Section 18: Introduction to Hibernate
2 / 2 | 10min

➢ What is Hibernate?

Hibernate is a framework for persisting/saving Java objects in a database, and our Java application can make use of it to save and retrieve from the DB.

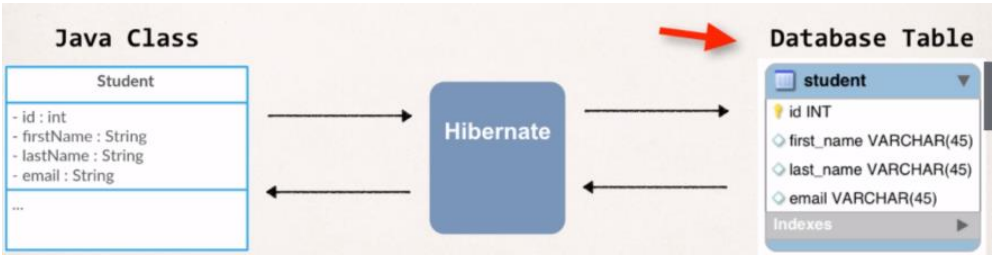
We can download it from www.hibernate.org.

Benefits of Hibernate:

- Hibernate handles all of the low-level SQL.
- Minimizes the amount of JDBC code that we need to develop.
- Hibernate actually provides the Object-to-Relational Mapping (ORM).

Object-To-Relational Mapping (ORM)

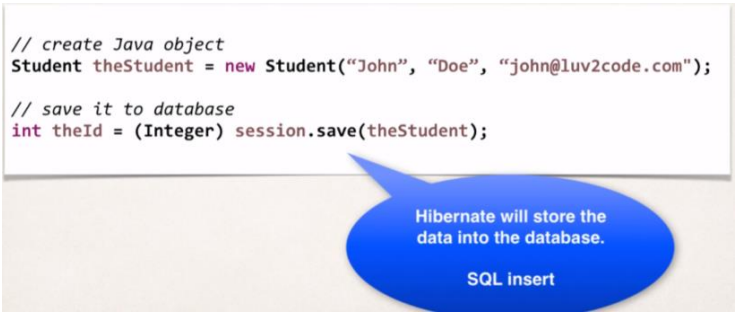
Being a Developer, we need to tell the Hibernate how our Java class or Object map to data in the DB and in fact we are going to map our Java Object to a database table.



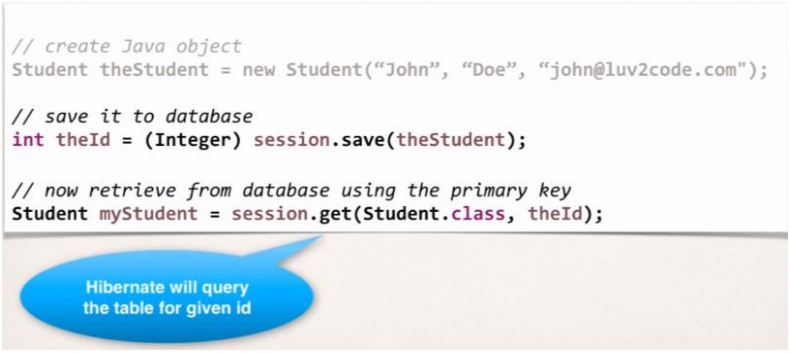
In order to set up the One-to-One mapping between the fields and columns in the Database, we can make use of XML or Java Annotations.

How to Save a Java Object with Hibernate

Session is a Hibernate object and on calling save method, hibernate will take the data Object and based on the mappings defined earlier, it will take the information and store in the Table at appropriate columns. And once we do ***session.save(theStudent)*** hibernate will return the primary key of that particular entry and we can use that at later point to retrieve that Object.



How to Retrieve the Object with Hibenate:

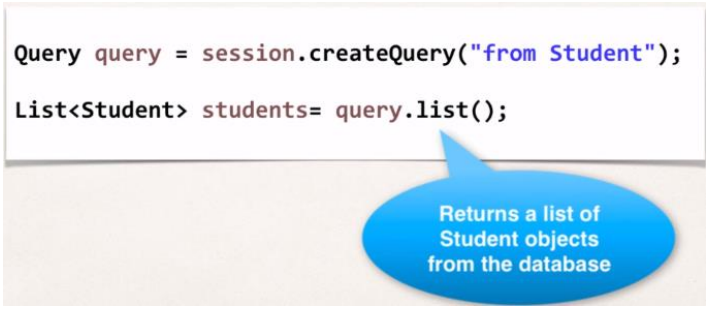


We are using the Primary Key returned by the *`session.save(theStudent)`* method, to retrieve the saved Student Object from the Database. Using *`session.get()`* to retrieve by telling it about what we want to get from DB i.e., *`Student.class`* and giving the Id of the object.

Then hibernate will go and look at the table called student and then it will find the student whose primary key is matches **`theId.`**

Querying for Java Objects:

In case we want all of the objects stores in the DB inside the Table. Hibernate has support for Query and query.list() will give us List of all the Student object.



Here we are making use of **Hibernate Query Language (HQL).**

- ## Relationship between Hibernate and JDBC

Hibernate actually uses JDBC for all the Database communication, and we can say that Hibernate is just a layer of Abstraction on top of JDBC. Therefor when our application uses the Hibernate framework then our app will store and retrieve Objects using the Hibernate API but behind the scenes Hibernate will do all the low level JDBC work and submitting the SQL query.

When we are actually Configuring hibernate to talk to our DB, we are actually configuring hibernate to make use of JDBC driver.

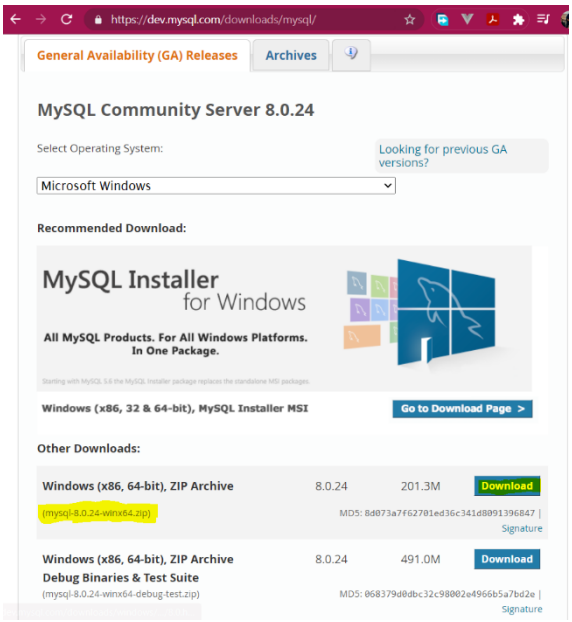
Section 19: Setting Up Hibernate Development Environment
7 / 7 | 29min

To Build Hibernate Applications, you need the following:

1. Java Integrated Development Environment (IDE)
2. Database Server
3. Hibernate JAR files and JDBC Driver

Installing MYSQL:

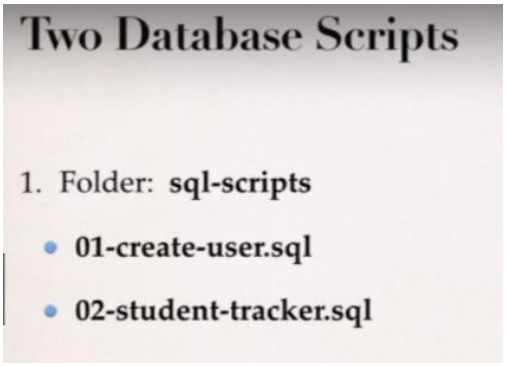
1. Visit https://dev.mysql.com/downloads/
2. Go to download the MySQL Community Server.

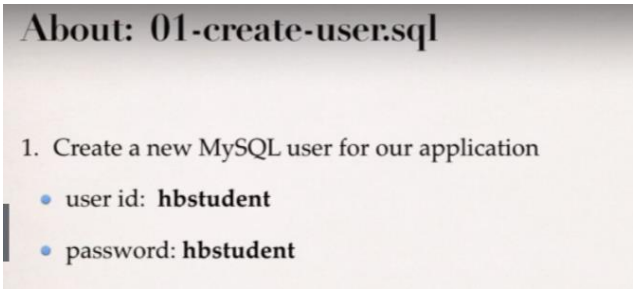


3. After downloading, start the installer.
4. Once the installer starts, select the **Developer Default** setup type, and in the Types and Networking keep all the things **default,** and at Accounts and Roles provide your root password, and in rest of the window keep things in default.

Setup Database Table

two Database Scripts that we are going to use:

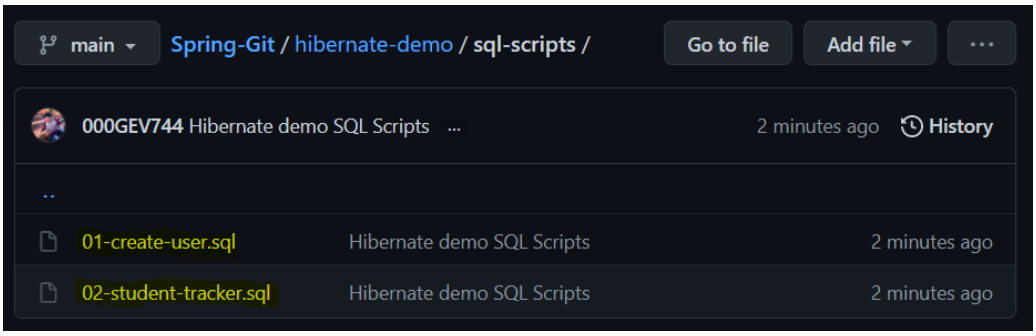




Our App will use this user id and password to connect with MySQL database.

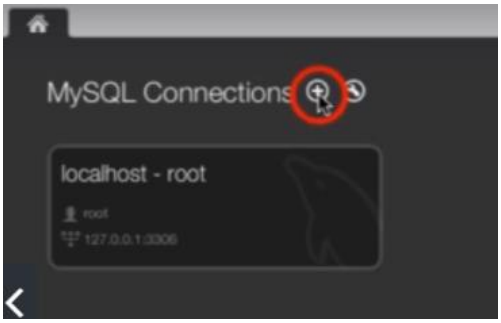
- To get the SQL Scripts visit:

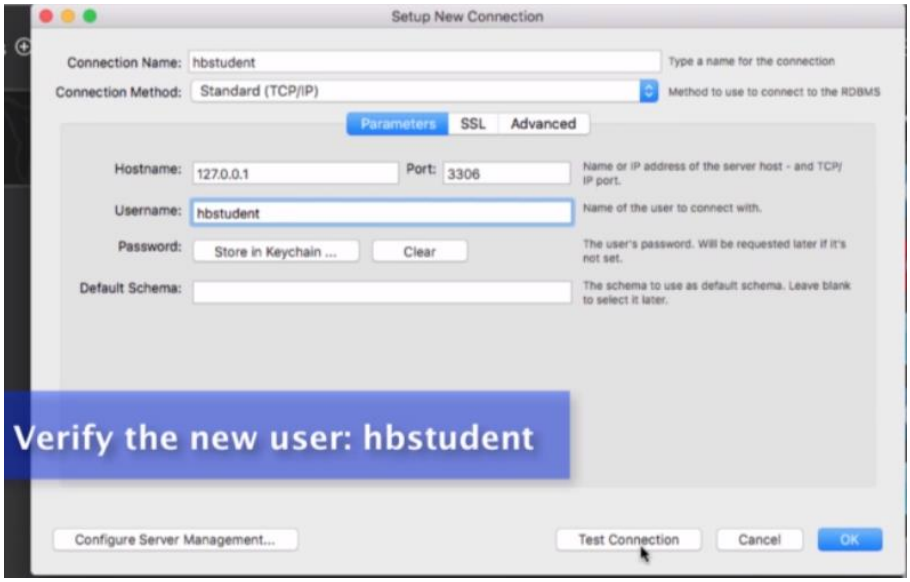
https://github.com/000GEV744/Spring-Git/tree/main/hibernate-demo



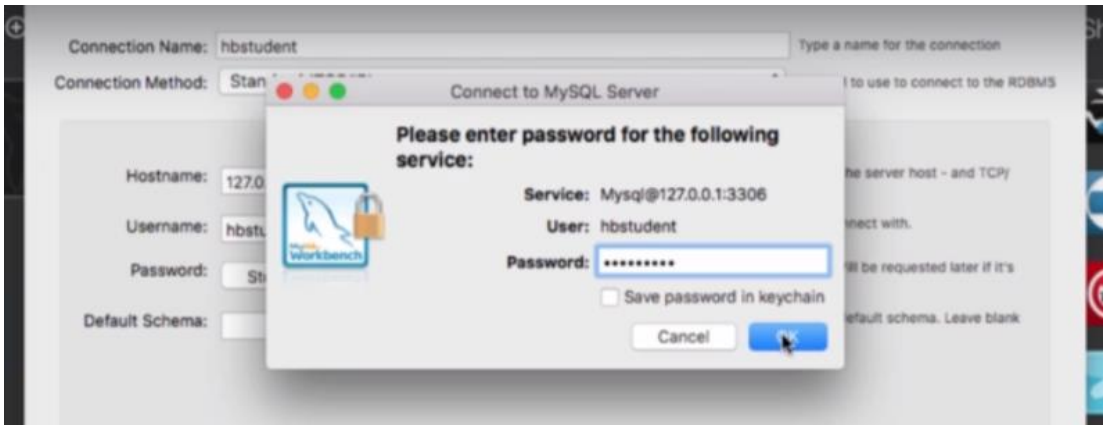
- We need to run these two SQL scripts on our SQL Workbench.

Once *create-user.sql* gets executed then our new user got created i.e., user id: **hbstudent.**Now close the root connection. And open a new Connection to verify the new user, and test connection.

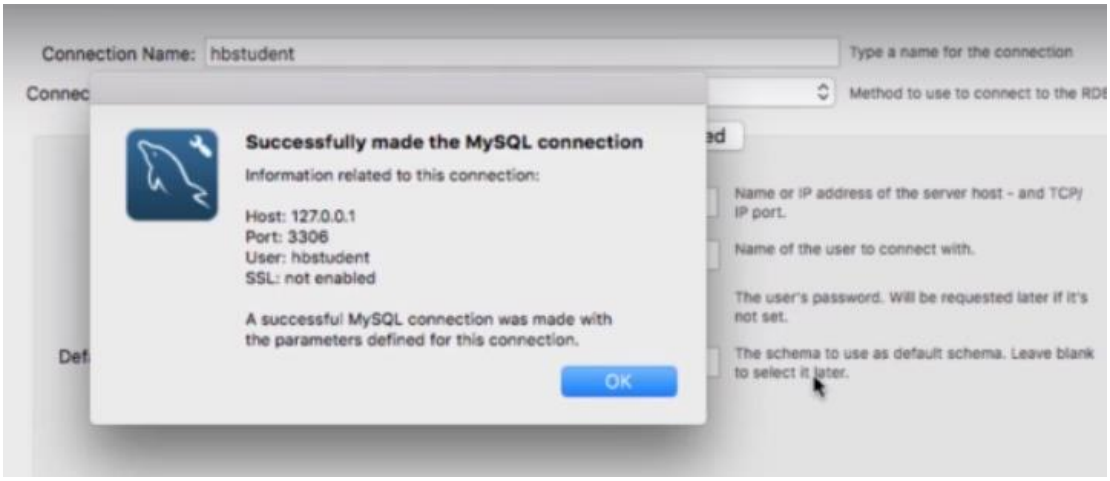




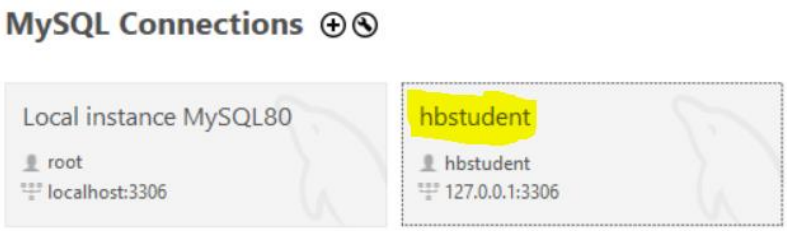
- Enter the password: hbstudent and click **OK**



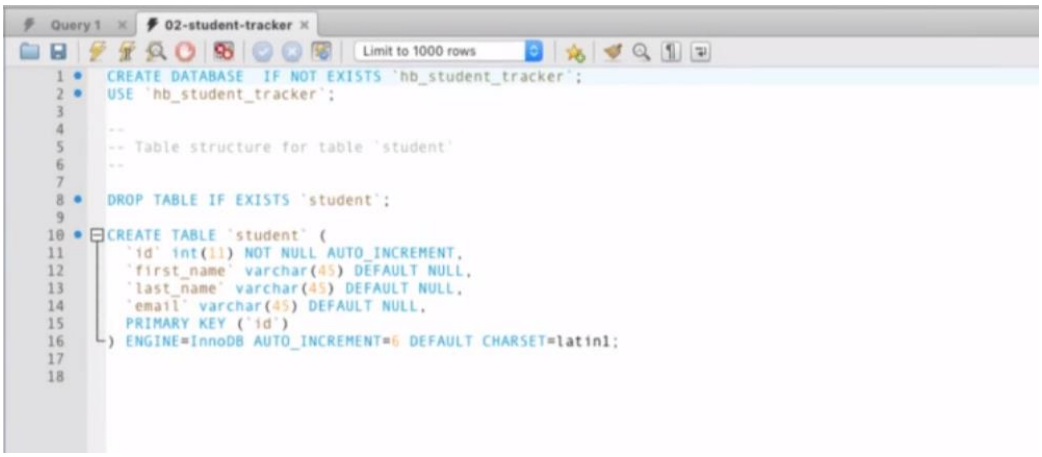
- And after that we get to see Success message as shown in below; So we know that our user has been created successfully, and now we can login with our newly created user id and password.

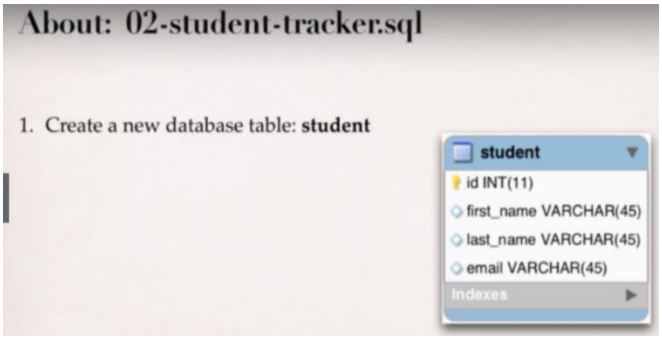


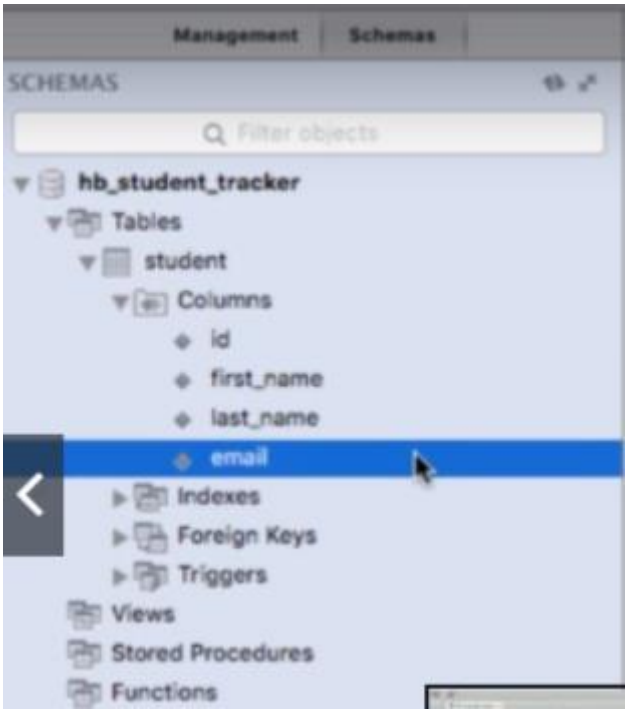
- and at last, we get to see our newly created user in the MySQL connection section at home page. Go and click on hbstudent section and connect to the DB.



About second SQL Script: Execute it into our newly created `hbstudent` connection, go on the left-hand side of workbench into the schema section and right click and do `refresh all` and then we will get to see our newly created student table.







- **Setup Hibernate in Eclipse:**

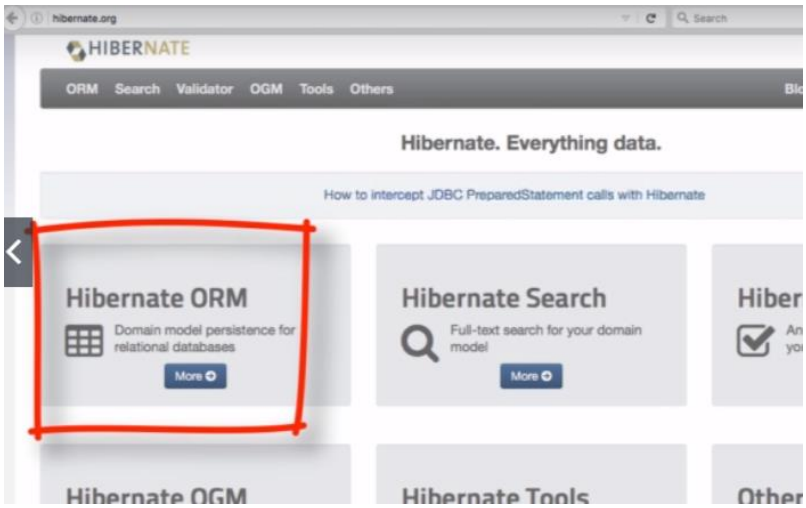
  To do List:

  1. Create the Eclipse Java Project.
  2. Download hibernate Files.
  3. Download MySQL JDBC Drivers.
  4. Add JAR Files to Eclipse Project …Build Path.

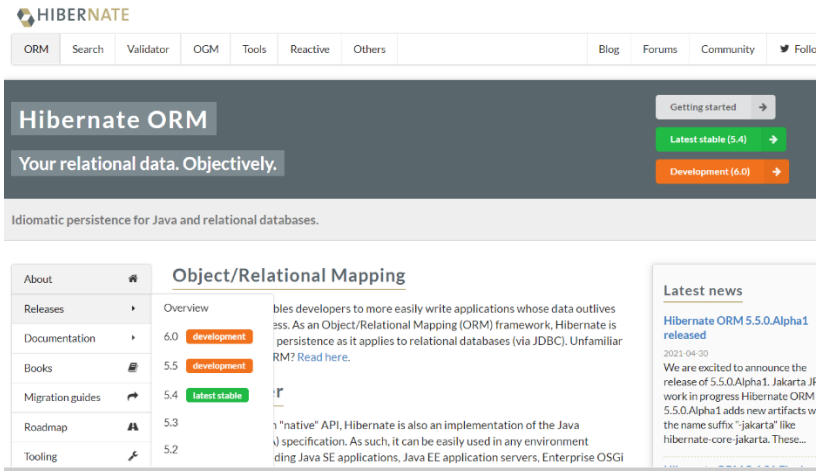
Create lib folder in your Eclipse Java Project and add Hibernate and JDBC jar files into it and later add them into your Classpath.

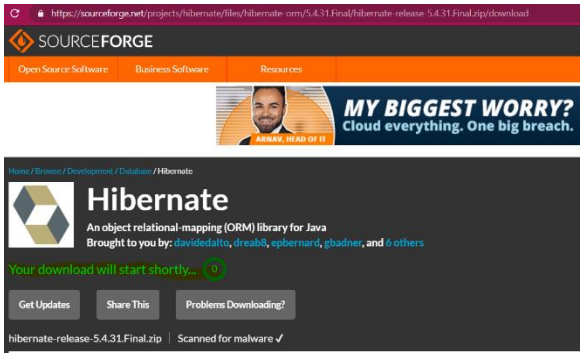
- To get the hibernate Jar files, visit www.hibernate.org

Go to **Hibernate ORM** and click on **More**

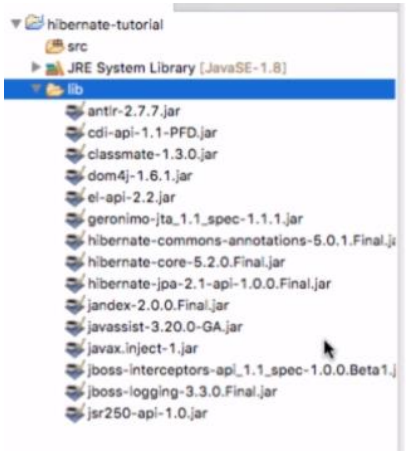


Go to releases and click on the latest stable version of hibernate ORM, and later we will redirected to the downlaod page. Unzip it after downloading.





Go to the location ***`hibernate-release-5.4.31.Final\lib\required`*** and copy all the jars available and paste into our Java project inside the **lib** folder.



- As we know Hibernate makes use of JDBC in the background so we

Need to get the MySQL JDBC driver and also add them to our

Project.

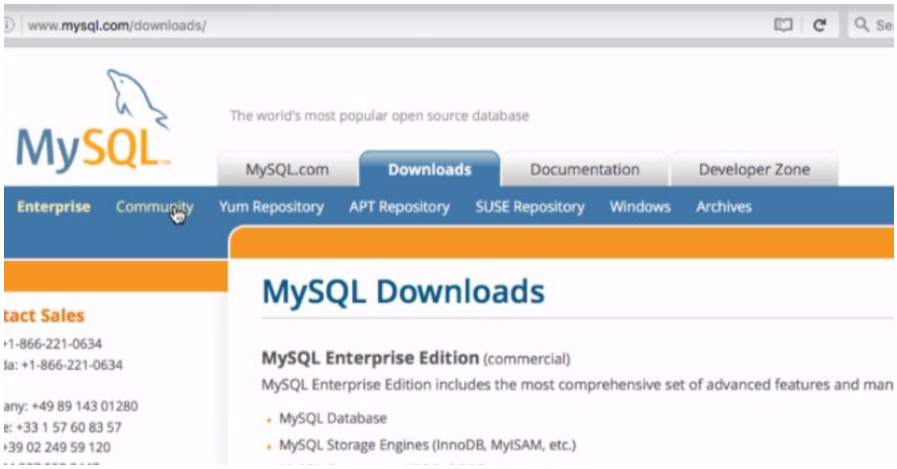
Goto Mysql website http://dev.mysql.com

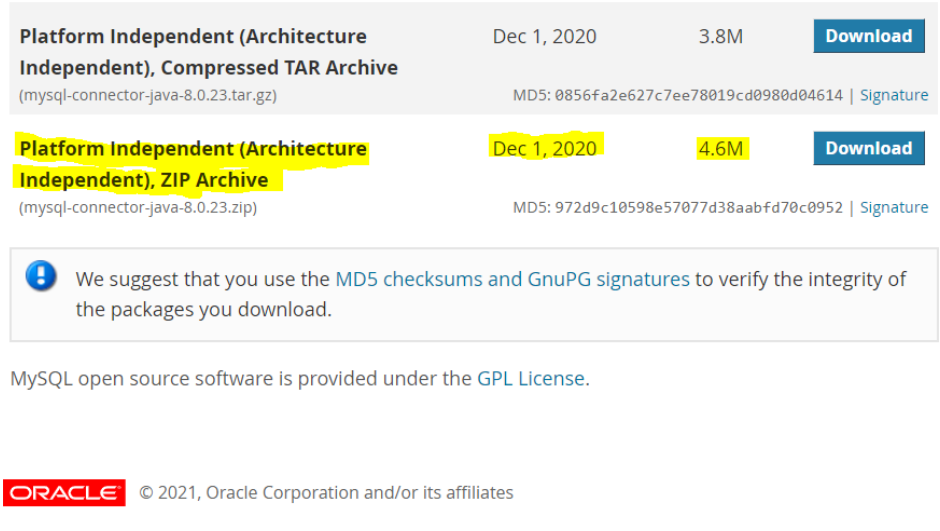
Go to downloads tab then choose the community link and after reaching

To the community page search for the MySQL Connectors, and these

Connectors will allow us to connect with MySQL.

Now click on ***Connector/J*** link which has the connectors for Java. And then select the platform independent zip folder, hit the download link then.





Now unzip the folder and then inside the folder we will get only one jar file named with ***mysql-connector-java-8.0.23*** and copy this and paste into the lib folder inside our Java Project, and similarly add them to the classpath and after that we will get a new folder into our project named with ***"referenced Libraries".***

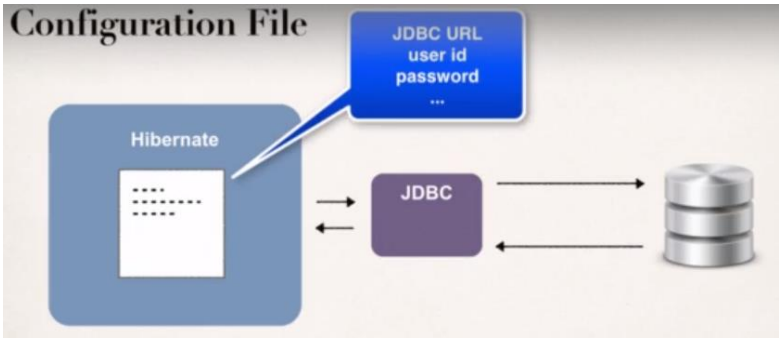
Now our Setup is ready and we can start coding now!

Section 20: Hibernate Configuration with Annotations
8 / 8 | 22min

Hibernate Development Process

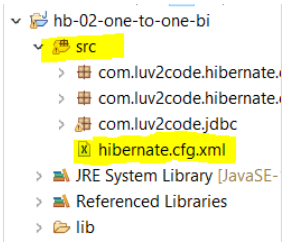
1. Add hibernate Configuration File

   Configuration File basically tells the database about how to connect to the Database. So, it will contain the bulk of the information which consist of JDBC configuration like JDBC url, userId , password and so on.



   https://github.com/000GEV744/Spring-Git/tree/main/hibernate-demo

   here we have starter files, which contains the basic hibernate configuration file (***hibernate-config.xml***), copy it and paste inside our project under the root of source directory, this makes it available for hibernate to use it, because it has to be in the actual classpath of our application. We can place it to some other locations but for simplicity we're keeping it here.




```
<!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>

        <!-- JDBC Database connection settings -->
        <property name="connection.driver_class">com.mysql.cj.jdbc.Driver</property>
        <property name="connection.url">jdbc:mysql://localhost:3306/hb-01-one-to-one-uni?useSSL=false&amp;serverTimezone=UTC</property>
        <property name="connection.username">hbstudent</property>
        <property name="connection.password">hbstudent</property>

        <!-- JDBC connection pool settings ... using built-in test pool -->
        <property name="connection.pool_size">1</property>

        <!-- Select our SQL dialect -->
        <property name="dialect">org.hibernate.dialect.MySQLDialect</property>

        <!-- Echo the SQL to stdout -->
        <property name="show_sql">true</property>

        <!-- Set the current session context -->
        <property name="current_session_context_class">thread</property>

    </session-factory>

</hibernate-configuration>
```

2. Annotate Java class

   Hibernate has the Concept of Entity Class, which is a fancy name for a Java class that is mapped to a database table. It's a simple plain old java class where we have getters and setters and have some annotations which helps us to actually map our Java class to database table.

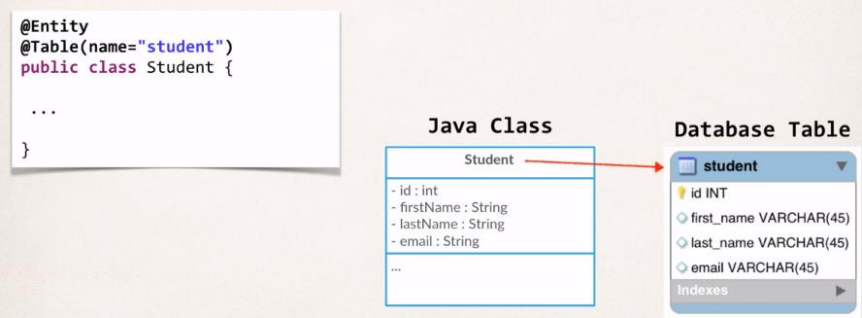
Two options for mapping a class:

1. XML config file (legacy); used when hibernate first came.
2. Java Annotations (modern and preferred)

How do we map Java Class?

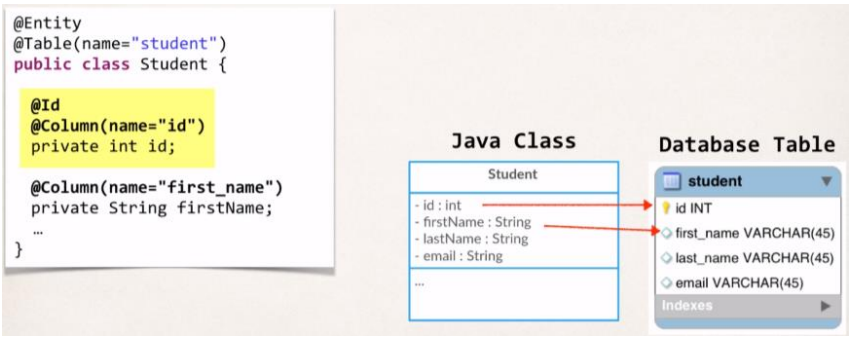
1. Map the Class to Database Table.

   @**Entity**: basically, tells the hibernate that this is the class which we are going to map to our Database Table.
   @Table(name="student"): Here we give the Actual name of the Database table we are this class to.
   This annotation is optional, because in our class java class name has the same name as our Database table does.



2. Map fields to Database Columns.

   @**Id**: Signifies that it is the primary key for this Class and @**Column:** gives the actual name of the column.
   So, here in this example field *`firstName`* is mapped to the column name *`first_name`*.
   There is no requirement for one-to-one naming convention it could be anything. And incase the field name and column name is exactly same then there is no need of @Column annotation, hibernate will automatically pick it up.



   And whatever annotations we have discussed so far must be from the `javax.persistence` package, as it's the standard interface which hibernate implements.

3. Develop Java code to perform Database Operation.

FAQ: Why we are using JPA Annotation instead of Hibernate?

FAQ: Why we are using JPA Annotation instead of Hibernate?

QUESTION:
Why we are using JPA Annotation instead of Hibernate?
For example, why we are not using this `org.hibernate.annotations.Entity`?

ANSWER:

JPA is a standard specification. Hibernate is an implementation of the JPA specification.

Hibernate implements all of the JPA annotations.

The Hibernate team recommends the use of JPA annotations as a best practice.

FAQ: Can Hibernate generate database tables based on the Java code?

Answer:

Yes, you can generate database tables from Java code. Here's a tutorial that shows you how.

https://www.dineshonjava.com/hibernate/hbm2ddl-configuration-and-name/

Section 21: Hibernate CRUD Features: Create, Read, Update and Delete
17 / 17 | 1hr 24min

- **Creating and Saving Java Objects with Hibernate:**

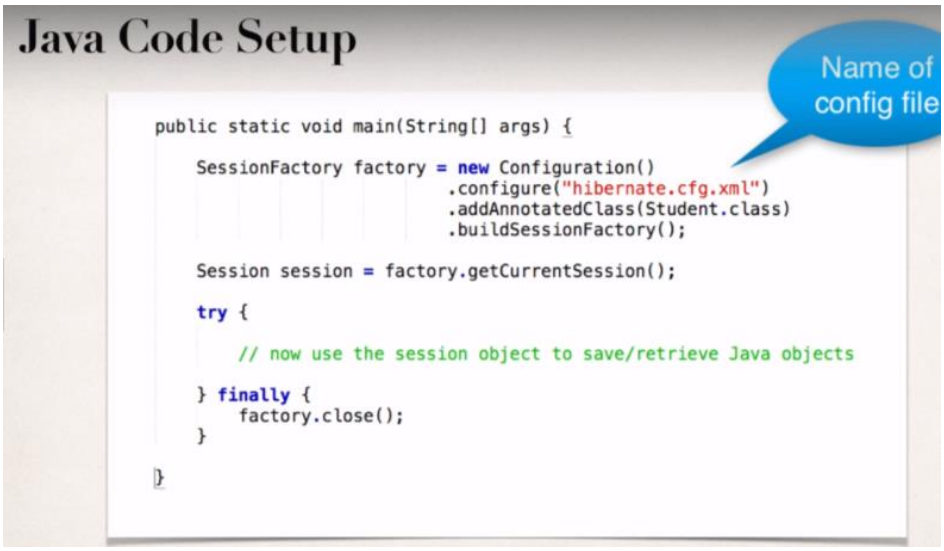
SessionFactory: It Read the config file and get the connection to the database. We only create it once in the app and we use again n again. And that SessionFactory will create sessions.

Session: This session object is mainly used for retrieving and saving the objects.

```
//create session factory
SessionFactory factory = new Configuration()
                         .configure("hibernate.cfg.xml")
                        .addAnnotatedClass(Student.class)
                        .buildSessionFactory();
```

Here we are getting the Reference to the sessionFactory, providing the hibernate configuration file name, and also giving the reference of Java class that is having special annotations on it, and buildSessionFactory() is giving us the object of SessionFactory that we can use later.

Now We have the SessionFactory object so, we can use it to get the object of a session, and then we can perform the normal database operation.



When we create the sessionFactory we are giving the config file name, that's actually not required, we can simple given .configiure() and by default hibernate will look for default file name hibernate.cfg.xml on our classpath.

Student.class

```
@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;
```

```
	public Student() {

	}

	public Student(String firstName, String lastName, String email) {
		this.firstName = firstName;
		this.lastName = lastName;
		this.email = email;
	}

	. . . . //getters and setters

	}
```

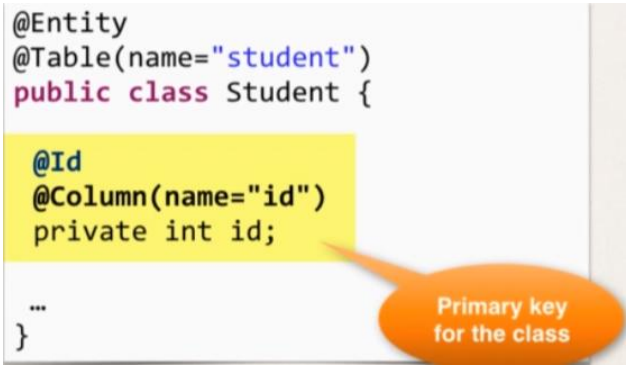
CreateStudentDemo.java

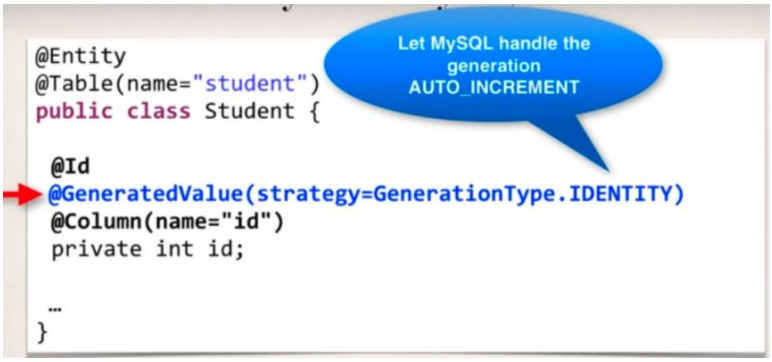
```
public class CreateStudentDemo {

	public static void main(String[] args) {

	//create session factory
	SessionFactory factory = new Configuration()
							.configure("hibernate.cfg.xml")
							.addAnnotatedClass(Student.class)
							.buildSessionFactory();

	//create session
	Session session = factory.getCurrentSession();

	try {

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

		//start a transaction
		session.beginTransaction();

		//save the Student Object
		System.out.println("Saving the student...");
		session.save(tempStudent);

		//commit transaction
		session.getTransaction().commit();

		System.out.println("Done!");

	}finally {
		session.close();
		factory.close();
	}

}

}
```

And the data gets saved in the DB.

@Id tells hibernate that the given field is Primary Key so this field gets mapped to the database table column “id”. And this way we leave up to hibernate to generate primary key for us.

And we can also tell hibernate explicitly how to perform generation and tell hibernate to use the given strategy for generating that id.





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

We can Also create our own custom Generation Strategy

- Create implementation of ***org.hibernate.id.IdentifierGenerator***
- Override the method: ***public Serialization generate(…)***

In the generate(…) method we can add our own custom business logic to generate the next Id and we return it as value and then hibernate will use this custom generator.

Warning: We need to look after these conditions before implementing the above method.

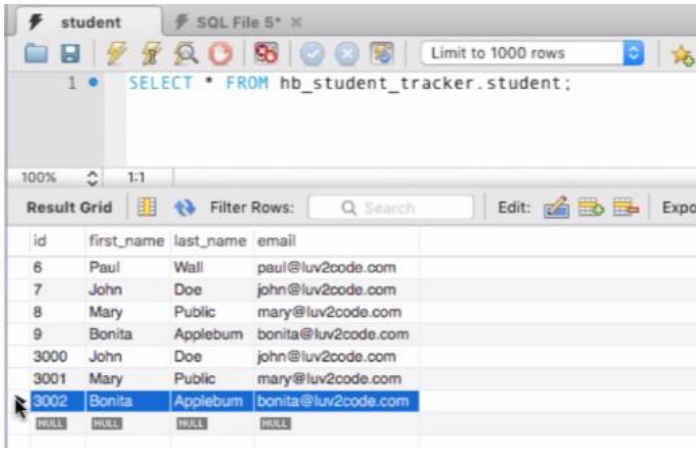
FAQ: How do I Change the AUTO_INCREMENT Values?

Answer: In to the new MySQL workbench tab.

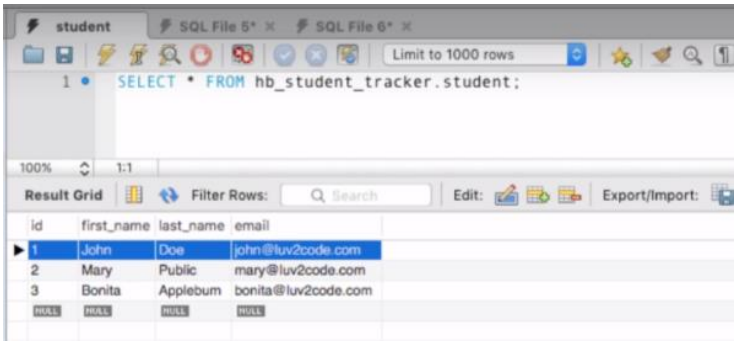
We will do alter table and will set the Auto_increment to 3000 and we can give the values here which are greater than our current value of id.

`ALTER TABLE hb_student_tracker.student AUTO_INCREMENT=3000;`

After executing whatever the next entries we are going to save it's id will start from 3000.

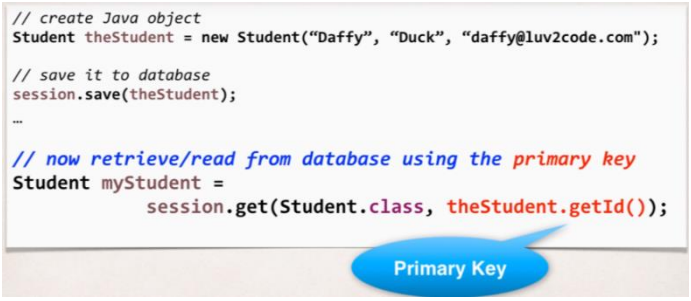


And to start with id = 1, truncate the table and start with new entries. `truncate hb_student_tracker.student;` and execute it. So, it will make our table empty and then add new entries and their id will start from 1.



- **Retrieving Object with Hibernate**

  In the hibernate we are retrieving the object using its Primary Key, and if the object is not found then it will simply return null.



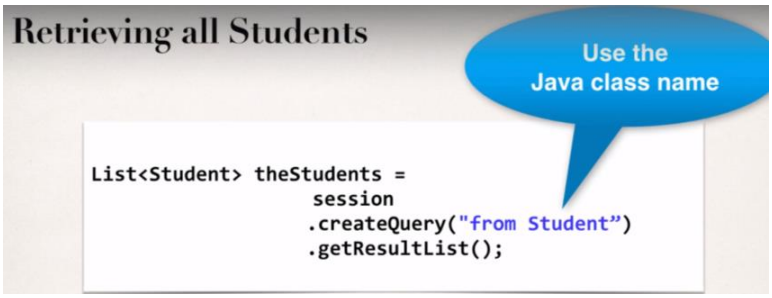
ReadStudentDemo.java

```
public class ReadStudentDemo {

	public static void main(String[] args) {

		//create session factory
		SessionFactory factory = new Configuration()
								.configure("hibernate.cfg.xml")
								.addAnnotatedClass(Student.class)
								.buildSessionFactory();

		//create session
		Session session = factory.getCurrentSession();

		try {

			//create the student object
			System.out.println("Create a Student Object");
			Student tempStudent = new Student("Daffy","Duck","daffy@luv2code.com");

			//start a transaction
			session.beginTransaction();

			//save the Student Object
			System.out.println("Saving the student...");
			System.out.println(tempStudent);
			session.save(tempStudent);

			//commit transaction
			session.getTransaction().commit();
			session.close();

			//My New Code
			System.out.println("Saved Student. Generated id: "+tempStudent.getId());

			//now get a new session and start transaction
			session = factory.getCurrentSession();
			session.beginTransaction();

			//retrieve the student based on the id: Primary key
			System.out.println("\nGetting student with id: "+tempStudent.getId());
			Student myStudent = session.get(Student.class, tempStudent.getId());
			System.out.println("Get complete: "+myStudent);

			//commit the transaction
			session.getTransaction().commit();

			System.out.println("Done!");

		}finally {
			session.close()
			factory.close();
		}

	}

}
```

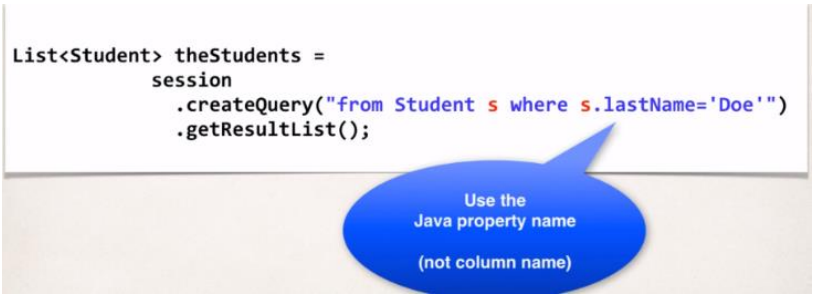
➢ Querying Objects with Hibernate

Hibernate has support query language for retrieving the objects and it's called as **Hibernate Query language (HQL)** and using this we get to query `where, like, order by, join, in,` etc.

Ex. The below query will print all the students inside the table.

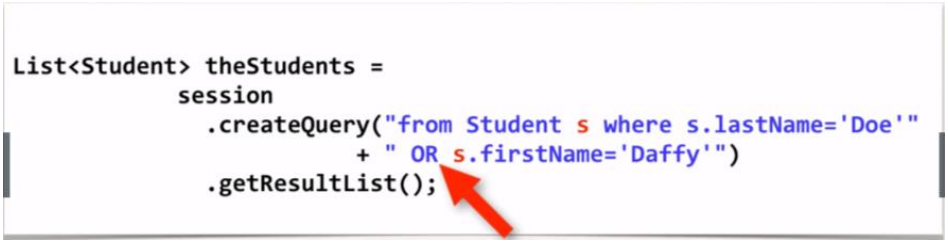


Ex. We are retrieving the object using the where clause of SQL.

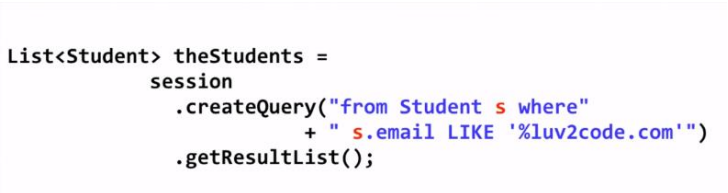


Note: So, here we refer everything using the object notation and the object properties.

Ex. We can also make use of predicate like **"OR"**, like we want to search for student whose First Name is "Daffy" or Last name is "Doe".



Ex. Also we can query using **"Like"** predicate.



We will get to see all these query and result in the java class: *QueryStudentDemo.java*

- **Update Objects with Hibernate:**

Let the session has begin and we need to perform the following steps:

Using the primary key, we will retrieve the Object from the DB, and then we can call setter method for updating the property and till here Object will remain in memory but once we commit the transaction then hibernate will apply the update to the DB.

Therefore, there is no hard requirement for session.save() or session.update() method because student object is a persistent object that we have retrieved from the DB and we can call the appropriate setters and finally do a commit and that will actually update the DB.

```
int studentId = 1;

Student myStudent = session.get(Student.class, studentId);

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

// commit the transaction
session.getTransaction().commit();
```

In the above case we are updating just for one Student Object and in case we need to perform update for all the student Objects stored in the DB then we need to create the Query using `session.createQuery()` method and then *`executeUpdate()`* has to be called.

```
session
    .createQuery("update Student set email='foo@gmail.com'")
    .executeUpdate();
```

And we can easily add where clause in case we want to update after applying certain filters.

Code regarding the update is available in Java class *`UpdateStudentDemo.java`*.

- **Delete Objects with Hibernate**

Again we need to keep our session started, get the primary key id and then we call *`session.delete(obj)`* inorder to delete the object from the DB.

```
int studentId = 1;

Student myStudent = session.get(Student.class, studentId);

// delete the student
session.delete(myStudent);

// commit the transaction
session.getTransaction().commit();
```

Another way to delete the Object:

```
session
    .createQuery("delete from Student where id=2")
    .executeUpdate();
```

In the previous example we need to retrieve the object first and then delete it but here we can simply delete it without retrieving the object directly from the DB.

Code regarding deletion is available in the class: `DeleteStudentDemo.java`

Java Project Link: **hibernate-tutorial**

FAQ: How to read Dates with Hibernate

FAQ: Handling Dates with Hibernate
How can I read date strings from the command-line and store them as dates in the database?

Answer:
You can make use of a combination of Java's date formatting class and Hibernate annotations.

Sample output:

Student [id=50, firstName=Paul, lastName=Doe, email=paul@luv2code.com, dateOfBirth=null]
Student [id=51, firstName=Daffy, lastName=Duck, email=daffy@luv2code.com, dateOfBirth=null]
Student [id=52, firstName=Paul, lastName=Doe, email=paul@luv.com, dateOfBirth=31/12/1998]

Development Process Overview
1. Alter database table for student
2. Add a date utils class for parsing and formatting dates
3. Add date field to Student class
4. Add toString method to Student class
5. Update CreateStudentDemo

Detailed steps

1. *Alter database table for student*

We need to alter the database table to add a new column for "date_of_birth".

Run the following SQL in your MySQL Workbench tool.

```
1. ALTER TABLE `hb_student_tracker`.`student`
2. ADD COLUMN `date_of_birth` DATETIME NULL AFTER `last_name`;
```

--

2. Add a date utils class for parsing and formatting dates

We need to add a DateUtils class to handle parsing and formatting dates. The source code is here. The class should be placed in the package: com.luv2code.hibernate.demo.

The date formatter uses special symbols for formatting/parsing.

- dd: day in month (number)
- MM: month in year (number)
- yyyy: year

See this link for details: https://docs.oracle.com/javase/tutorial/i18n/format/simpleDateFormat.html

```
1.  package com.luv2code.hibernate.demo;
2.
3.  import java.text.ParseException;
4.  import java.text.SimpleDateFormat;
5.  import java.util.Date;
6.
7.  public class DateUtils {
8.
9.      // The date formatter
10.     // - dd:   day in month (number)
11.     // - MM:   month in year (number)
12.     // - yyyy: year
13.     //
14.     // See this link for details: https://docs.oracle.com/javase/tutorial/i18n/format/simpleD
    ateFormat.html
15.     //
16.     //
17.     private static SimpleDateFormat formatter = new SimpleDateFormat("dd/MM/yyyy");
18.
19.     // read a date string and parse/convert to a date
20.     public static Date parseDate(String dateStr) throws ParseException {
21.         Date theDate = formatter.parse(dateStr);
22.
23.         return theDate;
24.     }
25.
26.     // read a date and format/convert to a string
27.     public static String formatDate(Date theDate) {
28.
29.         String result = null;
30.
31.         if (theDate != null) {
32.             result = formatter.format(theDate);
33.         }
34.
35.         return result;
36.     }
37. }
```

---

3. Add date field to Student class

We need to add a date field to the Student class. We map this field to the database column, "date_of_birth". Also, we make use of the @Temporal annotation. This is a Java annotation for storing dates.

```
1.      @Column(name="date_of_birth")
2.      @Temporal(TemporalType.DATE)
```

```
3.      private Date dateOfBirth;
```

Here's the full source code.

```
package com.luv2code.hibernate.demo.entity;
import java.util.Date;
import javax.persistence.Column;
import javax.persistence.Entity;
import javax.persistence.Id;
import javax.persistence.Table;
import javax.persistence.Temporal;
import javax.persistence.TemporalType;
import com.luv2code.hibernate.demo.DateUtils;
@Entity
@Table(name="student")
public class Student {

    @Id
    @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;

    @Column(name="date_of_birth")
    @Temporal(TemporalType.DATE)
    private Date dateOfBirth;

    public Student() {

    }
    public Student( String firstName, String lastName, String email, Date theDateOfBirth) {

        this.firstName = firstName;
        this.lastName = lastName;
        this.email = email;
        this.dateOfBirth = theDateOfBirth;

    }
    public int getId() {
        return id;
    }
    public void setId(int id) {
        this.id = id;
    }
    public String getFirstName() {
        return firstName;
    }
    public void setFirstName(String firstName) {
        this.firstName = firstName;
    }
    public String getLastName() {
```

```
        return lastName;
    }
    public void setLastName(String lastName) {
        this.lastName = lastName;
    }
    public String getEmail() {
        return email;
    }
    public void setEmail(String email) {
        this.email = email;
    }
    public Date getDateOfBirth() {
        return dateOfBirth;
    }
    public void setDateOfBirth(Date dateOfBirth) {
        this.dateOfBirth = dateOfBirth;
    }
    @Override
    public String toString() {
        return "Student [id=" + id + ", firstName=" + firstName + ", lastName=" + lastName + ",
email=" + email
                + ", dateOfBirth=" + DateUtils.formatDate(dateOfBirth) + "]";
    }

}
```

---

4. Add toString method to Student class

We will make an update to the toString method in our Student class. It will use the formatter from our DateUtils.class. This code is already included in Student.java from the previous step. I'm just highlighting it here for clarity.

```
1.          return "Student [id=" + id + ", firstName=" + firstName + ", lastName=" + lastName +
    ", email=" + email
2.                  + ", dateOfBirth=" + DateUtils.formatDate(dateOfBirth) + "]";
```

Note the use of DateUtils above.

---

5. Update CreateStudentDemo

Now for the grand finale. In the main program, read the date as a String and parse/convert it to a date. Here's the snippet of code.

```
1.              String theDateOfBirthStr = "31/12/1998";
2.              Date theDateOfBirth = DateUtils.parseDate(theDateOfBirthStr);
3.
4.              Student tempStudent = new Student("Pauly", "Doe", "paul@luv.com", theDateOfBirth
    );
```

Here's the full code:

```
1.  package com.luv2code.hibernate.demo;
2.
3.  import java.text.ParseException;
```

```
4.  import java.util.Date;
5.  import org.hibernate.Session;
6.  import org.hibernate.SessionFactory;
7.  import org.hibernate.cfg.Configuration;
8.  import com.luv2code.hibernate.demo.entity.Student;
9.
10. public class CreateStudentDemo {
11.
12.     public static void main(String[] args) {
13.
14.         // create session factory
15.         SessionFactory factory = new Configuration().configure("hibernate.cfg.xml").addAnno
    tatedClass(Student.class)
16.                 .buildSessionFactory();
17.
18.         // create a session
19.         Session session = factory.getCurrentSession();
20.
21.         try {
22.             // create a student object
23.             System.out.println("creating a new student object ...");
24.
25.             String theDateOfBirthStr = "31/12/1998";
26.
27.             Date theDateOfBirth = DateUtils.parseDate(theDateOfBirthStr);
28.
29.             Student tempStudent = new Student("Pauly", "Doe", "paul@luv.com", theDateOfBirt
    h);
30.
31.             // start transaction
32.             session.beginTransaction();
33.
34.             // save the student object
35.             System.out.println("Saving the student ...");
36.             session.save(tempStudent);
37.
38.             // commit transaction
39.             session.getTransaction().commit();
40.
41.             System.out.println("Success!");
42.         } catch (Exception exc) {
43.             exc.printStackTrace();
44.         } finally {
45.             factory.close();
46.         }
47.     }
48.
49. }
```

Section 22: Hibernate Advanced Mappings

2 / 2 | 8min

Advanced Mapping Overview

In the database, you most likely will have

- Multiple tables
- Relationships between tables

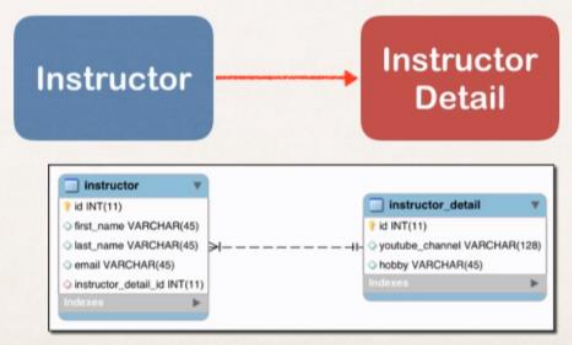
So, we need to model this with Hibernate.

Types of mapping:

- One-to-one
- One-to-Many, Many-to-One
- Many-to-Many

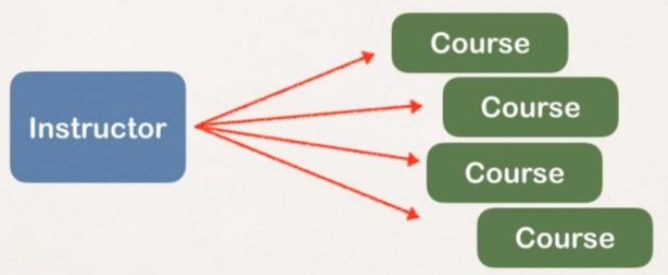
One-to-one Mapping

Ex: An Instructor can have an “instructor detail” entity. Similar to an “instructor profile”



One-To-Many Mapping

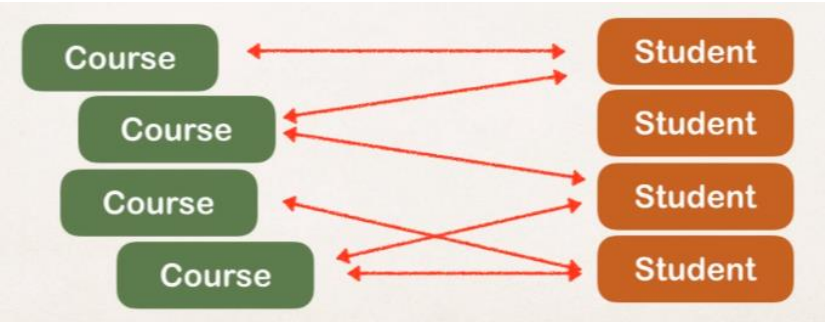
Ex: An instructor can have many courses.



Many-to-Many Mapping

Ex:

- A course can have many students.
- A student can have many courses.



Unidirectional Relationship:

Ex: We have Instructor and Instructor Detail so we start with loading the instructor and from there we can access the instructor detail. So, this is a one-way relationship and that's called Unidirectional relationship.

Bidirectional Relationship: In this case we can also load the instructor ausing the instructor Details object.

Section 23: Hibernate Advanced Mappings - @OneToOne
17 / 17 | 1hr 27min

One-to-one Mapping Overview: Java Project Link

- Development Process:

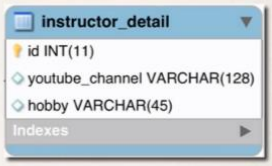
1. Prep Work - Define database tables
2. Create InstructorDetail class
3. Create Instructor class
4. Create Main App

✓ **Prep Work _ define Database Tables:**

- Instructor_detail table:

File: create-db.sql

```
CREATE TABLE `instructor_detail` (

  `id`              int(11)      NOT NULL AUTO_INCREMENT,
  `youtube_channel` varchar(128) DEFAULT NULL,
  `hobby`           varchar(45)  DEFAULT NULL,

  PRIMARY KEY (`id`)
);

...
```

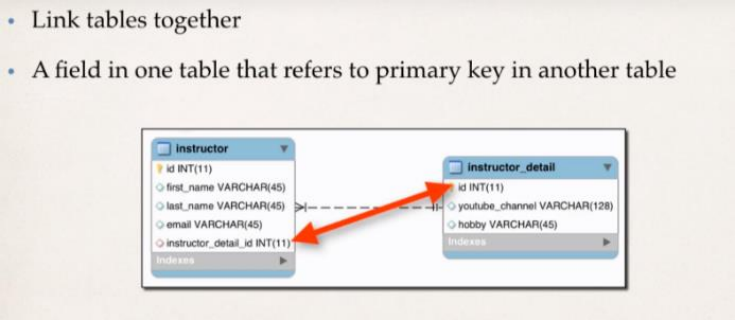


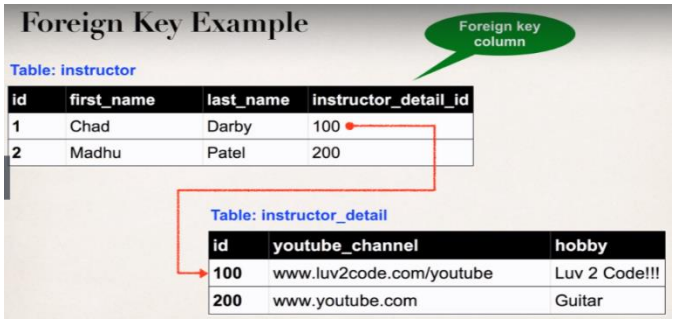
- Instructor Table:

File: create-db.sql

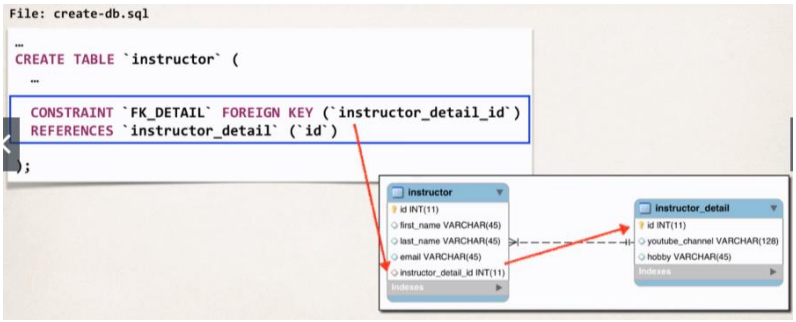
```
…
CREATE TABLE `instructor` (

  `id`         int(11) NOT NULL AUTO_INCREMENT,
  `first_name` varchar(45) DEFAULT NULL,
  `last_name`  varchar(45) DEFAULT NULL,
  `email`      varchar(45) DEFAULT NULL,
  `instructor_detail_id` int(11) DEFAULT NULL,

  PRIMARY KEY (`id`)
  ...
);
```

So, far we have two separate tables and we need to link the tables (instructor_detail and instructor tables) together.
We will use Foreign Key to link the tables together.

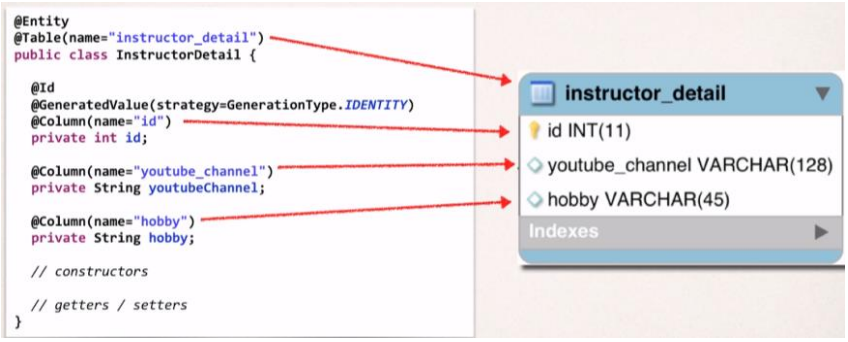




- Defining Foreign Key:

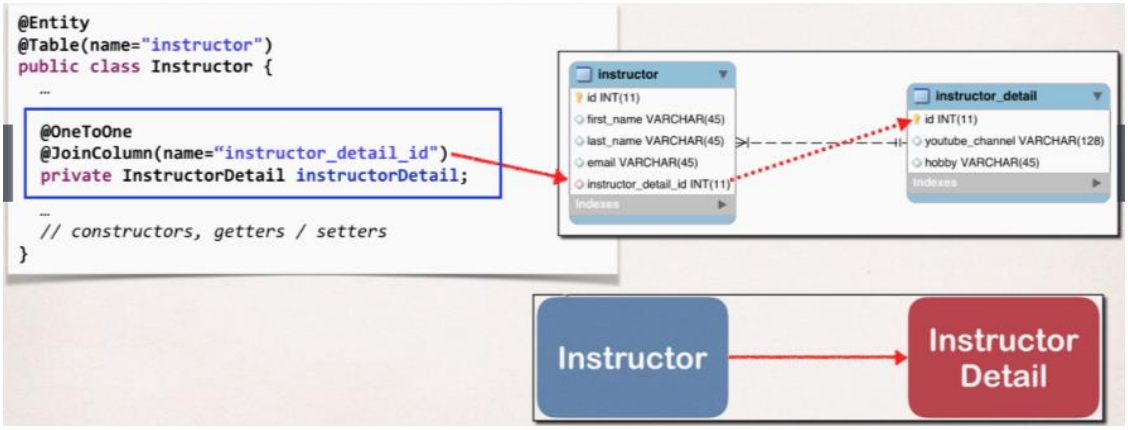


✓ **Create InstructorDetail Class:** InstructionDetail.java



✓ **Create Instructor Class**: Instruction.java

Basically, we have two independent classes but they are not linked so we need to add this @OneToOne annotation to map the two classes together.



After adding the @OneToOne Annotation we need to tell the hibernate how to hook the two tables; *@JoinColumn(name="instructor_detail_id")* will say hey hibernate for the *Instructor_detail* we have a join column called *instructor_detail_id* which is defined in the *instructor* table.

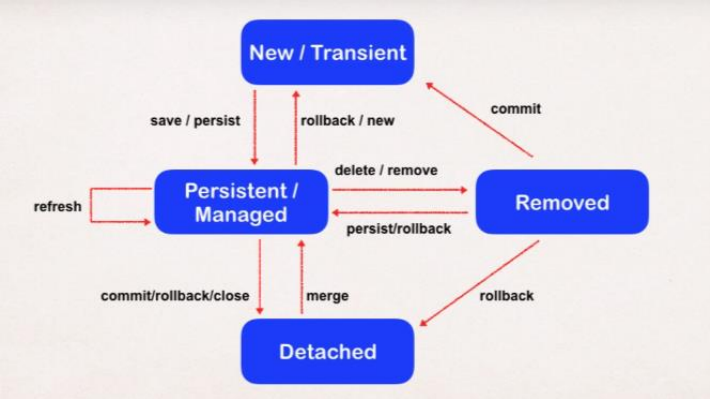
Then use that information hibernate will go off use the foreign key and find the *instructor_detail* record and it will load the data accordingly. Hibernate will do all the work in the background and then in memory we will have instructor object along with its related entity *instructor_detail.*

- **Entity LifeCycle:**

  It's a set of states that a hibernate entity class can go through.

Operations	Description
Detach	If entity is detached, it is not associated with a Hibernate session
Merge	If instance is detached from session, then merge will reattach to session
Persist	Transitions new instances to managed state. Next flush / commit will save in db.
Remove	Transitions managed entity to be removed. Next flush / commit will delete from db.
Refresh	Reload / synch object with data from db. Prevents stale data

Refresh allows us to reload or sync the object with data from the DB, this actually helps you to prevent from having stale data in memory which is different from the data in DB.



❖ **Cascade Types:**

- **PERSIST:** If entity is persisted /saved then the related entity will also be persisted. In our case if Instructor object is saved then the InstructorDetail object will also be saved.
- **REMOVE**: If entity is removed or deleted then the related entity will also be removed.
- **REFRESH**: if entity is refreshed i.e., synced from the DB then the related entity will also be refreshed or synced from DB.

Cascade Type	Description
PERSIST	If entity is persisted / saved, related entity will also be persisted
REMOVE	If entity is removed / deleted, related entity will also be deleted
REFRESH	If entity is refreshed, related entity will also be refreshed
DETACH	If entity is detached (not associated w/ session), then related entity will also be detached
MERGE	If entity is merged, then related entity will also be merged
ALL	All of above cascade types

❖ **How to configure cascade type?**

```
@Entity
@Table(name="instructor")
public class Instructor {
  …

  @OneToOne(cascade=CascadeType.ALL)
  @JoinColumn(name="instructor_detail_id")
  private InstructorDetail instructorDetail;

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

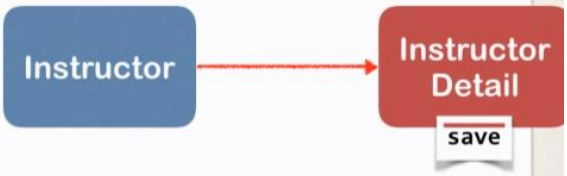
Note: By default, no operations are cascaded if we don't specify cascade then none of the operations are cascaded and we have to explicitly reference a cascade type that we want to apply for our given relationship.

❖ **Configure Multiple Cascade Types**

```
@OneToOne(cascade={CascadeType.DETACH,
                   CascadeType.MERGE,
                   CascadeType.PERSIST,
                   CascadeType.REFRESH,
                   CascadeType.REMOVE})
```

✓ **Create Main App:** CreateDemo.java

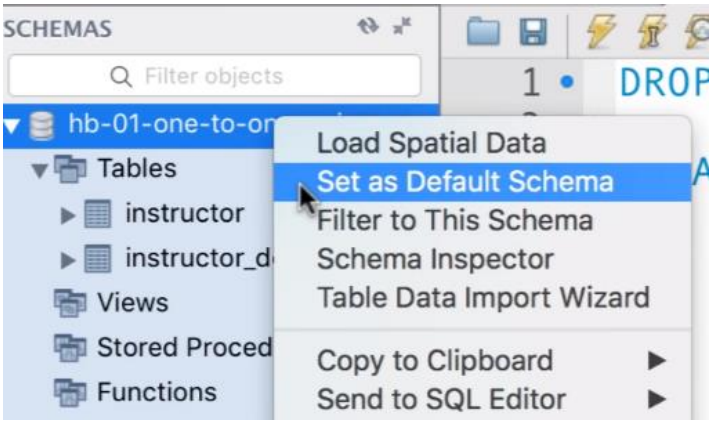
```
public static void main(String[] args) {
    …
    // create the objects
    Instructor tempInstructor = new Instructor("Chad", "Darby", "darby@luv2code.com");

    InstructorDetail tempInstructorDetail =
        new InstructorDetail("http://www.luv2code.com/youtube", "Luv 2 code!!!");

    // associate the objects
    tempInstructor.setInstructorDetail(tempInstructorDetail);

    // start a transaction
    session.beginTransaction();

    session.save(tempInstructor);

    // commit transaction
    session.getTransaction().commit();
    …
}
```



So, when we save the instructor and as we have defined our cascade reference, on saving the instructor on doing the commit transaction, it will also save the constructor details. Therefore, on saving one object, it will end up saving multiple objects.

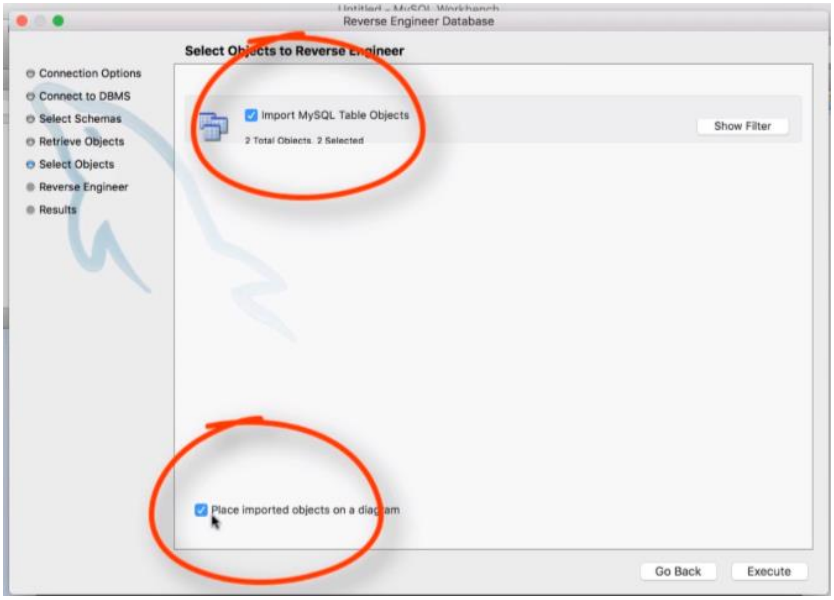
We can get the `Hibernate Database script` required for this demo from the link.

And then run the Script in the MySQL workbench, and the script will create the `instructor` table and `instructor_detail` table. And then set the schema as Default.



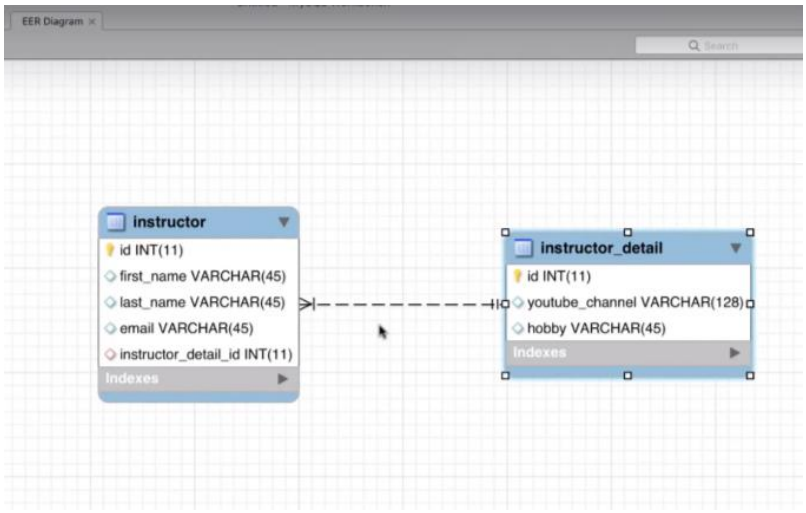
❖ To get the ER Diagram of the current Schema:

Click on Database > Reverse Engineer and then perform steps as directed in the next screens and basically it will read the database tables and create the ER diagram.



Make sure these two check Boxes are checked! And then Click on execute.

And here we get the ER diagram.



And Update the `hibernate.cfg.xml` file: with new schema name

```
<session-factory>

    <!-- JDBC Database connection settings -->
    <property name="connection.driver_class">com.mysql.cj.jdbc.Driver</property>
    <property name="connection.url">jdbc:mysql://localhost:3306/hb-01-one-to-one-uni?u
    <property name="connection.username">hbstudent</property>
    <property name="connection.password">hbstudent</property>
```

- **Delete an Instructor** DeleteDemo.java

  So, we have cascading in place, when we delete an Instructor it will also cascade down and delete the appropriate instruction detail.

Section 23: Hibernate Advanced Mappings - @OneToOne

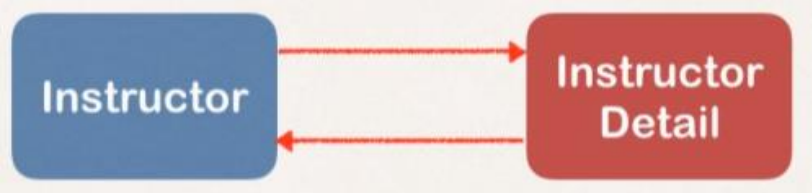
17 / 17 | 1hr 27min

- **Hibernate Bi-directional** java project link

New Use Case:

- If we load an InstructorDetail
- Then we'd like to get the associated Instructor with our InstructorDetail object.
- But we can't do this with current uni-directional relationship, as right now we can only start with Instructor and move to InstructorDetail.

Bidirectional relationship is the solution.



No change required to Database; we simply need to update Java code.

Development Process:

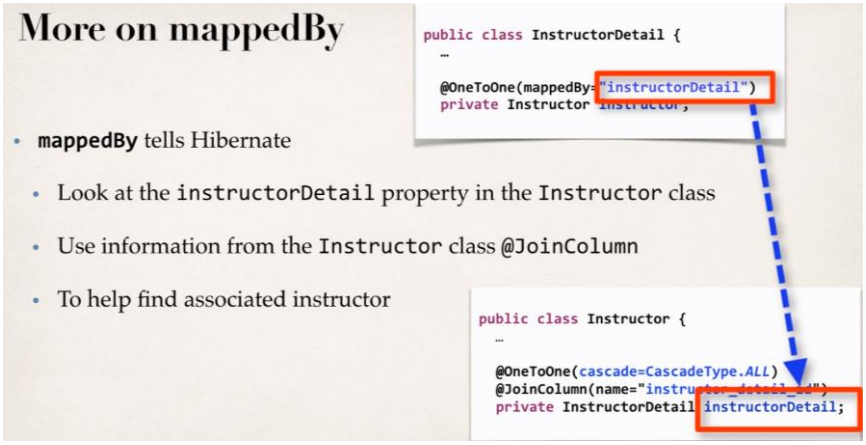
1. Make updates to `InstructorDetail` class:
   1. Add new field to reference Instructor.
   2. Add getter and setter methods for Instructor.

```
@Entity
@Table(name="instructor_detail")
public class InstructorDetail {
  …

  private Instructor instructor;

  public Instructor getInstructor() {
    return instructor;
  }

  public void setInstructor(Instructor instructor) {
    this.instructor = instructor;
  }

}
```

3. Add @oneToOne annotation

```
public class InstructorDetail {
  …

  @OneToOne(mappedBy="instructorDetail")
  private Instructor instructor;
```

What do *mappedBy* property mean?
This basically refers to the “instructorDetail” property in the Instructor Class. So, we are telling hibernate that this instructor field is mapped by the instructor detail property in the instructor class; and hibernate will figure out: look for the foreign key relationship and match everything up. So, hibernate use this information to know how these items are linked with each other an also find the appropriate instructor for this instructorDetail.

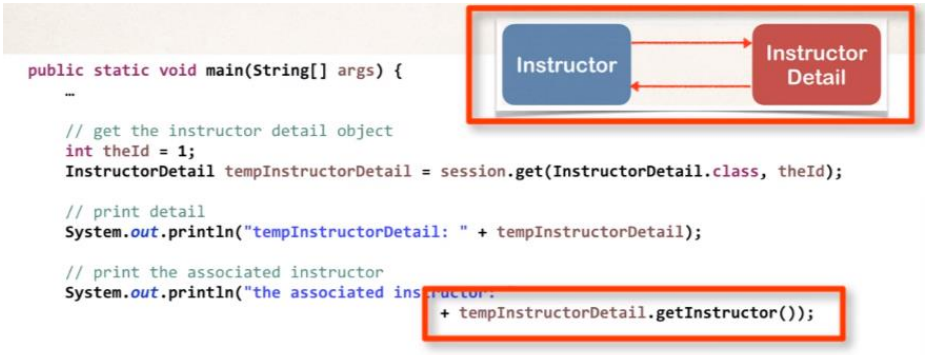


We will also add “CascadeType.ALL”. So, it cascades all operations to the associated instructor, and if we are loading `instructorDetail` and I am `deleting` that `instructorDetail` then it will also cascade this delete operation to the associated instructor.

```
// add @OneToOne annotation

@OneToOne(mappedBy="instructorDetail", cascade=CascadeType.ALL)
private Instructor instructor;
```

```
@Entity
@Table(name="instructor_detail")
public class InstructorDetail {
  …

  @OneToOne(mappedBy="instructorDetail")
  private Instructor instructor;

  public Instructor getInstructor() {
    return instructor;
  }

  public void setInstructor(Instructor instructor) {
    this.instructor = instructor;
  }
  …

}
```

2. Create the main App to test all this.



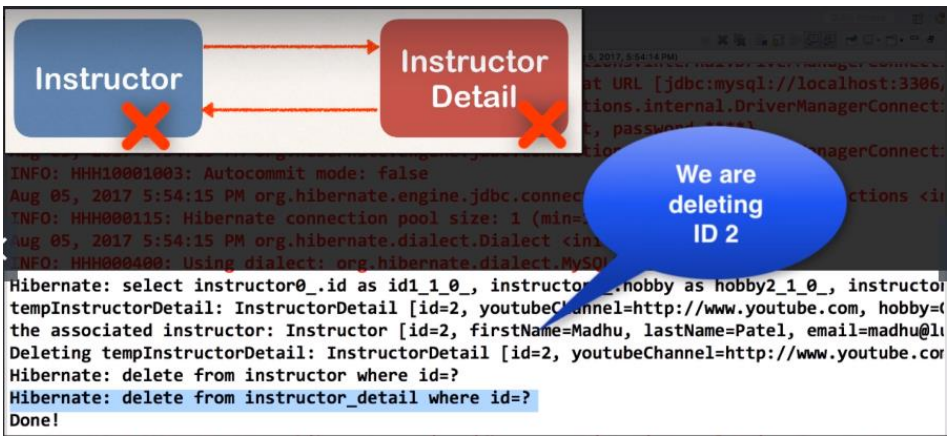
Cascade Delete:

Means deleting the InstructorDetail and the corresponding Instructor as well.

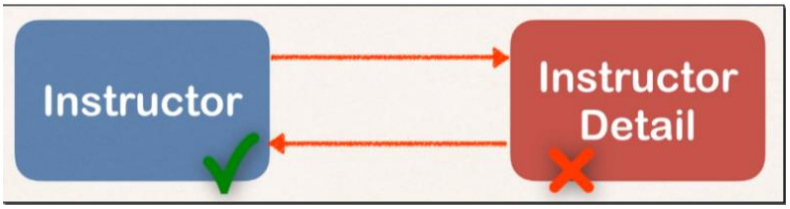
In the DeleteInstructorDetailDemo.java

```
//start a transaction
session.beginTransaction();

//get the instructor detail Object
int theId = 3;
InstructorDetail tempInstructorDetail =
            session.get(InstructorDetail.class, theId);

//print the instructor detail
System.out.println("temoInstructorDetail: "+tempInstructorDetail);

//also print the associated instructor
System.out.println("the associated Instructor:
"+tempInstructorDetail.getInstructor());

System.out.println("Deleteing tempInstructorDetail: "+tempInstructorDetail);

//now let's delete the instructor detail
session.delete(tempInstructorDetail);

//commit transaction
session.getTransaction().commit();
```

So, in the output we can see it's deleting both *InstructorDetail* as well as Instructor, due to cascading.



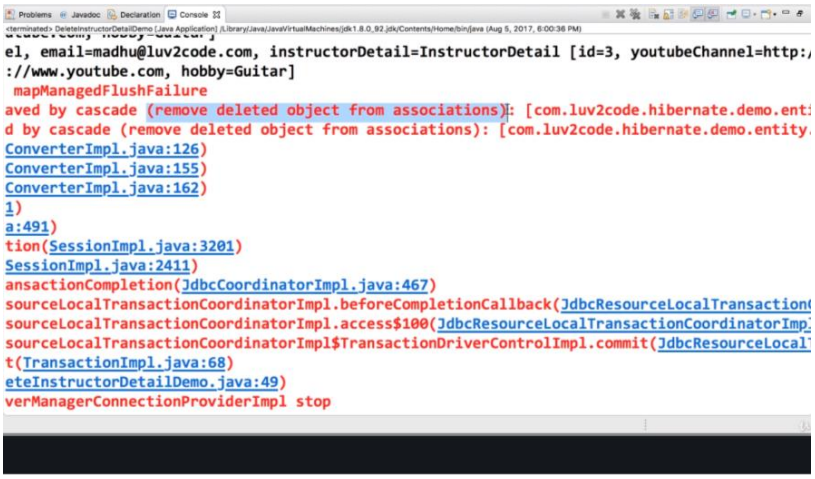
- In case, ***we only want to delete*** the *InstructorDetail* and want to keep the Instructor in the DB.



In the ***InstructorDetail.java*** class we will remove cascade type = ALL and enter all the other cascade types, so that on deleting `InstructorDetail` it will only delete InstructorDetail not the associated ***Instructor*** object as well.

```
// add @OneToOne annotation

@OneToOne(mappedBy="instructorDetail",
        cascade= {CascadeType.DETACH, CascadeType.MERGE, CascadeType.PERSIST,
                  CascadeType.REFRESH})
private Instructor instructor;
```

In case we are using the same code for deleting the instructorDetail and didn't set the **null reference** in the Instructor object for InstructorDetail Object then it will lead to some error: "remove deleted object from association"



So, we need to modify the main App: need to break the bidirectional link, then it will lead to desired outcome.

```
//start a transaction
session.beginTransaction();

//get the instructor detail Object
int theId = 3;
InstructorDetail tempInstructorDetail =
            session.get(InstructorDetail.class, theId);

//print the instructor detail
System.out.println("temoInstructorDetail: "+tempInstructorDetail);

//also print the associated instructor
System.out.println("the associated Instructor:
"+tempInstructorDetail.getInstructor());

//now let's delete the instructor detail
// remove the associated object reference
//break bidirectional link
tempInstructorDetail.getInstructor().setInstructorDetails(null);
System.out.println("Deleteing tempInstructorDetail: "+tempInstructorDetail);
```

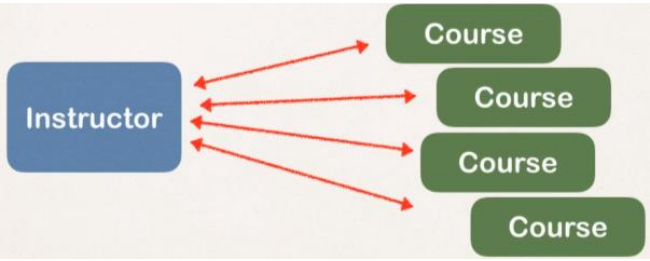
```java
//now let's delete the instructor detail
session.delete(tempInstructorDetail);

//commit transaction
session.getTransaction().commit();
System.out.println("Done!");
```

Section 24: Hibernate Advanced Mappings - @OneToMany

10 / 10 | 43min

- **Hibernate One-To-many Mapping Bidirectional** JAVA Project Link

Use Case: An Instructor can have many courses, and its bidirectional so we can go from instructor to course and from the course we can go to Instructor.



And Many-To-one Mapping: Many courses belong to one Instructor.

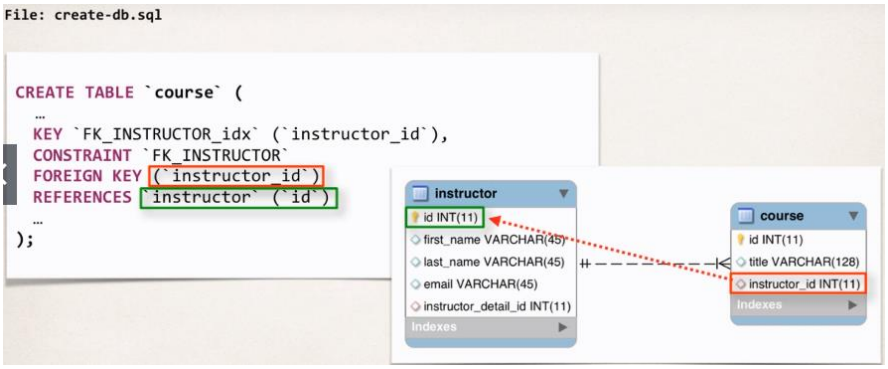
Real World Project Requirement:

- If you delete an Instructor, DONOT delete the courses.
- If you delete a course, DONOT delete the Instructor.

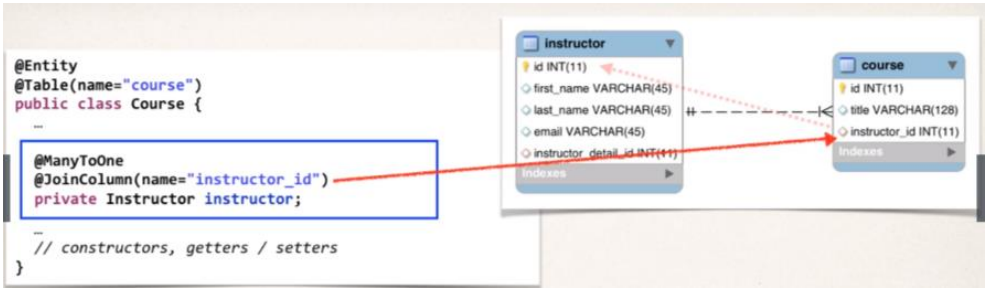
So, it means that; **do not apply cascading deletes!**

Development Process

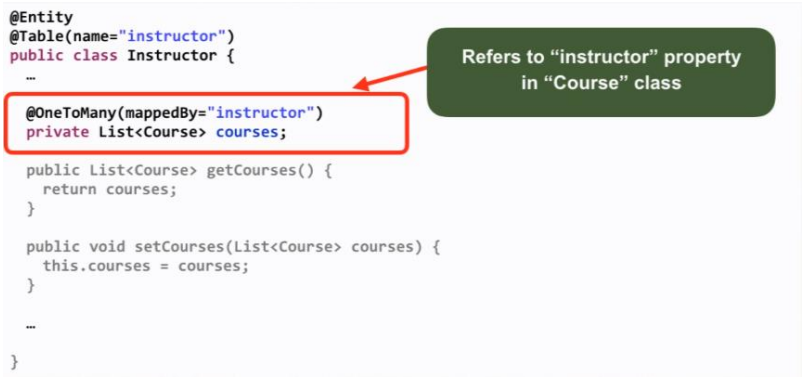
1. Define databases tables. SQL SCRIPTS



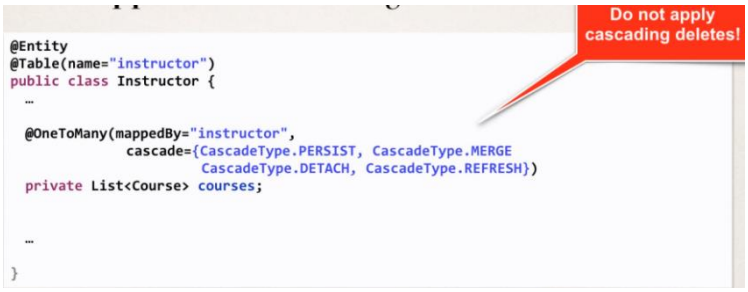
2. Create Course class. Course.java



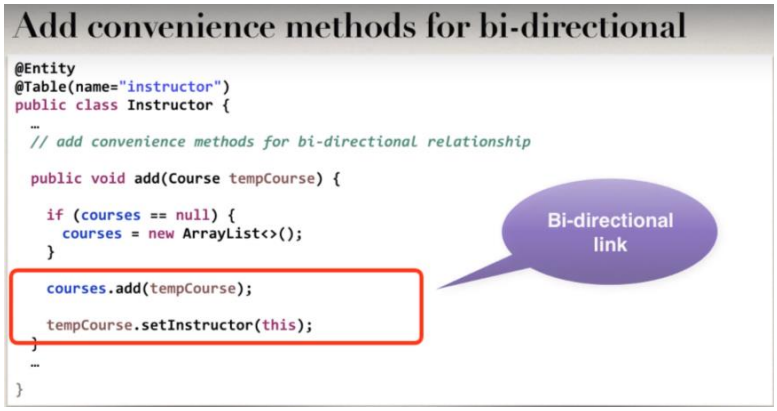
3. Update the Instructor Class. Instructor.java



Here we are handling our real-world project requirement and similarly we do in course class as we don't want to perform cascading deletes.



And finally, we are defining the bidirectional link between the Instructor and Courses.



4. Create the main Application: Make sure to update the DB schema to point to new Schema i.e., "hi-03-one-to-many" in the hibernate.config.xml.

   ✓ File: CreateInstructorDemo.java just adding some instructor for bootstrapping.

```
// create the objects
Instructor tempInstructor =
        new Instructor("Susan", "Public", "susan.public@luv2code.com");

InstructorDetail tempInstructorDetail =
        new InstructorDetail(
                "http://www.youtube.com",
                "Video Games");
```

   ✓ File: CreateCoursesDemo.java here we are creating courses and adding it to the instructor.

```
// get the instructor from db
int theId = 1;
Instructor tempInstructor = session.get(Instructor.class, theId);

// create some courses
Course tempCourse1 = new Course("Air Guitar - The Ultimate Guide");
Course tempCourse2 = new Course("The Pinball Masterclass");

// add courses to instructor
tempInstructor.add(tempCourse1);
tempInstructor.add(tempCourse2);

// save the courses
session.save(tempCourse1);
session.save(tempCourse2);
```

   ✓ File: GetInstructorCoursesDemo.java Here we are actually fetching the associated courses of an Instructor that we have saved in createCoursesDemo.java app.

```
// get the instructor from db
int theId = 1;
Instructor tempInstructor = session.get(Instructor.class, theId);

System.out.println("Instructor: " + tempInstructor);

// get courses for the instructor
System.out.println("Courses: " + tempInstructor.getCourses());
```

❖ **Delete a course:**

Requirement: If we delete a course then do not delete the instructor.

✓ File: DeleteCourseDemo.java

```
// get a course
int theId = 10;
Course tempCourse = session.get(Course.class, theId);

// delete course
System.out.println("Deleting course: " + tempCourse);

session.delete(tempCourse);
```

Section 25: Hibernate Advanced Mappings - Eager vs Lazy Loading

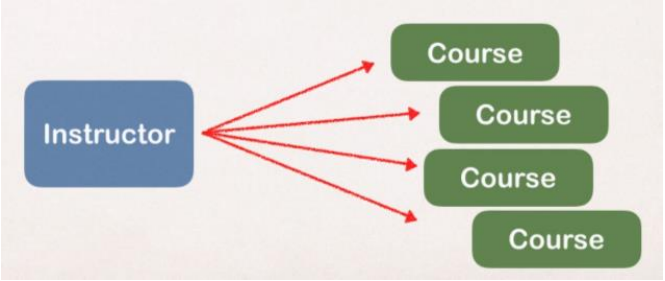
8 / 8 | 36min

Overview: Java Project Link

When we fetch / retrieve data, should we retrieve everything?

- **Eager** will retrieve everything.
- **Lazy** will retrieve on request.

➢ Eager Loading

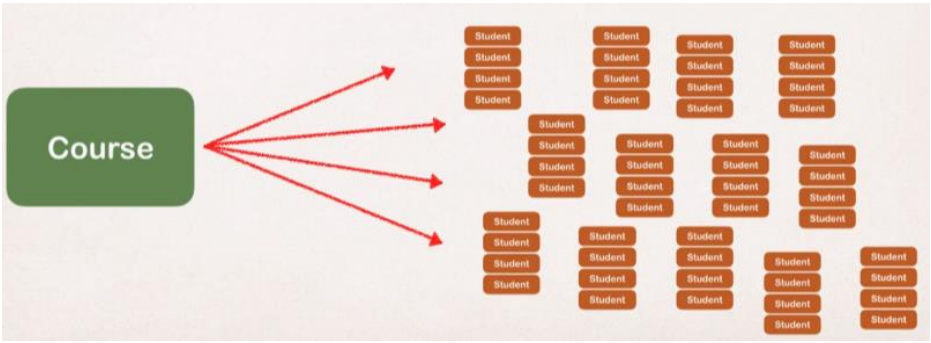


Eager loading will load all dependent entities.

Load Instructor and all of their courses at once.

Ex. Consider a course have some 20k students.

Then loading all those students would definitely turn the performance into nightmare.



- In case in our App, we want to search for course by keyword then,
- We only want a list of matching courses.
- But **EAGER** loading would still load all students of each course, which is not good.

❖ Best Practice:

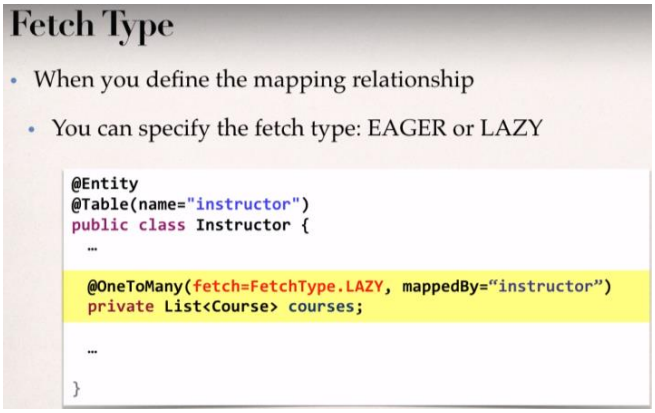
Only load data when absolutely needed. Prefer Lazy Loading instead of Eager Loading.

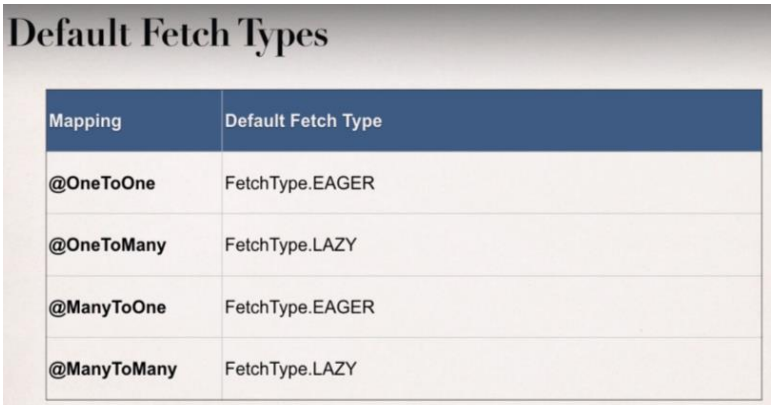
➢ Lazy Loading:

Lazy loading will load the main entity first and it will load the dependent entities on demand (Lazy).

As per our use case above: Lazy Loading will load the courses first and then when we need the list of students then we will actually go to the DB and load those students on demand.

Fetch Type:





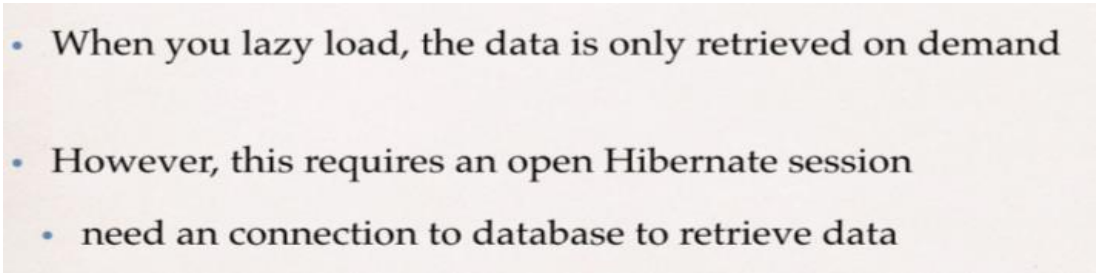
Overriding Default Fetch Type:

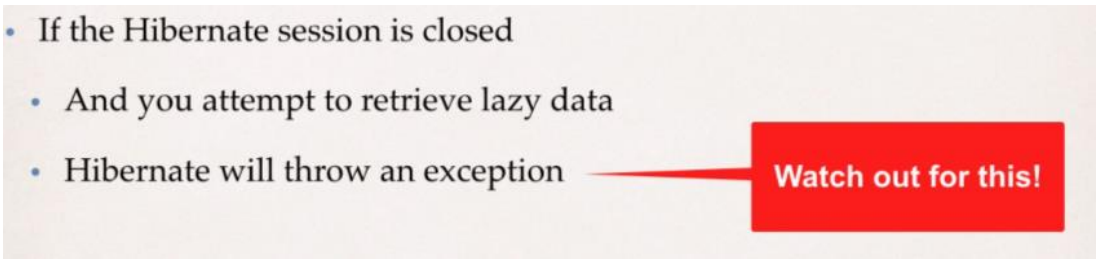
Specifying the fetch type, overrides the defaults.

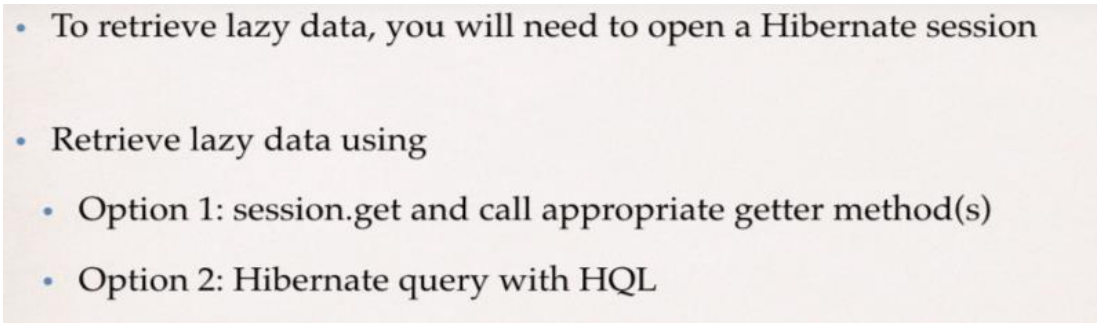
Now as we can see **@ManyToOne** default fetch type is **Eager** and we want to override it to **Lazy**.

```
@ManyToOne(fetch=FetchType.LAZY)
@JoinColumn(name="instructor_id")
private Instructor instructor;
```

- **Important Points:**







- **FetchType.EAGER**

File: Instructor.java

We are making the ***FetchType.EAGER***; and it will fetch all the course when it fetches the Instructor Object from the DB. (As @oneToMany has default fetch type equals to Lazy)

```
@OneToMany(fetch=FetchType.EAGER,
           mappedBy="instructor",
           cascade= {CascadeType.PERSIST, CascadeType.MERGE,
                     CascadeType.DETACH, CascadeType.REFRESH})
private List<Course> courses;
```

File: EagerLazyDemo.java

```
//get the instructor from the db
Instructor tempInstructor = session.get(Instructor.class, theId);

System.out.println("luv2code: Instructor: "+tempInstructor);

//get course for the instructor ---OPTION 1
System.out.println("luv2code: courses: "+tempInstructor.getCourses());
```

At the highlighted line, hibernate will fetch courses of Instructor as we have mentioned **FetchType**.**EAGER** for the Courses in Instructor class**,** and in the rest of the line hibernate doesn't need to fetch the associated instructor detail and course for that instructor form the DB.

- **FetchType.LAZY**

File: Instructor.java

```
@OneToMany(fetch=FetchType.LAZY,
           mappedBy="instructor",
           cascade= {CascadeType.PERSIST, CascadeType.MERGE,
                     CascadeType.DETACH, CascadeType.REFRESH})
private List<Course> courses;
```

Now hibernate will fetch courses associated to any particular instructor only when there is demand else it will simply load the instructor object only.

```
// get the instructor from db
int theId = 1;
Instructor tempInstructor = session.get(Instructor.class, theId);

System.out.println("luv2code: Instructor: " + tempInstructor);

// get courses for the instructor
System.out.println("luv2code: Courses: " + tempInstructor.getCourses());
```

When we are calling `tempInstructor.getCourses()` method then hibernate will fire up the DB query and fetch the courses and it is called fetchType **Lazy**.

➢ Close Session Before Accessing Lazy Data

File: EagerLazyDemo.java

```
   //commit transaction
1. session.getTransaction().commit();

   //close the session
2. session.close();

3. System.out.println("luv2code: session is closed now...\n\n\n\n");
   //since courses are lazy loaded ...this should fail

   //get course for the instructor
4. System.out.println("luv2code: courses: "+tempInstructor.getCourses());
```

At line 2, we are closing the session and then trying to load the courses and then hibernate will **throw Exception.**

Reason:

How to Resolve the Lazy Loading issue?

Option 1:

Call the getter method while the session is open.

File: EagerLazyDemo.java

```
        //start a transaction
1.      session.beginTransaction();
        int theId = 1;

        //get the instructor from the db
2.      Instructor tempInstructor = session.get(Instructor.class, theId);

        System.out.println("luv2code: Instructor: "+tempInstructor);

        //get course for the instructor ---OPTION 1
3.      System.out.println("luv2code: courses: "+tempInstructor.getCourses());


        //commit transaction
4.      session.getTransaction().commit();

        //close the session
5.      session.close();

        System.out.println("luv2code: session is closed now...\n\n\n\n");
        //since courses are lazy loaded ...this should fail

        //get course for the instructor
6.      System.out.println("luv2code: courses: "+tempInstructor.getCourses());
```

At line 3, we are calling the getter method while the session was open which means we are loading the courses associated when session was open, and later after closing the session we are again calling the getter method at line 6 and this time the code will work fine and it will not give any such exception as we have discussed above.

Option 2:

Query With HQL JOIN FETCH

File: FetchJoinDemo.java

```
    //OPTION 2: Hibernate Query with HQL
1.  int theId = 1;
    Query<Instructor> query =
              session.createQuery("select i from Instructor i "
                                                  +"JOIN FETCH i.courses "
                                              +"where i.id=:theInstructorId",
                                          Instructor.class);

    //set parameter on query
2.  query.setParameter("theInstructorId", theId);

    //execute query and get instructor
3.  Instructor tempInstructor = query.getSingleResult();

4.  System.out.println("luv2code: Instructor: "+tempInstructor);
```

```
        //commit transaction
5.      session.getTransaction().commit();

        //close the session
6.      session.close();

7.      System.out.println("luv2code: session is closed now...\n\n\n\n");
        //since courses are lazy loaded ...this should fail

        //get course for the instructor
8.      System.out.println("luv2code: courses: "+tempInstructor.getCourses());
```

- At Line 3, on executing the query it will load the instructor and courses all at once. And line 8, we are simply printing the courses associated.

FAQ: How to load the courses at a later time in the application?

FAQ: How load the courses at a later time in the application?
Question
I've watched your 2 solutions for loading related data after session closing. Both, either getting related courses before closing session and using JOIN FETCH seem to be negating of lazy loading (using those solutions we completely resign of lazy loading.
Is there any good solution to load these data somewhere else in the app? Should I open new session?

Answer
Yes, you can load it later with using a new session, just make use of HQL
Here's the code snippet.

```
1.              session = factory.getCurrentSession();
2.
3.              session.beginTransaction();
4.
5.              // get courses for a given instructor
6.              Query<Course> query = session.createQuery("select c from Course c "
7.                                                        + "where c.instructor.id=:theInstructo
    rId",
8.                                                        Course.class);
9.
10.             query.setParameter("theInstructorId", theId);
11.
12.             List<Course> tempCourses = query.getResultList();
13.
14.             System.out.println("tempCourses: " + tempCourses);
```

Here's the full example.

```
1.  package com.luv2code.hibernate.demo;
2.  import java.util.List;
3.  import org.hibernate.Session;
4.  import org.hibernate.SessionFactory;
5.  import org.hibernate.cfg.Configuration;
6.  import org.hibernate.query.Query;
7.  import com.luv2code.hibernate.demo.entity.Course;
8.  import com.luv2code.hibernate.demo.entity.Instructor;
9.  import com.luv2code.hibernate.demo.entity.InstructorDetail;
10. public class GetCoursesLater {
11.     public static void main(String[] args) {
12.         // create session factory
13.         SessionFactory factory = new Configuration()
14.                                 .configure("hibernate.cfg.xml")
15.                                 .addAnnotatedClass(Instructor.class)
16.                                 .addAnnotatedClass(InstructorDetail.class)
17.                                 .addAnnotatedClass(Course.class)
18.                                 .buildSessionFactory();
19.
20.         // create session
21.         Session session = factory.getCurrentSession();
22.
23.         try {
24.
25.             // start a transaction
26.             session.beginTransaction();
27.
28.             // get the instructor from db
29.             int theId = 1;
30.             Instructor tempInstructor = session.get(Instructor.class, theId);
31.
32.             System.out.println("luv2code: Instructor: " + tempInstructor);
33.
34.             // commit transaction
35.             session.getTransaction().commit();
36.
37.             // close the session
38.             session.close();
39.             System.out.println("\nluv2code: The session is now closed!\n");
40.             //
41.             // THIS HAPPENS SOMEWHERE ELSE / LATER IN THE PROGRAM
42.             // YOU NEED TO GET A NEW SESSION
43.             //
44.
45.             System.out.println("\n\nluv2code: Opening a NEW session \n");
46.             session = factory.getCurrentSession();
47.
48.             session.beginTransaction();
49.
50.             // get courses for a given instructor
51.             Query<Course> query = session.createQuery("select c from Course c "
52.                                                       + "where c.instructor.id=:theInstructorId"
    ,
53.                                                       Course.class);
54.
55.             query.setParameter("theInstructorId", theId);
56.
57.             List<Course> tempCourses = query.getResultList();
58.
59.             System.out.println("tempCourses: " + tempCourses);
60.
61.             // now assign to instructor object in memory
```

```
            tempInstructor.setCourses(tempCourses);

            System.out.println("luv2code: Courses: " + tempInstructor.getCourses());

            session.getTransaction().commit();

            System.out.println("luv2code: Done!");
        }
        finally {

            // add clean up code
            session.close();

            factory.close();
        }
    }
}
```

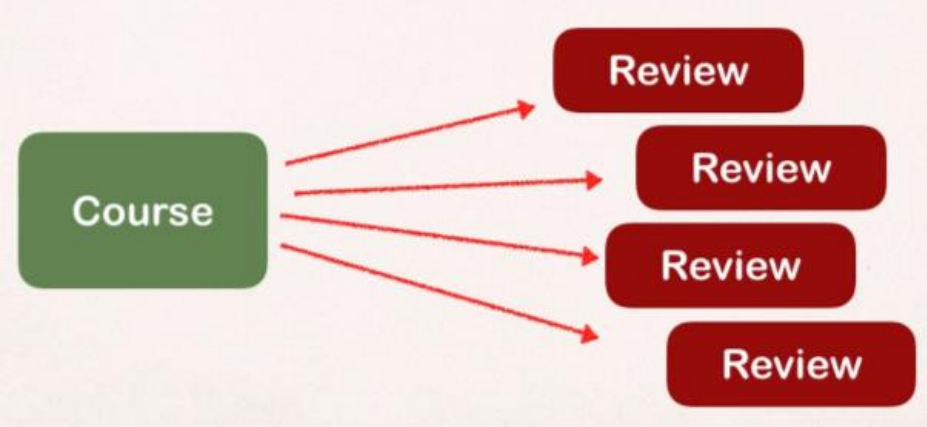
Section 26: Hibernate Advanced Mappings - @OneToMany - Unidirectional

9 / 9 | 38min

- **One-To-Many with Unidirectional Support** Java Project Link

Use Case

A course can have many reviews.

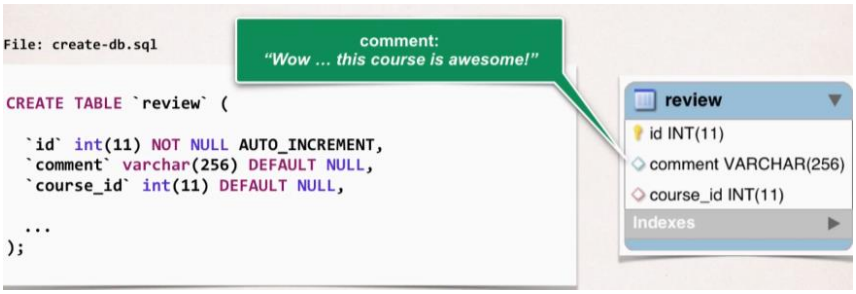


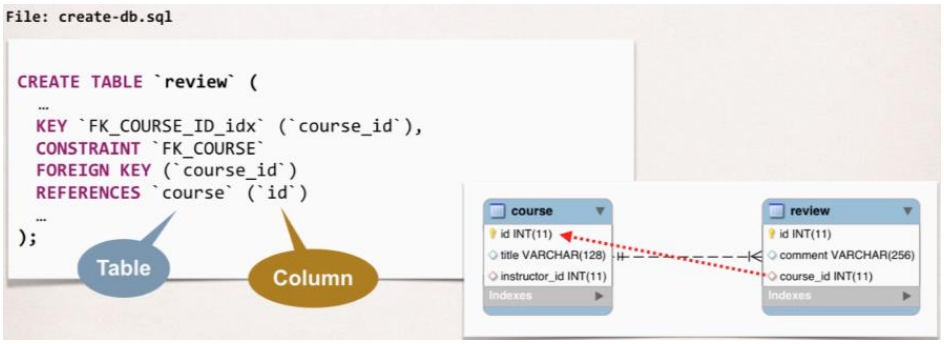
Real world Project Requirement

- If you delete a course, also delete the reviews.
- Reviews without a course have no meaning.

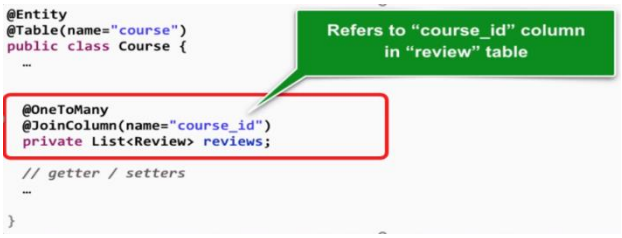
Development process

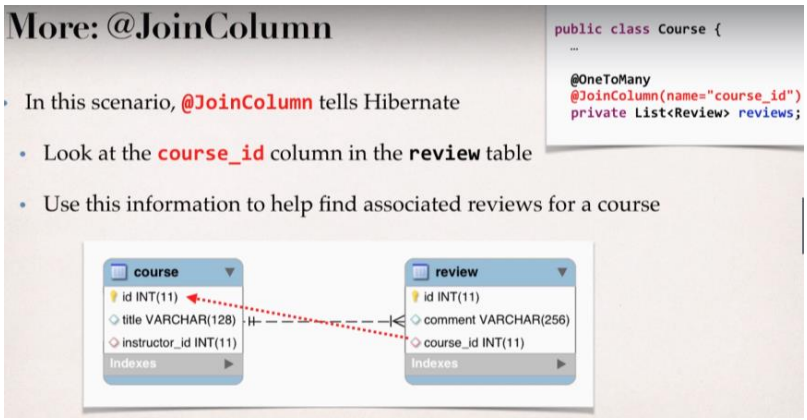
- Prep Work – Define Database Tables SQL SCRIPTS
  Create the schema into MySQL Workbench using the SQL scripts in the above link. And make sure to update hibernate.cfg.xml file with the new Schema.

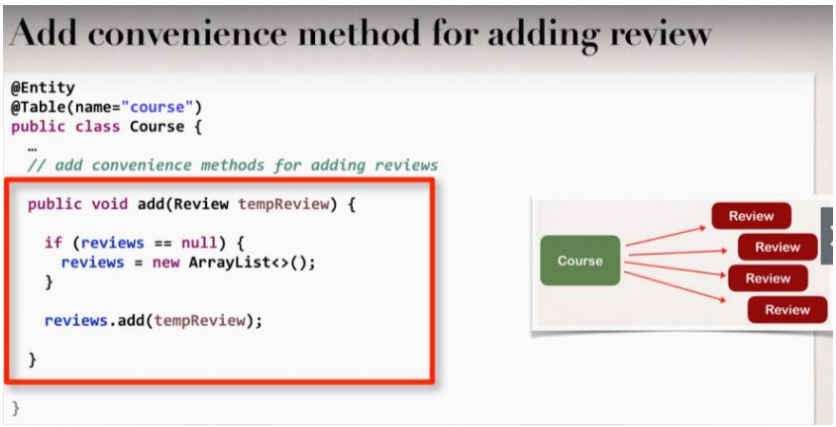




- Create **Reviews** Class Review.java
- Update **Course** Class Course.java







- Create **Main App**

FAQ: @JoinColumn ... where does it find the column?

@JoinColum ... where does it find the column?
Question

In the Course class, we have OneToMany relation with reviews with join column course_id.

But in course table we do not have column course_id.

Ideally when we say @JoinColumn a new column should be created in course table ... isn't it?

How does @JoinColum know where to find the join column?

Answer

The JoinColumn is actually fairly complex and it goes through a number of advanced steps to find the desired column.

This info below is from the documentation

Source: http://docs.oracle.com/javaee/7/api/javax/persistence/JoinColumn.html#name

---

The table in which it is found depends upon the context.

- If the join is for a `OneToOne` or ManyToOne mapping using a foreign key mapping strategy, the foreign key column is in the table of the source entity or embeddable.

- If the join is for a unidirectional `OneToMany` mapping using a foreign key mapping strategy, the foreign key is in the table of the target entity.

- If the join is for a `ManyToMany` mapping or for a `OneToOne` or `bidirectional ManyToOne/OneToMany` mapping using a join table, the foreign key is in a join table.

- If the join is for an element collection, the foreign key is in a collection table.

--

So, as you can see, it depends on the context.

In our training video, we are using `@OneToMany uni-directional` (course has one-to-many reviews).

As a result, the join column / foreign key column is in the target entity. In this case, the target entity is the Review class. So, you will find the join column "course_id" in the "review" table.

File: CreateCourseAndReviewsDemo.java

First, we are adding some courses and their reviews simultaneously.

```
//create a course
Course tempCourse = new Course("Pacman - How to score one Million Points");

//add some reviews
tempCourse.addReview(new Review("Great course....loved it!"));
tempCourse.addReview(new Review("Cool course, job well done"));
tempCourse.addReview(new Review("What a dumb course, your are an idiot!"));

//save the course and leverage the cascade all
session.save(tempCourse);
```

File: GetCourseAndReviewsDemo.java

Now we are fetching the courses and their revies using the id primary key.

```
//get the course
int theId = 10;
Course tempCourse = session.get(Course.class, theId);

//print the course
System.out.println(tempCourse);

//print the course reviews
System.out.println(tempCourse.getReviews());
```

File: DeleteCourseAndReviewDemo.java

Deleting the course and since we have used cascadeType.ALL then it will also delete the associated reviews.

```
//get the course
int theId = 10;
Course tempCourse = session.get(Course.class, theId);

//print the course
System.out.println("Deleting the course...");
System.out.println(tempCourse);

//Delete the course reviews
System.out.println(tempCourse.getReviews());
session.delete(tempCourse);
```

Section 27: Hibernate Advanced Mappings - @ManyToMany

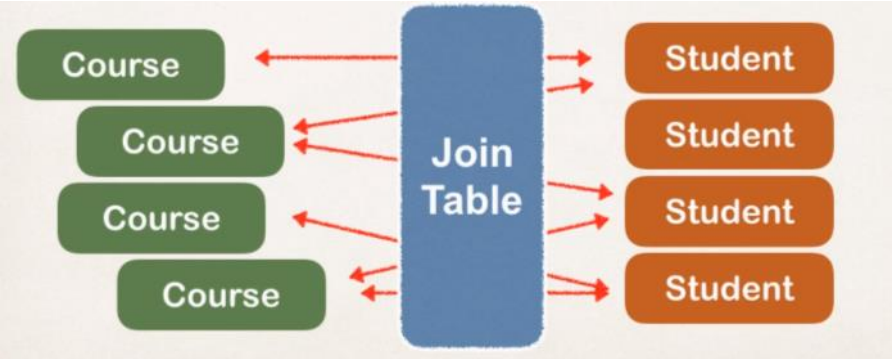
13 / 13 | 56min

Overview:

Use Case

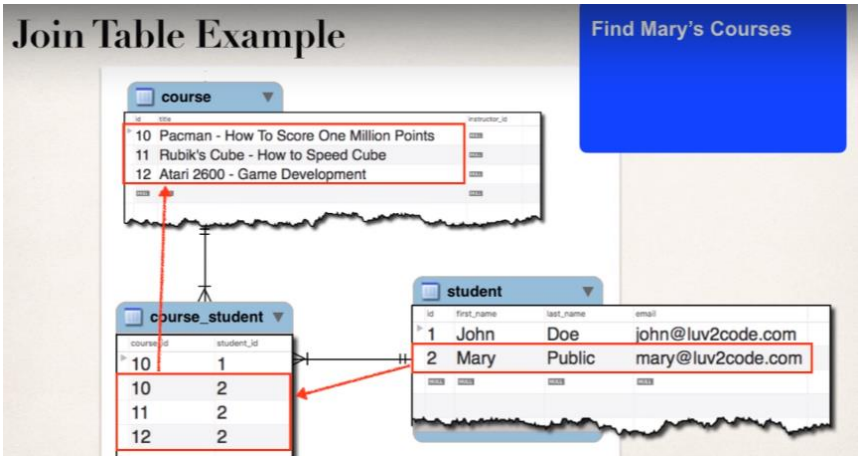
1. A course can have many students
2. A student can subscribe to many courses.

So, we need to keep track of which student is in which course and vice-versa.



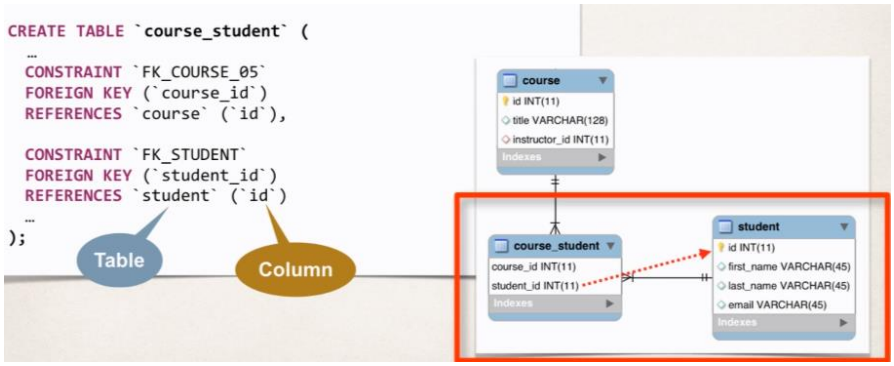
We can make use of Join Table and this Join table will tell us who joined of which course.

A Join Table that provides mapping between tables. It has foreign keys for each table to define the mapping relationship.



Development Process

- Prep work – Define Database Tables SQL Script and make sure to update hibernate.cfg.xml file to get connected t

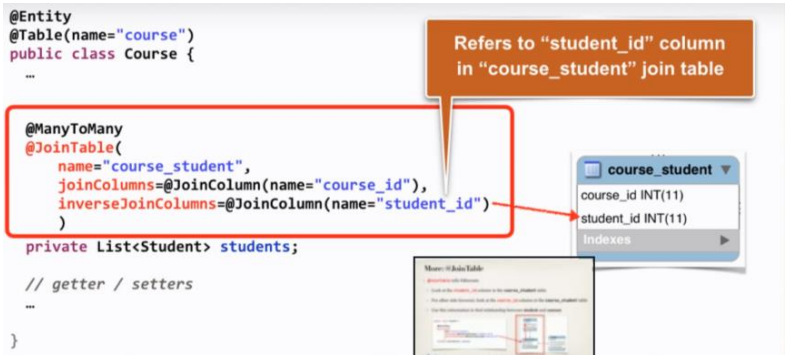


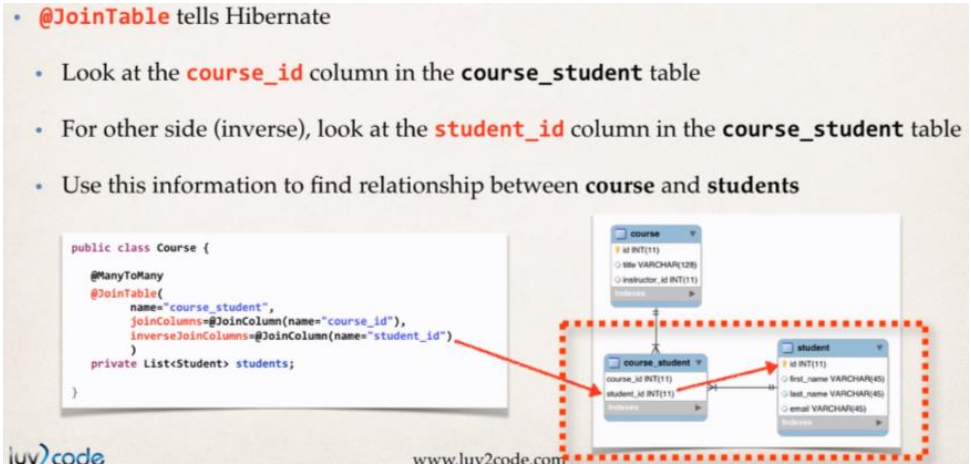
- Update **Course** class. Course.java

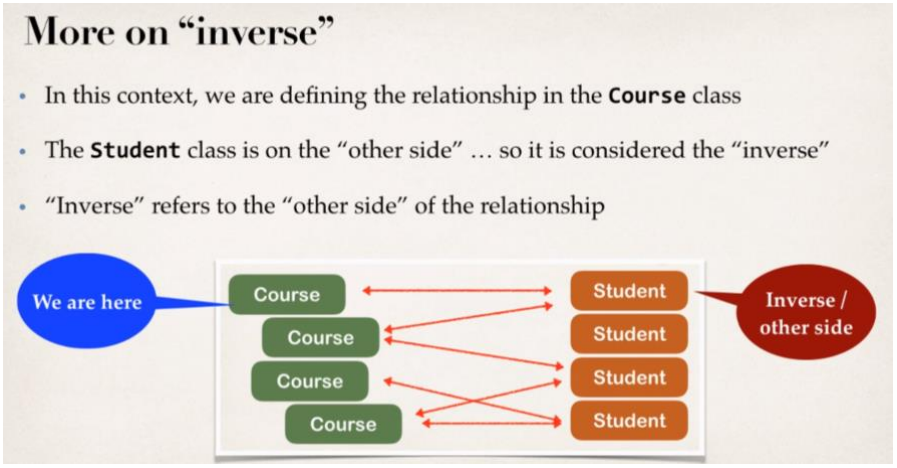
  Here we specify

```
@JoinTable(
name = "course_student",
joinColumns = @JoinColumn(name="course_id"),
inverseJoinColumns = @JoinColumn(name="student_id")
)
private List<Student> students;
```

  `name` tells hibernate which join table we are using in DB.
  `joinColumns` tells hibernate to refer to course_id column in course_student join table.
  `inverseJoinColumns` tells hibernate to refer student_id in course_student join table for inverse column to find the appropriate students in the table.





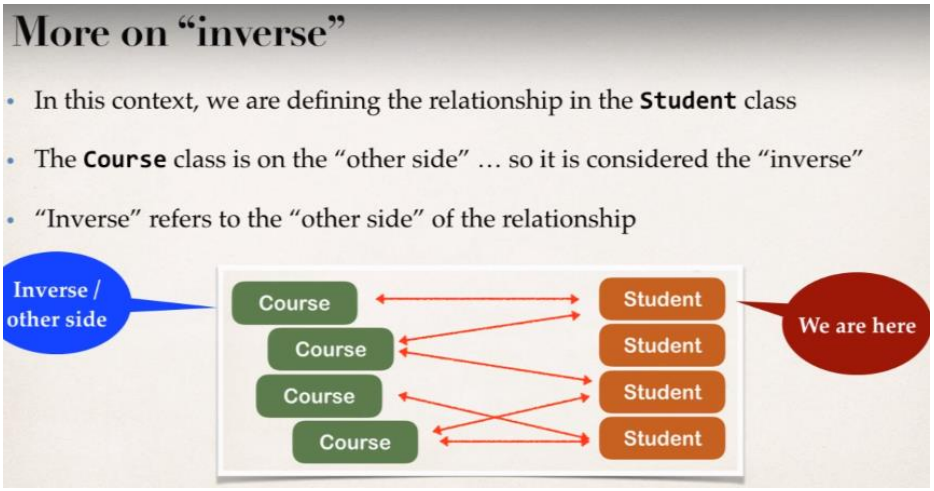


- Update **Student** Class. Student.java

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

  @ManyToMany
  @JoinTable(
      name="course_student",
      joinColumns=@JoinColumn(name="student_id"),
      inverseJoinColumns=@JoinColumn(name="course_id")
      )
  private List<Course> courses;

  // getter / setters
  …

}
```

Refers to "student_id" column in "course_student" join table

course_student
course_id INT(11)
student_id INT(11)
Indexes



- Create Main App

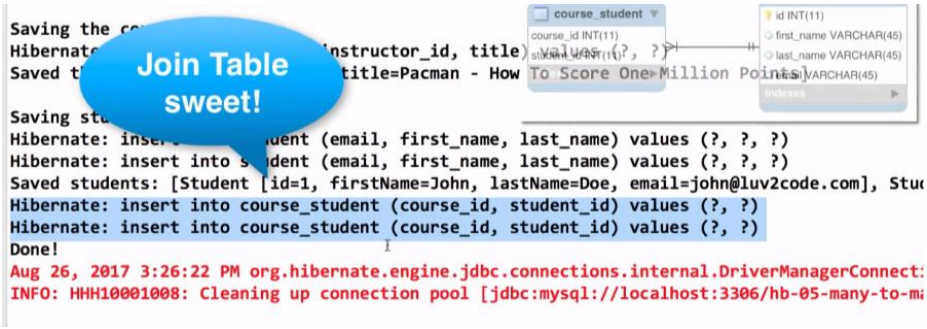
Project Requirement

1. If you delete a course, do not delete the students. So don't apply the cascading delete.
2. Similarly, we do on the other side as well.

- File: CreateCourseAndStudentDemo.java
  Creating course and some students and then adding students into the associated courses.

```
//create a course
Course tempCourse = new Course("Pacman - How To Score One Million Points");

//save the course
System.out.println("\nSaving the course..");
session.save(tempCourse);

//create the student
Student theStudent1 = new Student("John", "Doe", "john@luv2code.com");
Student theStudent2 = new Student("Mary", "Public", "mary@luv2code.com");

//add Students to the course
tempCourse.addStudent(theStudent1);
tempCourse.addStudent(theStudent2);

//save the students
System.out.println("\nSaving the Students....");
session.save(theStudent1);
session.save(theStudent2);
System.out.println("Saved Students: "+tempCourse.getStudents());
```

Here is the log output associated with our current main() app.



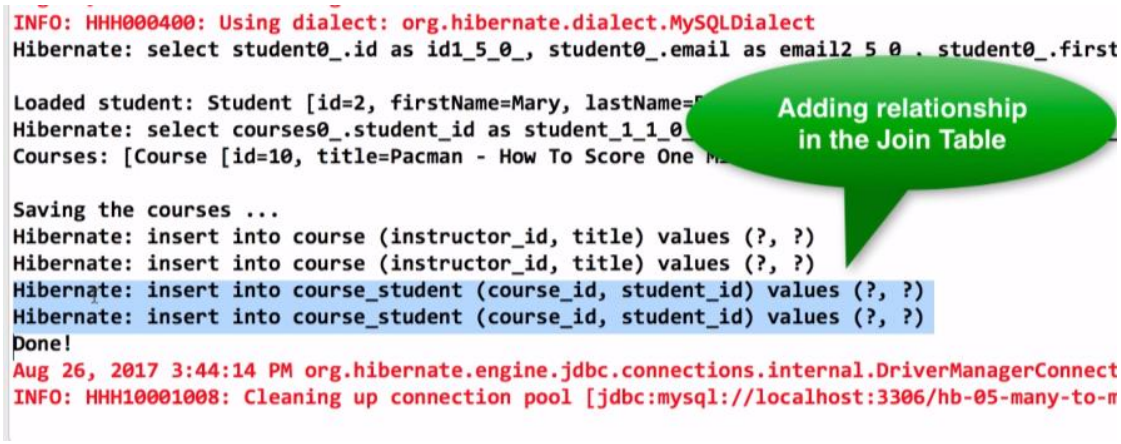
File: AddCoursesForMaryDemo.java

So, we creating some more courses and adding them to the student Mary.
By the id of Mary and fetching the Mary details; and the creating courses and later adding them to Mary as associated courses.

```
//get the student Mary from database
int theId = 2;
Student tempStudent = session.get(Student.class, theId);

System.out.println("\nLoaded student: "+tempStudent);
System.out.println("Course: "+tempStudent.getCourses());

//create more courses
Course tempCourse1 = new Course("Rubik's Cube - How to Speed Cube");
Course tempCourse2 = new Course("Atari 2600 - Game Development");
```

```
//add student to courses
tempCourse1.addStudent(tempStudent);
tempCourse2.addStudent(tempStudent);

//save the courses
System.out.println("\n\nSaving the courses...");
session.save(tempCourse1);
session.save(tempCourse2);
```

✓ log Output associated with current main() app:



File: GetCoursesForMaryDemo.java

Here, getting the courses that we have just added in previous main() app.

```
//get the student Mary from database
int theId = 1;
Student tempStudent = session.get(Student.class, theId);

System.out.println("\nLoaded student: "+tempStudent);
System.out.println("Course: "+tempStudent.getCourses());
```

➢ **Delete a course** but don't delete the student.

File: DeletePacmanCourseDemo.java

First, we are getting the "pacman" course from the database and then deleting the course. And it will delete the mapping for the pacman course but will not delete the associated student.

```
//get the pacman course from the db
int courseId = 10;
Course tempCourse = session.get(Course.class, courseId);

//delete the course
System.out.println("Deleting course: "+tempCourse);
session.delete(tempCourse);
```

- **Delete the Student:** Delete the student and it's mapping from the Join Table. But we should not delete the courses associated.

  File: DeleteMaryStudentDemo.java

```
//get the student Mary from database
int theId = 2;
Student tempStudent = session.get(Student.class, theId);

System.out.println("\nLoaded student: "+tempStudent);
System.out.println("Course: "+tempStudent.getCourses());


//delete the student
System.out.println("Deleting the Student");
session.delete(tempStudent);
```