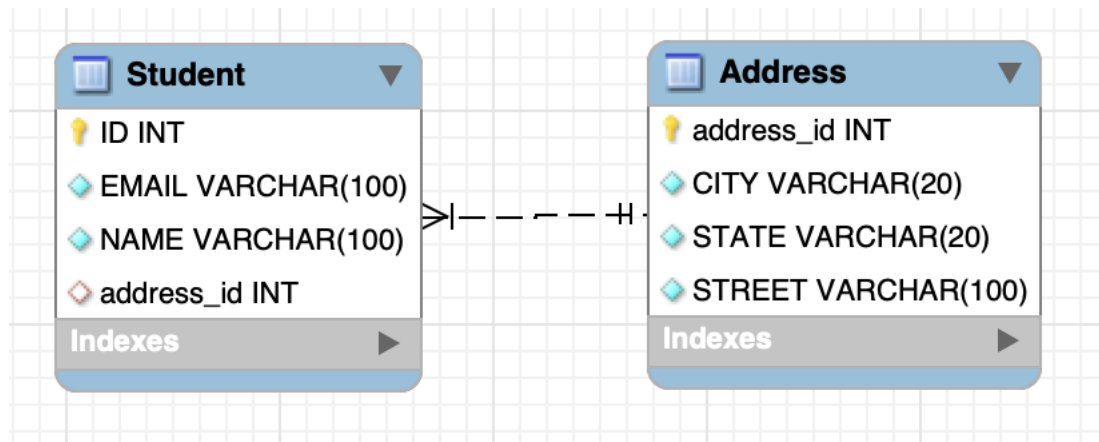
Hibernate: create table Student (ID integer not null auto_increment unique, EMAIL varchar(100) not null unique, NAME varchar(100) not null, address_id integer, primary key (ID), unique (EMAIL))

Hibernate: alter table Student add index FKf3371A1BE6F556A2 (address_id), add constraint FKf3371A1BE6F556A2 foreign key (address_id) references Address (address_id)

Hibernate: insert into Address (CITY, STATE, STREET) values (?, ?, ?)

Hibernate: insert into Student (address_id, EMAIL, NAME) values (?, ?, ?)

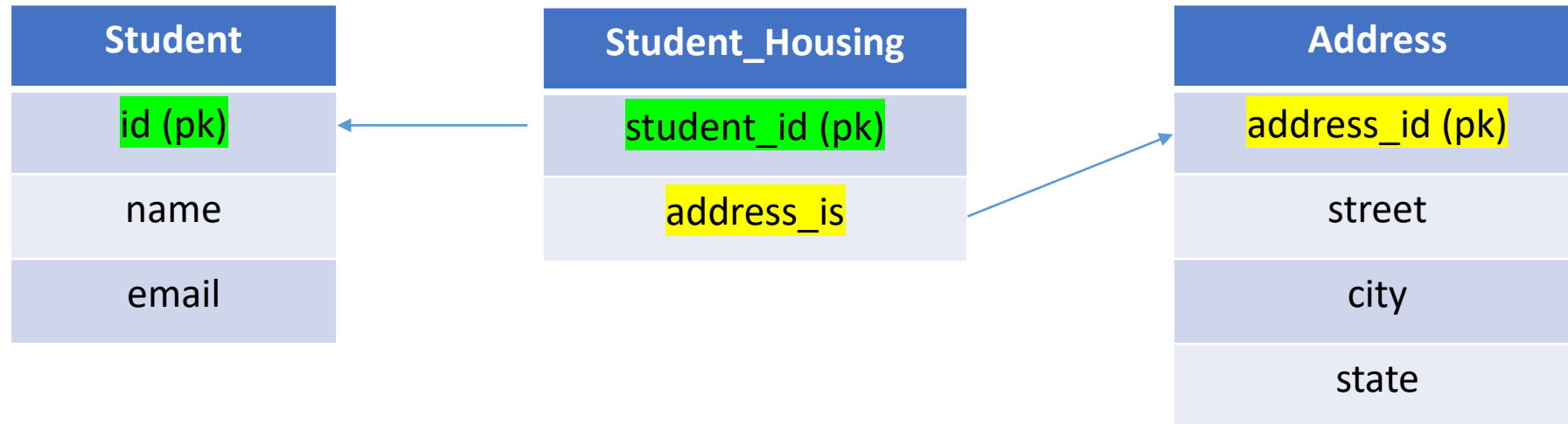
Check in MySQL Workbench:



1.2 One-to-One with a common join table

1.2 One-to-One with a common join table

- ❑ In this technique, main annotation to be used is `@JoinTable`. This annotation is used to define the new table name (mandatory) and foreign keys from both of the tables.



1.2 One-to-One with a common join table

- ❑ **@JoinTable** annotation is used in Student class. It declares that a new table Student_Housing will be created with two columns student_id(primary key of student table) and address_id (primary key of address table).

Student.java

```
@OneToOne(cascade = CascadeType.ALL)
@JoinTable(name="Student_Housing", joinColumns = @JoinColumn(name="student_id"),
inverseJoinColumns = @JoinColumn(name="address_id"))
private Address address;
```

1.2 One-to-One with a common join table

Console

Hibernate: create table Address (address_id integer not null auto_increment unique, CITY varchar(20) not null, STATE varchar(20) not null, STREET varchar(100) not null, primary key (address_id))

Hibernate: create table Student (ID integer not null auto_increment unique, EMAIL varchar(100) not null unique, NAME varchar(100) not null, primary key (ID), unique (EMAIL))

Hibernate: create table Student_Housing (address_id integer, student_id integer not null, primary key (student_id))

Hibernate: alter table Student_Housing add index FK9415A999E6F556A2 (address_id), add constraint FK9415A999E6F556A2 foreign key (address_id) references Address (address_id)

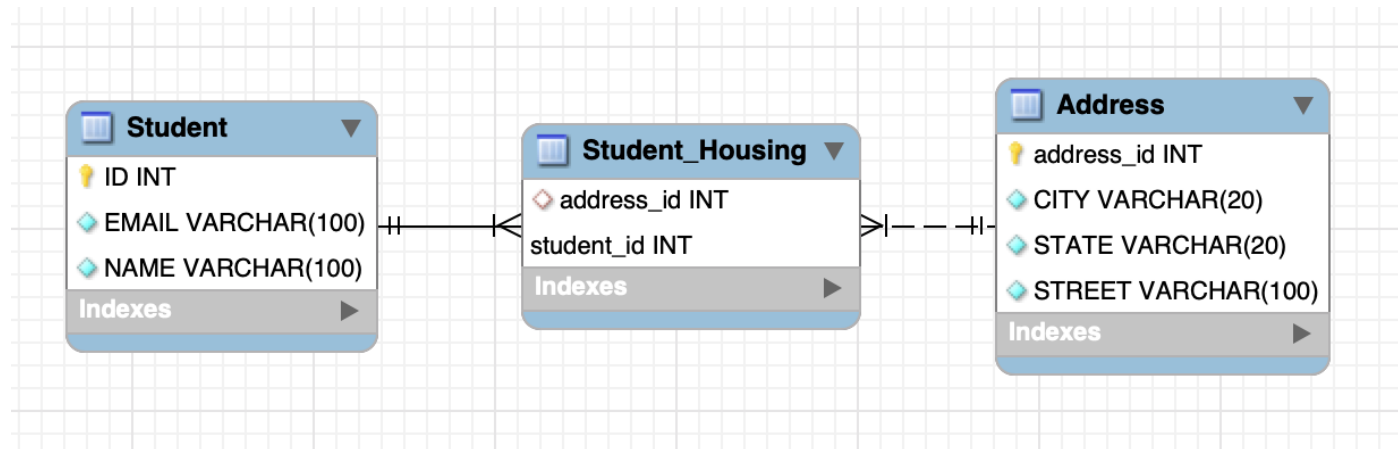
Hibernate: alter table Student_Housing add index FK9415A9992DE3DD42 (student_id), add constraint FK9415A9992DE3DD42 foreign key (student_id) references Student (ID)

Hibernate: insert into Address (CITY, STATE, STREET) values (?, ?, ?)

Hibernate: insert into Student (EMAIL, NAME) values (?, ?)

Hibernate: insert into Student_Housing (address_id, student_id) values (?, ?)

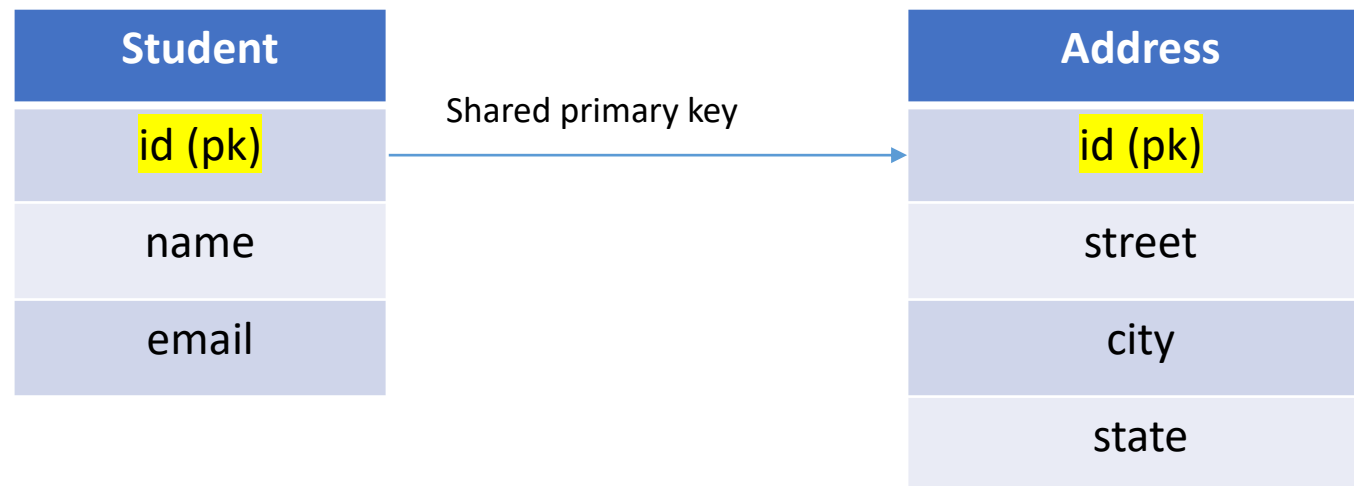
Check in MySQL Workbench:



1.3 One-to-One with @MapsId

1.3 One-to-One with @MapsId

- ❑ In this technique, hibernate assumes both the source and target share the same primary key values.
- ❑ Table structure will be like this:



1.3 One-to-One with @MapsId

- In this approach, **@MapsId** is the main annotation to be used.

Student.java

```
@OneToOne
@MapsId
private Address address;
```

Address.java

```
@OneToOne(mappedBy="address", cascade=CascadeType.ALL)
private Student student;
```

1.2 One-to-One with a common join table

Console

Hibernate: create table Address (ID integer not null auto_increment unique, CITY varchar(20) not null, STATE varchar(20) not null, STREET varchar(100) not null, primary key (ID))

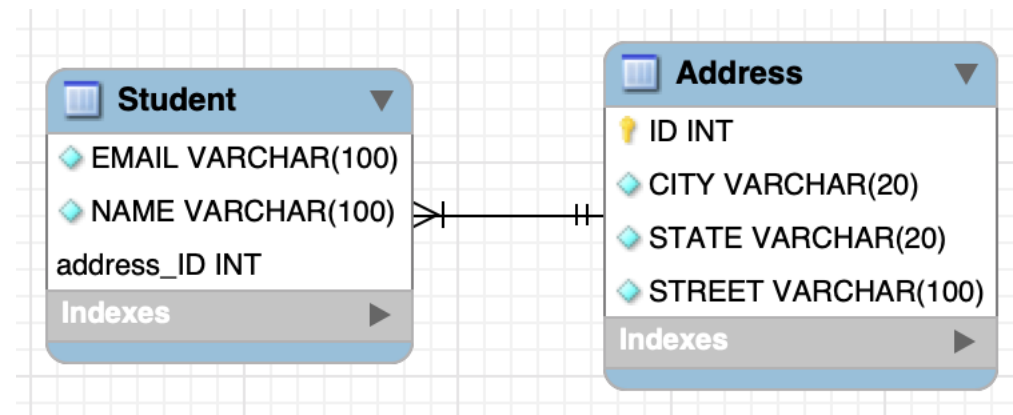
Hibernate: create table Student (EMAIL varchar(100) not null unique, NAME varchar(100) not null, address_ID integer not null, primary key (address_ID))

Hibernate: alter table Student add index FKF3371A1BE6F556A2 (address_ID), add constraint FKF3371A1BE6F556A2 foreign key (address_ID) references Address (ID)

Hibernate: insert into Address (CITY, STATE, STREET) values (?, ?, ?)

Hibernate: insert into Student (EMAIL, NAME, address_ID) values (?, ?, ?)

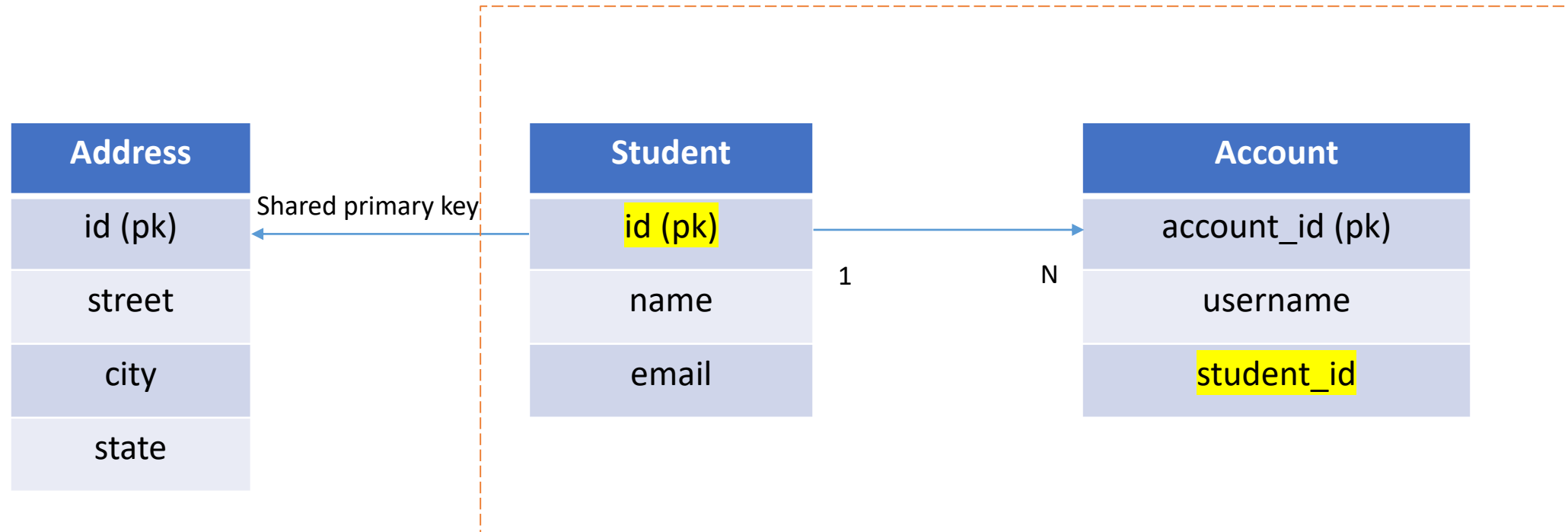
Check in MySQL Workbench:



2.1 One-to-Many with foreign key association

2.1 One-to-Many with foreign key association

- ❑ **Hibernate one to many mapping** is made between two entities where first entity can have relation with multiple second entity instances but second can be associated with only one instance of first entity. Its **1 to N** relationship.
- ❑ For example, in this demo, we assume one student can have multiple LMS accounts, while each account can only be associated with one student



2.1 One-to-Many with foreign key association

❑ Account entity class:

```
@Entity(name = "ForeignKeyAssoAccountEntity")
@Table(name = "ACCOUNT", uniqueConstraints = {
    @UniqueConstraint(columnNames = "ACCOUNT_ID")})
public class Account {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    @Column(name = "ACCOUNT_ID", unique = true, nullable = false)
    private Integer account_id;

    @Column(name = "USERNAME", unique = true, nullable = false, length = 20)
    private String username;

    @ManyToOne
    private Student student;

    ...Getters and Setters...

    @Override
    public String toString() {
        return "Account [account_id=" + account_id + ", username=" + username + ", student=" + student.getName() + "]";
    }
}
```

2.1 One-to-Many with foreign key association

❑ Student entity class:

```
@Entity
@Table(name="STUDENT", uniqueConstraints = {
    @UniqueConstraint(columnNames = "ID"),
    @UniqueConstraint(columnNames = "EMAIL")})
public class Student {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    @Column(name = "ID", unique = true, nullable = false)
    private int id;

    @Column(name = "NAME", unique = false, nullable = false, length = 100)
    private String name;

    @Column(name = "EMAIL", unique = true, nullable = false, length = 100)
    private String email;

    @OneToOne
    @JoinColumn(name="address_id")
    private Address address;

    @OneToMany(cascade=CascadeType.ALL)
    @JoinColumn(name="Student_ID")
    private List<Account> accounts = new ArrayList<Account>();

    @Override
    public String toString() {...}
}
```

2.1 One-to-Many with foreign key association

```
Configuration con = new Configuration().configure()
.addAnnotatedClass(Student.class)
.addAnnotatedClass(Address.class)
.addAnnotatedClass(Account.class);

// Prepare the transaction
ServiceRegistry serviceRegistry = new
ServiceRegistryBuilder().applySettings(con.getProperties()).buildServiceRegistry();

SessionFactory sf = con.buildSessionFactory(serviceRegistry);
Session session = sf.openSession();
Transaction tx = session.beginTransaction();

// ***** create 2 students *****
Student student_1 = new Student();
student_1.setEmail("student_1@mail.com");
student_1.setName("Happy Name");

Student student_2 = new Student();
student_2.setEmail("student_2@mail.com");
student_2.setName("Creamy Name");

// ***** create 2 addresses *****
Address address_1 = new Address();
address_1.setStreet("1st Happy Street");
address_1.setCity("New York");
address_1.setState("NY");

Address address_2 = new Address();
address_2.setStreet("2nd Icecream Road");
address_2.setCity("Atlanta");
address_2.setState("GA");
```

continues from the left...

```
// ***** Assign addresses to students *****
student_1.setAddress(address_1);
student_2.setAddress(address_2);
// Save addresses to session
session.saveOrUpdate(address_1);
session.saveOrUpdate(address_2);
//

// ***** create 3 accounts *****
Account account1 = new Account();
account1.setUsername("happy_1");
Account account2 = new Account();
account2.setUsername("happy_2");
Account account3 = new Account();
account3.setUsername("icecream_1");

// ***** Assign accounts to students *****
student_1.getAccounts().add(account1);
student_1.getAccounts().add(account2);
student_2.getAccounts().add(account3);

// ***** Save students *****
session.save(student_1);
session.save(student_2);

tx.commit();
```

2.1 One-to-Many with foreign key association

Console

Hibernate: create table ACCOUNT (ACCOUNT_ID integer not null auto_increment unique, USERNAME varchar(20) not null unique, student_ID integer, primary key (ACCOUNT_ID))
Hibernate: create table Address (ID integer not null auto_increment unique, CITY varchar(20) not null, STATE varchar(20) not null, STREET varchar(100) not null, primary key (ID))
Hibernate: create table STUDENT (ID integer not null auto_increment unique, EMAIL varchar(100) not null unique, NAME varchar(100) not null, address_id integer, primary key (ID), unique (EMAIL))

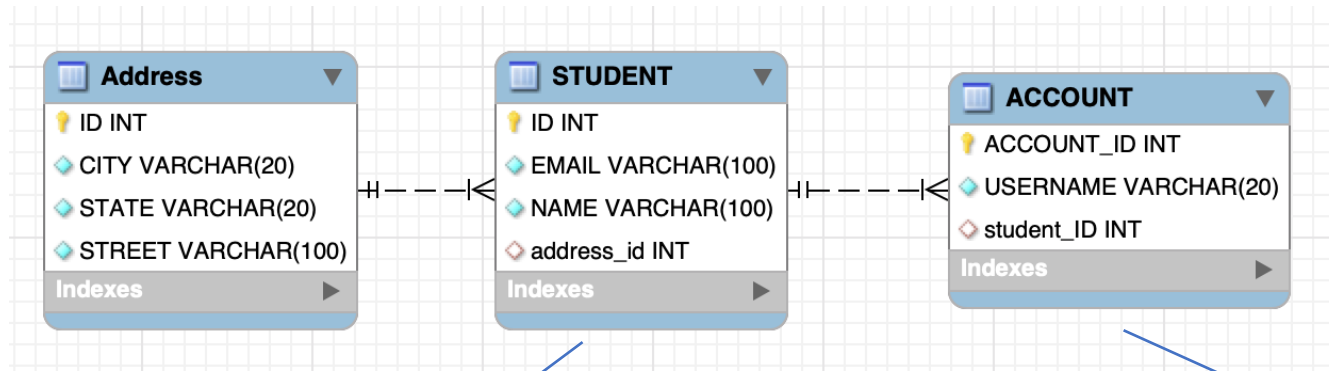
Hibernate: alter table ACCOUNT add index FKE49F160D2DE3DD42 (student_ID), add constraint FKE49F160D2DE3DD42 foreign key (student_ID) references STUDENT (ID)
Hibernate: alter table STUDENT add index FKBACA0E1BE6F556A2 (address_id), add constraint FKBACA0E1BE6F556A2 foreign key (address_id) references Address (ID)

Hibernate: insert into Address (CITY, STATE, STREET) values (?, ?, ?)
Hibernate: insert into Address (CITY, STATE, STREET) values (?, ?, ?)
Hibernate: insert into STUDENT (address_id, EMAIL, NAME) values (?, ?, ?)
Hibernate: insert into ACCOUNT (student_ID, USERNAME) values (?, ?)
Hibernate: insert into ACCOUNT (student_ID, USERNAME) values (?, ?)
Hibernate: insert into STUDENT (address_id, EMAIL, NAME) values (?, ?, ?)
Hibernate: insert into ACCOUNT (student_ID, USERNAME) values (?, ?)

Hibernate: update ACCOUNT set Student_ID=? where ACCOUNT_ID=?
Hibernate: update ACCOUNT set Student_ID=? where ACCOUNT_ID=?
Hibernate: update ACCOUNT set Student_ID=? where ACCOUNT_ID=?

2.1 One-to-Many with foreign key association

Check in MySQL Workbench



14 • `select * from student;`

15 •

100%	↕	20:6	
Result Grid			
Filter Rows: <input type="text"/>			
ID	EMAIL	NAME	address_id
1	student_1@mail.com	Happy Name	1
2	student_2@mail.com	Creamy Name	2
NULL	NULL	NULL	NULL

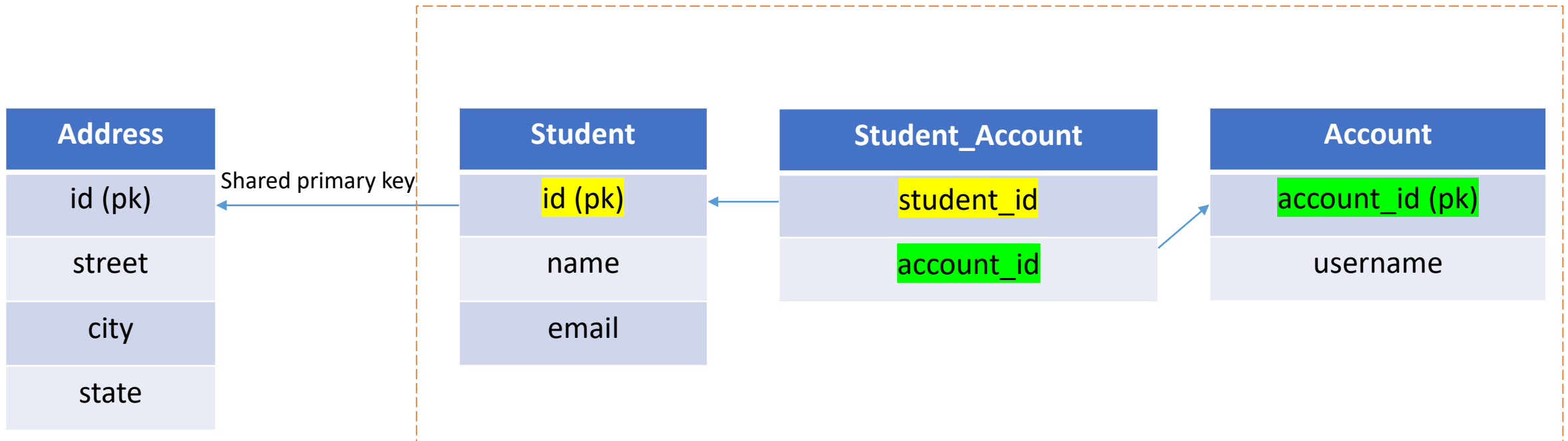
14 • `select * from account;`

100%	↕	1:4	
Result Grid			
Filter Rows: <input type="text"/>			
ACCOUNT_ID	USERNAME	student_ID	
1	happy_1	1	
2	happy_2	1	
3	icecream_1	2	
NULL	NULL	NULL	

2.2 One-to-Many with join table

2.2 One-to-Many with foreign key association

- ❑ This approach uses a **join table** to store the associations between account and employee entities.



2.2 One-to-Many with join table

□ Account entity class:

```
@Entity(name = "ForeignKeyAssoAccountEntity")
@Table(name = "ACCOUNT", uniqueConstraints = {
    @UniqueConstraint(columnNames = "ACCOUNT_ID")})
public class Account {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    @Column(name = "ACCOUNT_ID", unique = true, nullable = false)
    private Integer account_id;

    @Column(name = "USERNAME", unique = true, nullable = false, length = 20)
    private String username;

    // Getters and Setters

    @Override
    public String toString() {
        return "Account [account_id=" + account_id + ", username=" + username + "]";
    }
}
```

Notice that, we don't have to mention student in account class

2.2 One-to-Many with join table

❑ Student entity class:

```
@Entity
@Table(name="STUDENT", uniqueConstraints = {
    @UniqueConstraint(columnNames = "ID"),
    @UniqueConstraint(columnNames = "EMAIL")})
public class Student {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    @Column(name = "ID", unique = true, nullable = false)
    private int id;

    @Column(name = "NAME", unique = false, nullable = false, length = 100)
    private String name;

    @Column(name = "EMAIL", unique = true, nullable = false, length = 100)
    private String email;

    @OneToOne
    @JoinColumn(name="address_id")
    private Address address;

    @OneToMany(cascade=CascadeType.ALL)
    @JoinColumn(name="Student_ID")
    private List<Account> accounts = new ArrayList<Account>();

    @Override
    public String toString() {...}
}
```

Compare to foreignkey association, just remove the @JoinColumn

2.2 One-to-Many with join table

Console

Hibernate: create table ACCOUNT (ACCOUNT_ID integer not null auto_increment unique, USERNAME varchar(20) not null unique, primary key (ACCOUNT_ID))

Hibernate: create table Address (ID integer not null auto_increment unique, CITY varchar(20) not null, STATE varchar(20) not null, STREET varchar(100) not null, primary key (ID))

Hibernate: create table STUDENT (ID integer not null auto_increment unique, EMAIL varchar(100) not null unique, NAME varchar(100) not null, address_id integer, primary key (ID), unique (EMAIL))

Hibernate: create table **STUDENT_ACCOUNT** (STUDENT_ID integer not null, accounts_ACCOUNT_ID integer not null, unique (accounts_ACCOUNT_ID))

Hibernate: alter table STUDENT add index FKBACA0E1BE6F556A2 (address_id), add constraint FKBACA0E1BE6F556A2 foreign key (address_id) references Address (ID)

Hibernate: alter table STUDENT_ACCOUNT add index FK23427FE93EE8CDFB (accounts_ACCOUNT_ID), add constraint FK23427FE93EE8CDFB foreign key (accounts_ACCOUNT_ID) references ACCOUNT (ACCOUNT_ID)

Hibernate: alter table STUDENT_ACCOUNT add index FK23427FE92DE3DD42 (STUDENT_ID), add constraint FK23427FE92DE3DD42 foreign key (STUDENT_ID) references STUDENT (ID)

Hibernate: insert into Address (CITY, STATE, STREET) values (?, ?, ?)

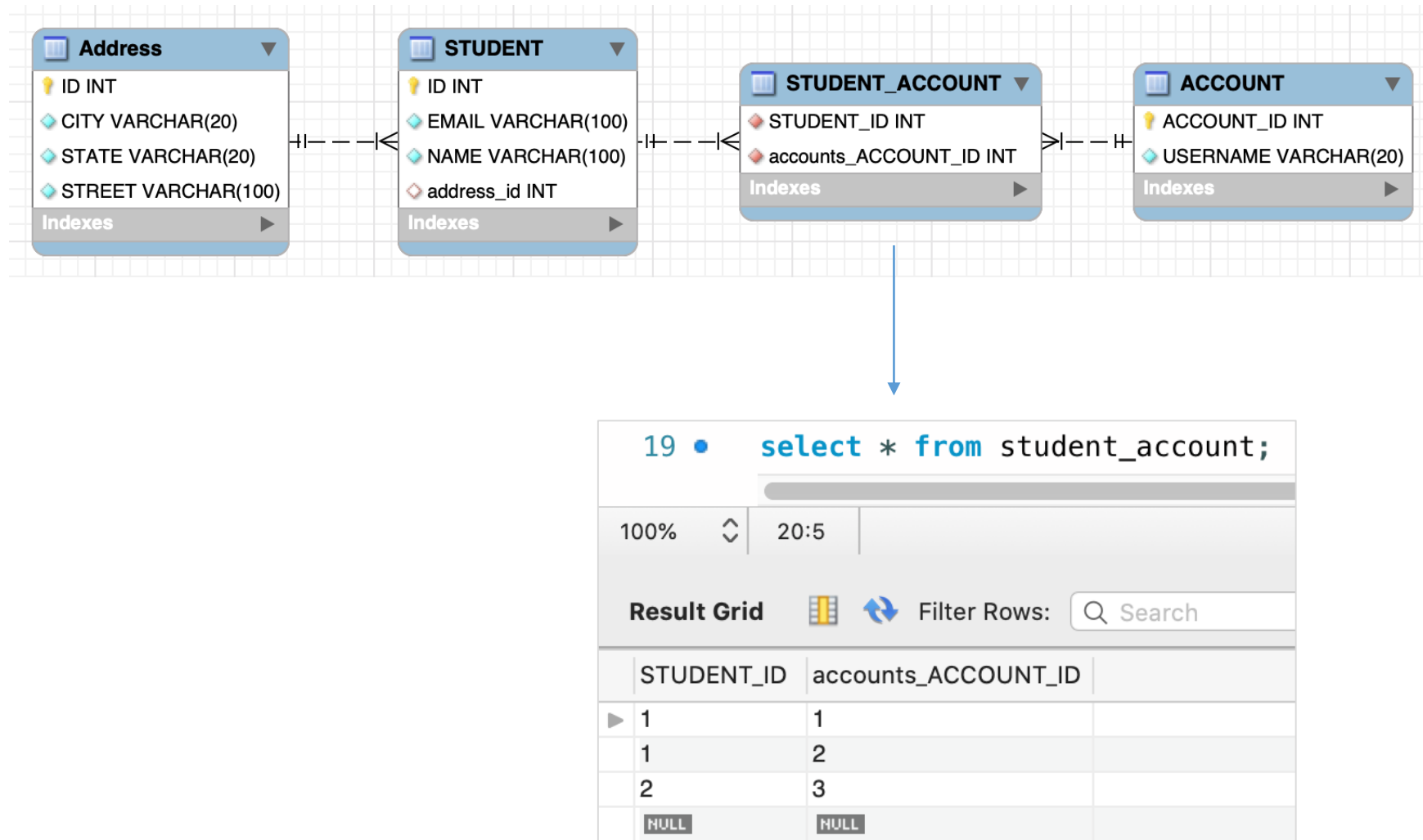
Hibernate: insert into Address (CITY, STATE, STREET) values (?, ?, ?)

Hibernate: insert into STUDENT (address_id, EMAIL, NAME) values (?, ?, ?)

Hibernate: insert into ACCOUNT (USERNAME) values (?)

2.1 One-to-Many with foreign key association

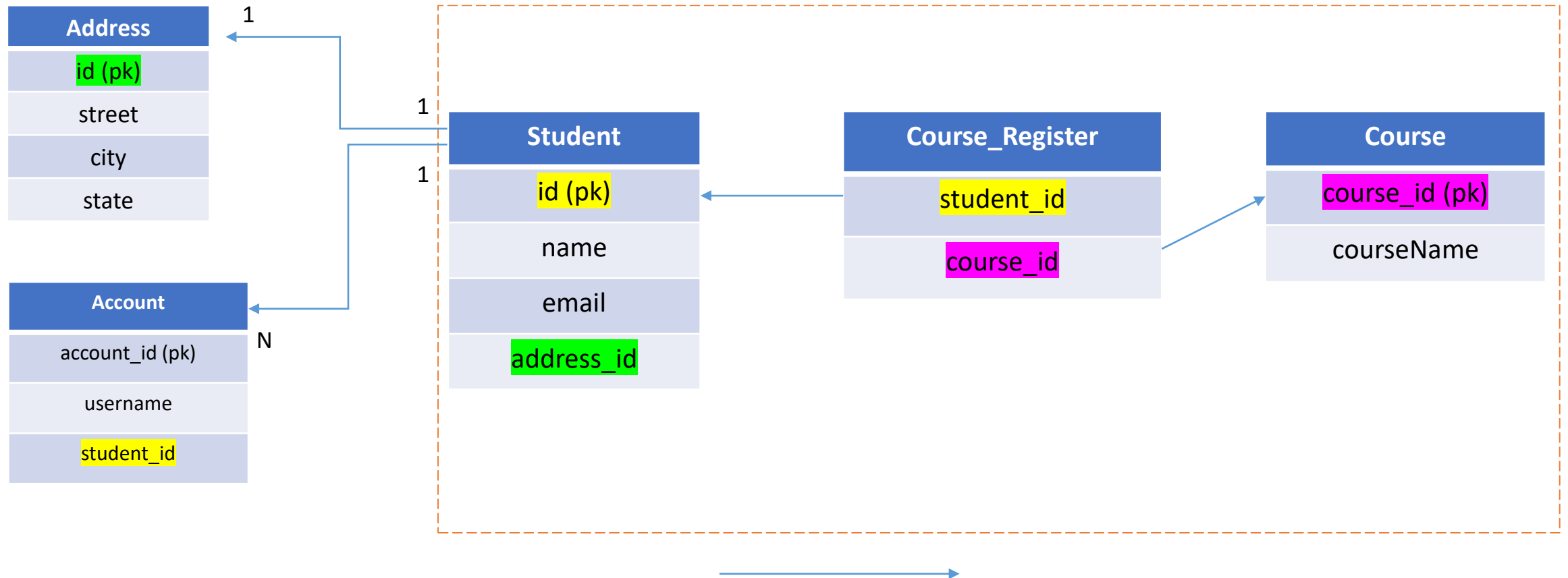
Check in MySQL Workbench



3. Many-to-Many

3. Many-to-Many with join table

- ❑ **Hibernate many to many mapping** is made between two entities where one can have relation with multiple other entity instances.
- ❑ For example, in this demo, we assume every student can register multiple training courses, and each course can be registered by multiple students



3. Many-to-Many with join table

❑ Course entity class:

```
@Entity
@Table(name = "COURSE", uniqueConstraints = {
    @UniqueConstraint(columnNames = "ID")})
public class Course {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    @Column(name = "ID", unique = true, nullable = false)
    private int id;

    @Column(name = "Course_Name", unique = true, nullable = false, length = 100)
    private String courseName;

    @ManyToMany(mappedBy="courses")
    private List<Student> students = new ArrayList<Student>();

    ...Getters and Setters...

}
```

3. Many-to-Many with join table

❑ Student entity class:

```
@Entity
@Table(name="STUDENT", uniqueConstraints = {
    @UniqueConstraint(columnNames = "ID"),
    @UniqueConstraint(columnNames = "EMAIL")})
public class Student {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    @Column(name = "ID", unique = true, nullable = false)
    private int id;

    @Column(name = "NAME", unique = false, nullable = false, length = 100)
    private String name;

    @Column(name = "EMAIL", unique = true, nullable = false, length = 100)
    private String email;

    @OneToOne
    @JoinColumn(name="address_id")
    private Address address;

    @OneToMany(cascade=CascadeType.ALL)
    @JoinColumn(name="Student_ID")
    private List<Account> accounts = new ArrayList<Account>();

    @ManyToMany(cascade=CascadeType.ALL)
    @JoinTable(name="Course_Register", joinColumns={@JoinColumn(referencedColumnName="ID")},
        inverseJoinColumns={@JoinColumn(referencedColumnName="ID")})
    private List<Course> courses = new ArrayList<Course>();
}
```

3. Many-to-Many with join table

```
Configuration con = new Configuration().configure()
.addAnnotatedClass(Student.class).addAnnotatedClass(Address.class)
.addAnnotatedClass(Account.class).addAnnotatedClass(Course.class);

// Prepare the transaction
ServiceRegistry serviceRegistry = new
ServiceRegistryBuilder().applySettings(con.getProperties()).buildServiceRegistry();

SessionFactory sf = con.buildSessionFactory(serviceRegistry);
Session session = sf.openSession();
Transaction tx = session.beginTransaction();

// ***** create 2 students *****
Student student_1 = new Student();
student_1.setEmail("student_1@mail.com");
student_1.setName("Happy Name");

Student student_2 = new Student();
student_2.setEmail("student_2@mail.com");
student_2.setName("Creamy Name");

// ***** create 2 addresses *****
Address address_1 = new Address();
address_1.setStreet("1st Happy Street");
address_1.setCity("New York");
address_1.setState("NY");

Address address_2 = new Address();
address_2.setStreet("2nd Icecream Road");
address_2.setCity("Atlanta");
address_2.setState("GA");

// Assign addresses to students
student_1.setAddress(address_1);
student_2.setAddress(address_2);

// Save addresses to session
session.saveOrUpdate(address_1);
session.saveOrUpdate(address_2);
```

continues from the left...

```
// ***** create 3 accounts *****
Account account1 = new Account();
account1.setUsername("happy_1");
Account account2 = new Account();
account2.setUsername("happy_2");
Account account3 = new Account();
account3.setUsername("icecream_1");

// Assign accounts to students
student_1.getAccounts().add(account1);
student_1.getAccounts().add(account2);
student_2.getAccounts().add(account3);

// ***** create 4 courses *****
Course c1 = new Course();
c1.setCourseName("Java");
Course c2 = new Course();
c2.setCourseName("Python");
Course c3 = new Course();
c3.setCourseName("JavaScript");
Course c4 = new Course();
c4.setCourseName("C++");

// Assign courses to students
student_1.getCourses().add(c1);
student_1.getCourses().add(c4);
student_2.getCourses().add(c2);
student_2.getCourses().add(c3);
student_2.getCourses().add(c4);

// ***** Save students *****
session.save(student_1);
session.save(student_2);

tx.commit();
```


3. Many-to-Many with join table

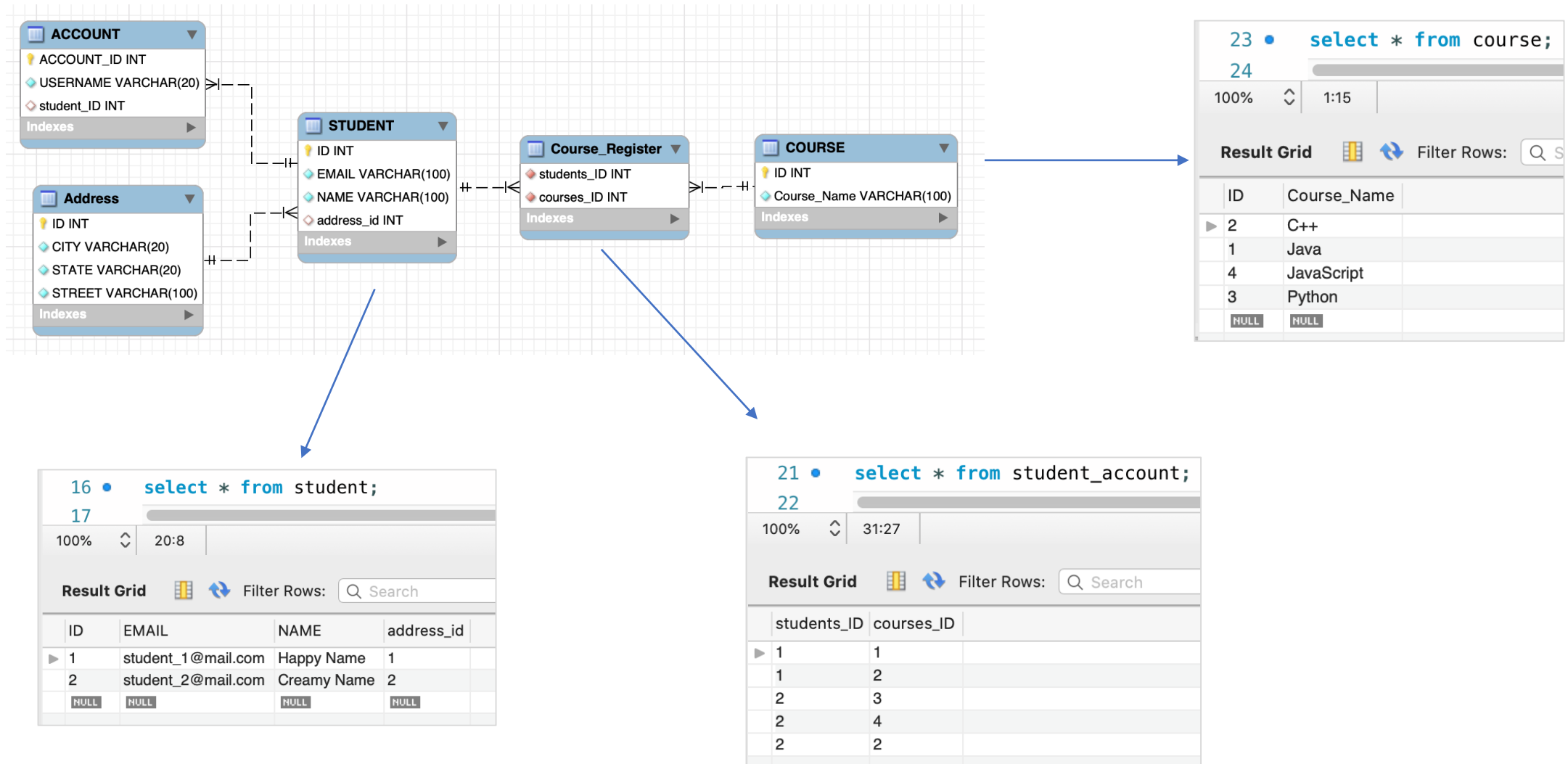
Console

```
Hibernate: create table ACCOUNT (ACCOUNT_ID integer not null auto_increment unique, USERNAME varchar(20) not null unique, student_ID integer, primary key (ACCOUNT_ID))
Hibernate: create table Address (ID integer not null auto_increment unique, CITY varchar(20) not null, STATE varchar(20) not null, STREET varchar(100) not null, primary key (ID))
Hibernate: create table COURSE (ID integer not null auto_increment unique, Course_Name varchar(100) not null unique, primary key (ID))
Hibernate: create table Course_Register (students_ID integer not null, courses_ID integer not null)
Hibernate: create table STUDENT (ID integer not null auto_increment unique, EMAIL varchar(100) not null unique, NAME varchar(100) not null, address_id integer, primary key (ID), unique (EMAIL))
Hibernate: alter table ACCOUNT add index FKE49F160D2DE3DD42 (student_ID), add constraint FKE49F160D2DE3DD42 foreign key (student_ID) references STUDENT (ID)
Hibernate: alter table Course_Register add index FK6668A47F7D000C5 (students_ID), add constraint FK6668A47F7D000C5 foreign key (students_ID) references STUDENT (ID)
Hibernate: alter table Course_Register add index FK6668A476BE44CB5 (courses_ID), add constraint FK6668A476BE44CB5 foreign key (courses_ID) references COURSE (ID)
Hibernate: alter table STUDENT add index FKBACA0E1BE6F556A2 (address_id), add constraint FKBACA0E1BE6F556A2 foreign key (address_id) references Address (ID)

Hibernate: insert into Address (CITY, STATE, STREET) values (?, ?, ?)
Hibernate: insert into Address (CITY, STATE, STREET) values (?, ?, ?)
Hibernate: insert into STUDENT (address_id, EMAIL, NAME) values (?, ?, ?)
Hibernate: insert into ACCOUNT (student_ID, USERNAME) values (?, ?)
Hibernate: insert into ACCOUNT (student_ID, USERNAME) values (?, ?)
Hibernate: insert into COURSE (Course_Name) values (?)
Hibernate: insert into COURSE (Course_Name) values (?)
Hibernate: insert into STUDENT (address_id, EMAIL, NAME) values (?, ?, ?)
Hibernate: insert into ACCOUNT (student_ID, USERNAME) values (?, ?)
Hibernate: insert into COURSE (Course_Name) values (?)
Hibernate: insert into COURSE (Course_Name) values (?)
Hibernate: update ACCOUNT set Student_ID=? where ACCOUNT_ID=?
Hibernate: update ACCOUNT set Student_ID=? where ACCOUNT_ID=?
Hibernate: insert into Course_Register (students_ID, courses_ID) values (?, ?)
Hibernate: insert into Course_Register (students_ID, courses_ID) values (?, ?)
Hibernate: update ACCOUNT set Student_ID=? where ACCOUNT_ID=?
Hibernate: insert into Course_Register (students_ID, courses_ID) values (?, ?)
Hibernate: insert into Course_Register (students_ID, courses_ID) values (?, ?)
Hibernate: insert into Course_Register (students_ID, courses_ID) values (?, ?)
```

3. Many-to-Many with join table

Check in MySQL Workbench



Hibernate Lazy loading

Why we need lazy loading in Hibernate?

- ❑ Consider one of common Internet web application: the online store. The store maintains a catalog of products. At the crudest level, this can be modeled as a catalog entity managing a series of product entities. In a large store, there may be tens of thousands of products grouped into various overlapping categories.
- ❑ When a customer visits the store, the catalog must be loaded from the database. We probably don't want the implementation to load every single one of the entities representing the tens of thousands of products to be loaded into memory. For a sufficiently large retailer, this might not even be possible, given the amount of physical memory available on the machine.
- ❑ Even if this was possible, it would probably cripple the performance of the site. Instead, we want only the catalog to load, possibly with the categories as well. Only when the user drills down into the categories should a subset of the products in that category be loaded from the database.
- ❑ To manage this problem, Hibernate provides a facility called **lazy loading**. When enabled, an entity's associated entities will be loaded only when they are directly requested.

How to enable lazy loading in hibernate

❑ The **default behavior** of lazy loading in case of using hibernate mappings vs annotations:

- Load 'property values eagerly' and to load 'collections lazily'. (Contrary to plain Hibernate 2 (mapping files) before, where all references (including collections) are loaded eagerly by default)
- **@OneToMany** and **@ManyToMany** associations are defaulted to **LAZY loading**;
- **@OneToOne** and **@ManyToOne** are defaulted to **EAGER loading**.

❑ To enable lazy loading explicitly you must use **"fetch = FetchType.LAZY"** on a association which you want to lazy load when you are using hibernate annotations.

- A **hibernare lazy load example** will look like this:

```
@OneToMany(cascade=CascadeType.ALL, fetch = FetchType.LAZY)  
@JoinColumn(name="Student_ID")  
private List<Account> accounts = new ArrayList<Account>();
```

❑ Another attribute parallel to "FetchType.LAZY" is **"FetchType.EAGER"** which is just opposite to LAZY i.e. it will load association entity as well when owner entity is fetched first time.

Entity LifeCycle

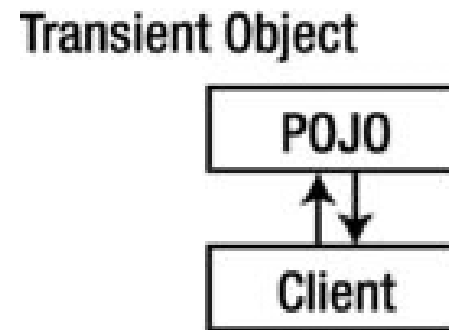
(TYI)

Entity LifeCycle

- ❑ We know that Hibernate works with plain Java objects (POJO). In raw form (without hibernate specific annotations), hibernate will not be able to identify these java classes. But when these POJOs are properly annotated with required annotations then hibernate will be able to identify them and then work with them e.g. store in the database, update them, etc. These POJOs are said to mapped with hibernate.
- ❑ Given an instance of a class that is mapped to Hibernate, it can be in any one of four different persistence states (known as hibernate entity lifecycle states):
 - Transient
 - Persistent
 - Detached
 - Removed

Transient

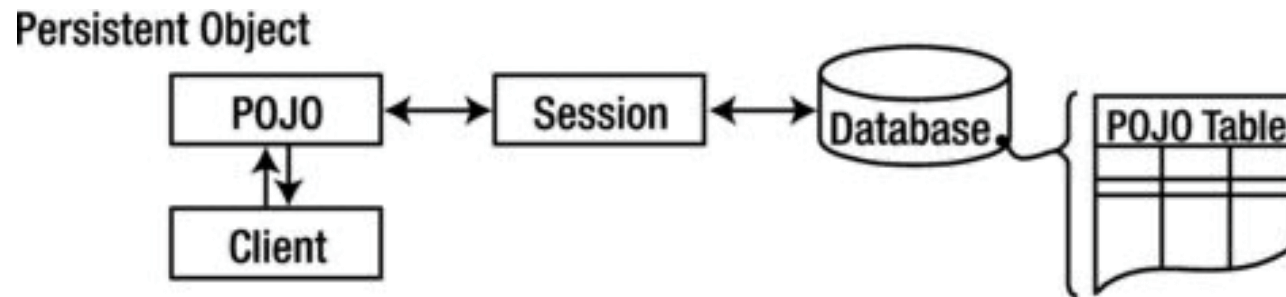
- ❑ Transient entities exist in heap memory as normal Java objects. Hibernate does not manage transient entities or persist changes done on them. **Transient objects are independent of Hibernate**



- ❑ To persist the changes to a transient entity, we would have to ask the hibernate session to save the transient object to the database, at which point Hibernate assigns the object an identifier and marks the object as being in persistent state.

Persistent

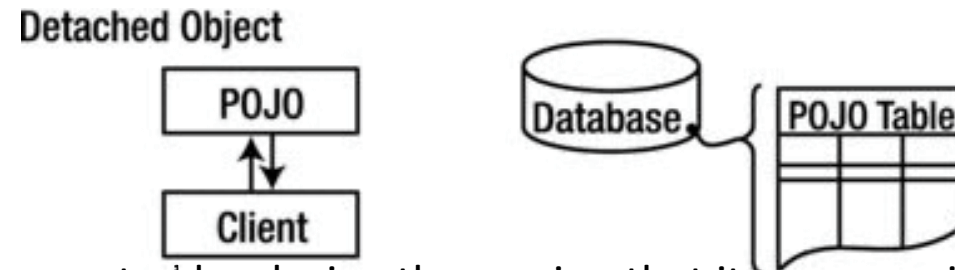
- ❑ Persistent entities exist in the database, and **Hibernate manages the persistence for persistent objects.**



- ❑ If fields or properties change on a persistent object, Hibernate will keep the database representation up to date when the application marks the changes as to be committed.

Detached

- ❑ Detached entities are those who were once persistent in the past, and now they are no longer persistent. To persist changes done in detached objects, you must re-attach them to hibernate session:



- ❑ A detached entity can be created by closing the session that it was associated with, or by evicting it from the session with a call to the session's `evict()` method.
- ❑ In order to persist changes made to a detached object, the application must re-attach it to a valid Hibernate session. A detached instance can be associated with a new Hibernate session when your application calls one of the `load()`, `refresh()`, `merge()`, `update()`, or `save()` methods on the new session with a reference to the detached object. After the method call, the detached entity would be a persistent entity managed by the new Hibernate session.

Removed

- ❑ Removed entities are persistent objects that have been passed to the session's `remove()` method and soon will be deleted as soon as changes held in the session will be committed to database.