دانشکده ریاضی و علوم کامپیوتر

برنامه‌سازی پیشرفته و کارگاه

Hibernate

استاد درس

دکتر مهدی قطعی

استاد دوم

بهنام یوسفی مهر

نگارش

سیدآرمان حسینی

بهار ۱۴۰۳

فهرست

"The evening's the best part of the day. You've done your day's work. Now you can put your feet up and enjoy it."

— Kazuo Ishiguro, The Remains of the Day

مقدمه

به انتهای مسیر رسیدیم! بالاخره، آخرین داک برنامه‌نویسی پیشرفته‌تون این‌جاست. توی داک‌های قبلی، ما در کنار هم جاوا رو، از println تا class و inheritance یاد گرفتیم، کلی repo درست کردیم، با گرافیک ور رفتیم، و حتی یه خورده دیتابیس یاد گرفتیم. یادگیری هر کدوم از این‌ها، یک دستاورد بزرگه، این‌جا متوقف شید و به خودتون افتخار کنید![1]

توی این داک، دانش‌مون از جاوا و SQL رو کنار هم می‌ذاریم و یاد می‌گیریم که چطور توی برنامه‌هامون از دیتابیس‌های مختلف استفاده کنیم. برای این کار، از یه ابزار خیلی قدرتمند به اسم Hibernate استفاده می‌کنیم.

[1] من این نقاشی رو از کتابِ Crafting Interpreters دزدیدم! خیلی کتاب قشنگیه، اگر خواستین یه نگاه بهش بندازین.

ORMها

ما بلدیم که کلاس‌های جدید توی جاوا تعریف کنیم:

```
class Task {
    int id;
    String title;
    LocalDate dueDate;
}
```

همچنین، بلدیم که جدول‌های جدید هم توی MySQL تعریف کنیم:

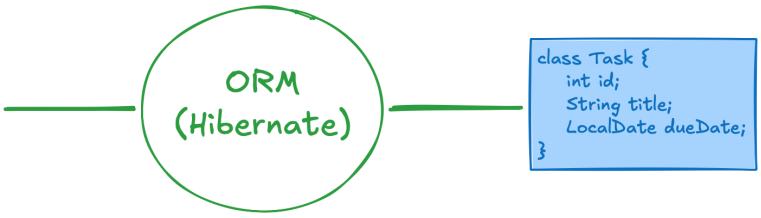
```
create table if not exists tasks(
    id int primary key auto_increment,
    title nvarchar(255),
    due_date date
);
```

واضحا، کلاس Task و جدول tasks به هم مربوطن. ولی چطور این ارتباط رو به جاوا بفهمونیم؟ چطور بهش بگیم که «ما یه جدول به اسم tasks داریم، که دقیقا شبیه کلاس Taskئه. می‌تونی ازش برای ذخیره کردن و خوندن Taskهای مختلف استفاده کنی.». این کار، با این که نسبتا ساده به نظر می‌رسه، در عمل خیلی پیچیده می‌شه.

ORMها، دقیقا توی همین کار به ما کمک می‌کنن. ORMها، که مخفف Object-Relational Mapping هستن، به ما اجازه می‌دن که به راحتی آبجکت‌های برنامه‌مون رو به جدول‌های دیتابیس‌مون مرتبط کنیم و توی برنامه‌هامون از دیتابیس استفاده کنیم. یه ORM، بین دیتابیس و برنامهٔ شما قرار می‌گیره و اون‌ها رو به هم پیوند می‌ده:

tasks

id	name	due_date
1	Meeting with costumers	2025-12-13
2	Fix bathroom	2025-12-13
3	Develop "task completion"...	2025-12-14
5	Fix "task lists" bug	2025-12-14
6	Test "task completion"...	2025-12-20
7	Test "tag" feature	2025-12-20
8	Test "task lists" bug	2025-12-25
4	Develop "tag" feature	2025-12-15

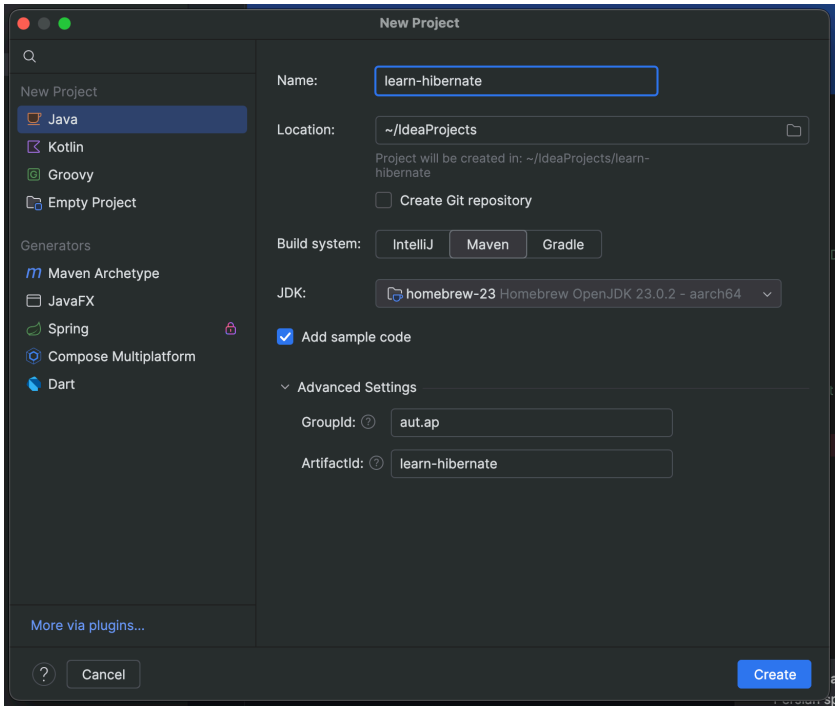


چندتا ORM خیلی خوب برای جاوا هست که یکی از معروف‌ترین‌هاشون Hibernateئه. این‌جا هم، همون‌طور که از اسم داک پیداست، ما از همین framework برای ارتباط با دیتابیس استفاده می‌کنیم.

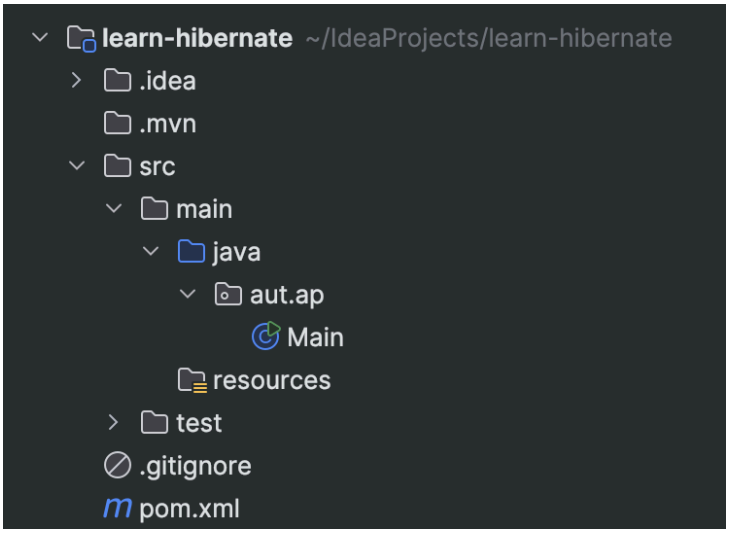
آماده‌سازی پروژه

Hibernate، چیزی نیست جز کدهای جاوایی که یه تعدادی Developer دیگه زدن و منتشر کردن تا شما هم بتونید ازشون استفاده کنید. برای به کار گرفتن این فریم‌ورک، ما از ابزاری به اسم Maven استفاده می‌کنیم.

توی IntelliJ یه پروژهٔ جدید درست کنید، ولی توی صفحهٔ ایجاد پروژهٔ جدید، Build System رو از روی IntelliJ بردارین و Maven رو انتخاب کنین. علاوه بر این، توی Additional Settings، GroupId رو به aut.ap تغییر بدین:



نهایتا دکمهٔ Create رو بزنید. به اولین پروژهٔ Mavenتون خوش اومدین! ساختار پروژه‌تون یه خورده عوض شده و الآن یه خورده جدی‌تر به نظر می‌رسه. ولی نگران نباشید، هنوزم همه چیزش جاواست! بیاین ساختار جدید پروژه‌مون رو ببینیم:

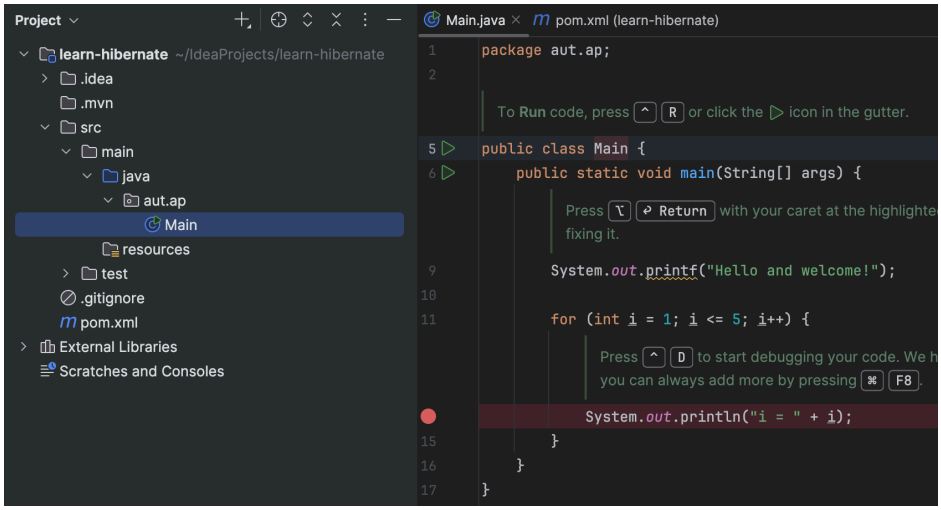


با دایرکتوری‌های idea. و mvn. کار خاصی نداریم. IntelliJ و Maven اطلاعاتی که از پروژهٔ ما نیاز دارن رو توی این دایرکتوری‌ها نگه می‌دارن و ما بهشون دست نمی‌زنیم. ولی باقی فایل‌ها و دایرکتوری‌ها به این شکل هستن:

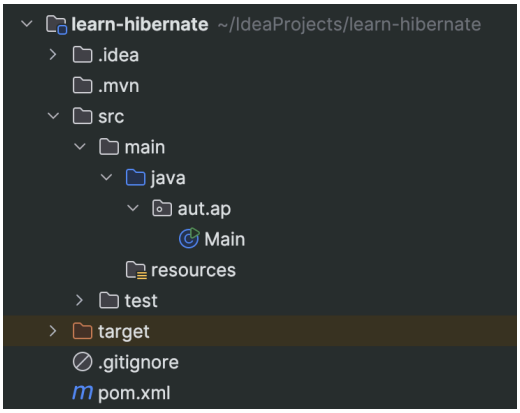
- **دایرکتوری src:** این دایرکتوری، الآن شامل دو دایرکتوریِ جدید main و testئه.
  - **دایرکتوری test:** توی دنیای واقعی، برنامه‌های موفق قبل از انتشار نسخه‌های جدید تست می‌شن تا دولوپرها (و رئیس‌ها!) مطمئن بشن که برنامهٔ جدیدشون باگ خاصی نداره. خیلی از این تست‌ها، به جای آدمیزاد، توسط کامپیوتر انجام می‌شه و دایرکتوری test شامل کدهای پشت این testهاست. ما توی AP کار خاصی با این دایرکتوری نداریم، ولی اگر دوست دارین خودتون می‌تونید این ویدئو رو راجع بهش ببینید و برنامه‌هاتون رو راحت‌تر تست کنید!
  - **دایرکتوری main:** این دایرکتوری، شامل کدهای برنامه‌مونه. خود main شامل دو دایرکتوری java و resources هست.
    - **دایرکتوری java:** این دایرکتوری جاییه که کدهای جاوامون رو می‌نویسیم. شبیه همون srcی پروژه‌های قدیمی‌مونه! اگر دقت کنید، الآن هم پکیج aut.ap و فایل Mainمون توشه.
    - **دایرکتوری resources:** لزوما همهٔ فایل‌های مرتبط با برنامهٔ ما، کد جاوا نیستن. خیلی‌هاشون عکسن، configان، و کلی چیزهای دیگه. فایل‌های غیر کدی برنامه‌مون رو این‌جا می‌ذاریم.

- **فایل gitignore.**: دیگه تا این‌جای کار خوب می‌دونید که این فایل چه کار می‌کنه!
- **فایل pom.xml:** این فایل، مهم‌ترین چیزیه که به پروژه‌مون اضافه شده. از این به بعد که ما برای build کردن کدهامون از Maven استفاده می‌کنیم، می‌تونیم توی این فایل، اطلاعات پروژه‌مون رو به Maven توضیح بدیم. مثلا اگر بخوایم بگیم که «پروژهٔ ما از Hibernate استفاده می‌کنه»، اون رو این‌جا می‌نویسیم. یه مقدار جلوتر با هم دیگه این فایل رو عوض می‌کنیم.

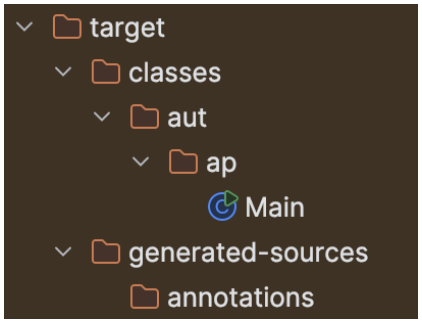
روی فایل Main کلیک کنید:



جاوای آشنای ما این‌جاست! دکمهٔ run رو بزنید تا برنامه‌تون اجرا بشه. بعد از اجرای برنامه‌تون، دایرکتوری target به ساختار پروژه‌تون اضافه می‌شه:



این دایرکتوری، شبیه همون دایرکتوری out قدیمی‌تونه و خروجی buildمون رو نگه می‌داره. اگر یه نگاه داخلش بندازین هم خروجی build برنامه‌تون رو می‌بینید:



خیلی هم خوب. حالا که با ساختار جدید پروژه‌هاتون آشنا شدین[2]، بیاین تا Hibernate رو به برنامه‌مون اضافه کنیم.

اضافه کردن Hibernate

همون‌طور که گفتیم، Hibernate صرفا یک کتابخونه‌ست پر از کدهای جاوا[3] که به شما توی برقراری ارتباط با دیتابیس‌تون کمک می‌کنه. Hibernate دقیقا شبیه پکیج‌های دیگه‌ایه که تا الآن ابتدای کد جاواتون import می‌کردین، با این تفاوت که به طور پیش‌فرض توی پروژهٔ شما وجود نداره و باید قبل از استفاده از اون، کدهاش رو دریافت کنید.

به فایل pom.xml برین. این فایل الآن یک همچین شکل و قیافه‌ای داره:

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>

    <groupId>aut.ap</groupId>
    <artifactId>learn-hibernate</artifactId>
    <version>1.0-SNAPSHOT</version>

    <properties>
        <maven.compiler.source>23</maven.compiler.source>
        <maven.compiler.target>23</maven.compiler.target>
        <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
    </properties>
</project>
```

[2] در مورد ساختار پروژه‌هایی که با Maven درست می‌شن، این‌جا رو هم یه نگاه بندازین.

[3] اگر بخواین می‌تونید یه نگاه به ریپوی گیت‌هاب Hibernate بندازین. یه دور توش بزنید و ببینید که دروغ نگفتم! اون‌ها هم واقعا class و interface و annotation تعریف کردن!

به همچین فایلی، XML می‌گن. اگر قبلا با HTML کار کرده باشین ظاهرش براتون یه خورده آشناست. این فایل پروژهٔ شما رو برای Maven توصیف می‌کنه.

زیر تگ <properties/> توی خط یکی مونده به آخر، قبل از <project/>، بخش dependencies رو اضافه کنید:

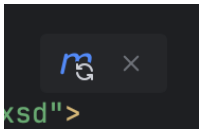
```
<dependencies>
    <dependency>
        <groupId>org.apache.logging.log4j</groupId>
        <artifactId>log4j-core</artifactId>
        <version>2.20.0</version>
    </dependency>

    <dependency>
        <groupId>jakarta.persistence</groupId>
        <artifactId>jakarta.persistence-api</artifactId>
        <version>3.1.0</version>
    </dependency>

    <dependency>
        <groupId>org.hibernate.orm</groupId>
        <artifactId>hibernate-core</artifactId>
        <version>6.6.13.Final</version>
    </dependency>

    <dependency>
        <groupId>com.mysql</groupId>
        <artifactId>mysql-connector-j</artifactId>
        <version>9.3.0</version>
    </dependency>
</dependencies>
```

این‌ها، همگی packageها و libraryهایی‌ان که برای پروژه‌مون نیاز داریم. بیاین یه خورده اون‌ها رو بهتون معرفی کنیم:

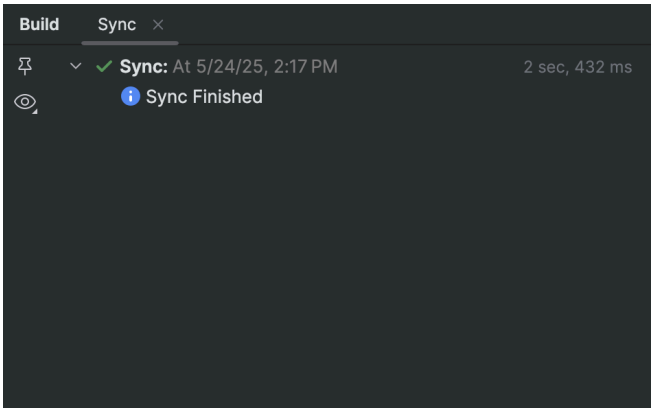
- **log4j-core:** این پکیج، به Hibernate توی لاگ کردن بعضی اطلاعات کمک می‌کنه. در ادامه می‌بینین که Hibernate چطور ازش استفاده می‌کنه.
- **jakarta.persistence-api:** همون‌طور که گفتیم، ORMهای مختلفی توی دنیای جاوا وجود دارن. حدودا همهٔ این ORMها، من جمله Hibernate، interfaceهای تعریف شده توی پکیج jakarta.persistence-api رو پیاده‌سازی می‌کنن. ما خیلی کار مستقیمی با این پکیج نداریم و شما هم لازم نیست الآن خودتون رو درگیرش کنید. فقط لازمه بدونید وقتی که از Hibernate استفاده می‌کنین، لازمه که این پکیج هم به پروژه‌تون اضافه کنید.
- **hibernate-core:** اصل کار ما با اینه! این پکیج، کدهای Hibernate رو نگه‌داری می‌کنه.

- **mysql-connector-j:** Hibernate با دیتابیس‌های مختلفی کار می‌کنه که MySQL تنها یکی از اون‌هاست. ممکنه دیتابیس شما، PostgreSQL، Microsoft SQL Server، H2 یا هر چیز دیگه‌ای باشه. این پکیج، به Hibernate توی ارتباط با دیتابیس‌تون کمک می‌کنه. همین!

حالا که dependencyهای پروژه‌مون رو اضافه کردیم، وقتشه اون‌ها رو نصب کنیم. یه دکمهٔ کوچولو، بالا سمت چپ فایل pom.xml پدیدار شده:



روش کلیک کنید و صبر کنید تا dependencyهاتون کاملا sync بشن. در صورت موفقت یه همچین صفحه‌ای می‌بینید:

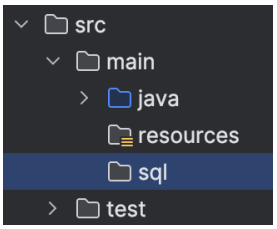


خیلی خب، حالا دیگه می‌تونید از Hibernate استفاده کنید. بریم سراغ قدم بعدی.

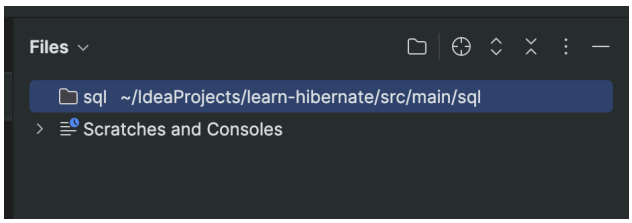
Configuration

حالا که Hibernate روی سیستم‌تونه، باید یه خورده اون رو تنظیم کنید. مثلا، باید بهش بگید که دیتابیس‌تون کجاست و چجوری باید بهش وصل بشه. این کار رو از طریق یه فایل به اسم hibernate.cfg.xml انجام می‌دیم.

قبلش، بیاید اصلا یه دیتابیس برای برنامه‌مون درست کنیم. توی دایرکتوری src/main، یه دایرکتوری جدید به اسم sql درست کنید. همهٔ اسکریپت‌های دیتابیسی‌مون رو این‌جا می‌ذاریم[4]:



دایرکتوری sql رو با DataGrip باز کنید. برای این کار کافیه به DataGrip برین، روی File کلیک کنین و Open رو بزنین. بعدش هم دایرکتوری sql رو توی سیستم‌تون پیدا کنید و اون رو باز کنید. بعد از این کار، تب Files دیتاگریپ‌تون باید همچین شکلی داشته باشه:

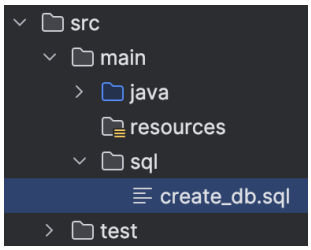


یه فایل به اسم create_db.sql ایجاد کنید و توی اون، یه دیتابیس جدید برای پروژه‌تون درست کنید:

```
create database learn_hibernate_db;
```

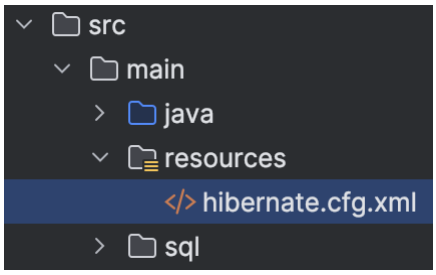
این اسکریپت رو اجرا کنید تا دیتابیس learn_hibernate_db ساخته بشه. توی برنامه‌مون هم از همین دیتابیس استفاده می‌کنیم.

وقتی که مطمئن شدین دیتابیس‌تون ساخته شده، به IntelliJ برگردین. باید فایل create_db.sql رو توی تب Project ببینین:



[4] اسکریپت‌های دیتابیسی مثل create table، بخشی از سورس کد برنامهٔ شمان. مشتری‌هاتون برای این که بتونن دیتابیس خودشون رو بالا بیارن و جداول برنامه‌تون رو توش ایجاد کنن، به این اسکریپت‌ها نیاز دارن. به خاطر همین هم مهمه که اون‌ها رو توی سورس کدتون ذخیره کنید.

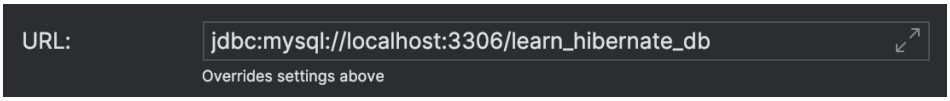
حالا، باید به Hibernate بگیم که چطور می‌تونه به این دیتابیس وصل بشه. توی دایرکتوری src/main/resources، فایل hibernate.cfg.xml رو ایجاد کنید:



توی این فایل، اطلاعات زیر رو بنویسید:

```xml
<?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>
        <!-- JDBC database connection settings -->
        <property name="hibernate.connection.driver_class">
            com.mysql.cj.jdbc.Driver
        </property>
        <property name="hibernate.connection.url">
            YOUR_DB_URL
        </property>
        <property name="hibernate.connection.username">
            YOUR_USERNAME
        </property>
        <property name="hibernate.connection.password">
            YOUR_PASSWORD
        </property>

        <!-- Hibernate settings -->
        <property name="hibernate.show_sql">true</property>
        <property name="hibernate.format_sql">true</property>
        <property name="hibernate.highlight_sql">true</property>
    </session-factory>
</hibernate-configuration>
```

به جای YOUR_DB_URL، آدرس دیتابیس‌تون رو بنویسین. اگر یادتون رفته، این آدرس همون URLای بود که موقع اضافه کردن Data Source جدید، به DataGrip می‌دادین، مثلا برای من این شکلیه:



دقت کنید که حتما اسم خود دیتابیسی که درست کردین هم توی انتهای این URL باشه. به جای YOUR_USERNAME و YOUR_PASSWORD هم یوزرنیم و پسورد دیتابیس‌تون رو بنویسید. مثلا برای من، به ترتیب root و 123 بودن. در نهایت، اطلاعات اتصال به دیتابیس‌تون باید همچین شکلی داشته باشه:

```
<!-- JDBC database connection settings -->
<property name="hibernate.connection.driver_class">
    com.mysql.cj.jdbc.Driver
</property>
<property name="hibernate.connection.url">
    jdbc:mysql://localhost:3306/learn_hibernate_db
</property>
<property name="hibernate.connection.username">
    root
</property>
<property name="hibernate.connection.password">
    123
</property>
```

اگر دقت کنید، من اسم دیتابیسی که درست کرده بودیم هم توی URL دیتابیس‌ام نوشتم. حتما این کار رو بکنید که Hibernate بدونه که باید به کدوم دیتابیس متصل بشه. همچنین، توی property اولمون به Hibernate می‌گیم که این دیتابیس MySQLئه.

سه‌تا property دیگه هم این‌جا مشخص کردیم، این‌ها هم بامزه‌ن:

```
<!-- Hibernate settings -->
<property name="hibernate.show_sql">true</property>
<property name="hibernate.format_sql">true</property>
<property name="hibernate.highlight_sql">true</property>
```

این propertyها بهتون کوئری‌هایی که Hibernate روی دیتابیس می‌زنه رو نشون می‌دن. یکم جلوتر می‌بینید که چه تاثیری روی برنامه‌مون گذاشتن. فعلا بهشون دست نزنید!

تا این‌جایید، فایل log4j2.properties هم توی دایرکتوری resources درست کنید. این فایل، تنظیمات پکیج log4j رو نگه می‌داره و باعث می‌شه خروجی برنامه‌تون خوشگل‌تر باشه. همین! توی این فایل هم اطلاعات زیر رو کپی کنید:

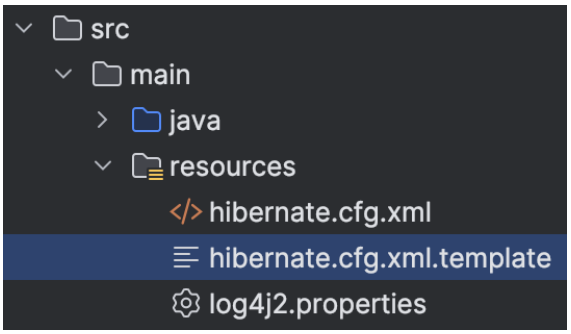
```
rootLogger.level = info

rootLogger.appenderRefs = console
rootLogger.appenderRef.console.ref = console

logger.hibernateSQL.name = org.hibernate.SQL
logger.hibernateSQL.level = info
```

```
logger.hibernateBind.name = org.hibernate.orm.jdbc.bind
logger.hibernateBind.level = trace

appender.console.name = console
appender.console.type = Console
appender.console.layout.type = PatternLayout
appender.console.layout.pattern = %highlight{[%p]} %m%n
```

اطلاعات دیتابیس‌تون رو کامیت نکنید

اگر الآن فایل hibernate.cfg.xml رو کامیت کنید، هر کسی که ریپوتون رو ببینه می‌تونه به یوزرنیم و پسورد دیتابیس‌تون دسترسی داشته باشه. خصوصا اگر از دیتابیس لیارا استفاده می‌کنید، نباید به هیچ وجه این کار رو بکنید. ولی خب، شما که نمی‌تونید تنظیمات Hibernate رو کامیت نکنید، راه حل چیه؟

یه فایل جدید، به اسم hibernate.cfg.xml.template توی دایرکتوری resources درست کنید:



توی فایل جدید، اطلاعات زیر رو وارد کنید:

```
<?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>
        <!-- JDBC database connection settings -->
        <property name="hibernate.connection.driver_class">
            com.mysql.cj.jdbc.Driver
        </property>
        <property name="hibernate.connection.url">
            YOUR_DB_URL
        </property>
        <property name="hibernate.connection.username">
            YOUR_USERNAME
        </property>
        <property name="hibernate.connection.password">
            YOUR_PASSWORD
        </property>
```

```
        <!-- Hibernate settings -->
        <property name="hibernate.show_sql">true</property>
        <property name="hibernate.format_sql">true</property>
        <property name="hibernate.highlight_sql">true</property>
    </session-factory>
</hibernate-configuration>
```

ولی این بار، اطلاعات دیتابیس‌تون رو توی فایل template وارد نکنید. حالا فایل hibernate.cfg.xml رو به انتهای gitignore. اضافه کنید:

```
### Config ###
src/main/resources/hibernate.cfg.xml
```

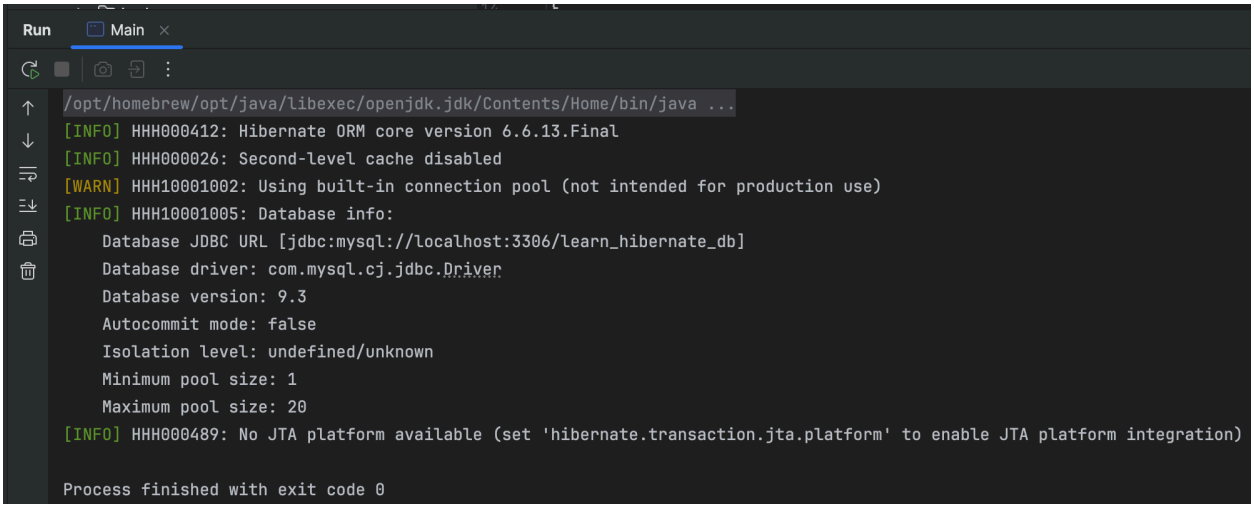
حالا اگر کسی یه نگاه به ریپوی پروژه‌تون بندازه، به جای فایل hibernate.cfg.xml، فایل hibernate.cfg.xml.template رو می‌بینه که اطلاعات حساس دیتابیس‌تون رو توش ننوشتین! عالی! فقط حواستون باشه که اگر توی فایل hibernate.cfg.xml تغییری دادین، فایل hibernate.cfg.xml.template رو هم تغییر بدین.

تست پروژه

قبل از شروع به کدزنی، باید تست کنیم که تنظیمات‌مون تا این‌جای کار درست بوده یا نه. به سراغ تابع main برین و کد زیر رو اون‌جا کپی کنید. فعلا کاری با این که این کد چطور کار می‌کنه نداریم و صرفا می‌خوایم اتصال‌تون به دیتابیس رو تست کنیم:

```
package aut.ap;

import org.hibernate.SessionFactory;
import org.hibernate.cfg.Configuration;

public class Main {
    public static void main(String[] args) {
        SessionFactory sessionFactory = new Configuration()
                .configure("hibernate.cfg.xml")
                .buildSessionFactory();

        sessionFactory.close();
    }
}
```

importهای اول کدمون رو می‌بینید؟ این، اولین کد Hibernate شماست! باید بدون خطا هم برای شما اجرا بشه:

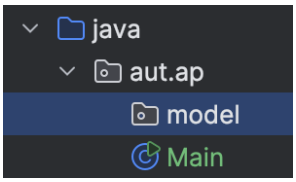


خروجی برنامه‌تون، شامل یه سری log از طرف Hibernateئه. مثلا اطلاعات دیتابیس‌تون رو می‌تونید این‌جا ببینید. می‌بینید که خروجی این برنامه، چقدر قشنگ و مرتبه؟ این‌ها به خاطر پکیج log4jئه که بالاتر نصبش کردیم.

اگر توی اجرای این برنامه به خطا خوردین، خطا رو بخونید و سعی کنید برطرفش کنید. اگر نشد، اولین تدریس‌یار نزدیک‌تون رو خفت کنید که توی برطرف کردن خطا کمک‌تون کنه!

Entityها در Hibernate

بیاید کلاس Task رو به برنامه‌مون اضافه کنیم. توی پکیج aut.ap، یه پکیج جدید به اسم model ایجاد کنید:



و به این پکیج، کلاس Task رو اضافه کنید:

```
public class Task {
    private Integer id;

    private String name;

    private LocalDate dueDate;
}
```

اگر دقت کنید، توی این کلاس فیلد dueDate به جای این که از جنس Date باشه، از جنس LocalDateئه. وقتی دارید با Hibernate کار می‌کنید بهتره که فیلدهای زمانی‌تون از جنس Date نباشن.[5]

راستی، حواستون باشه که ما id رو به جای int، Integer تعریف کردیم. وقتی ببینیم که idها چطور توی Hibernate هندل می‌شن، توضیح می‌دیم که چرا این کار رو کردیم.

حالا برای فیلدهامون getter و setter تعریف می‌کنیم:

```
public Integer getId() {
    return id;
}

public String getName() {
    return name;
}

public void setName(String name) {
    this.name = name;
}

public LocalDate getDueDate() {
```

[5] منبع این توصیه، این‌جاست.

```
    return dueDate;
}

public void setDueDate(LocalDate dueDate) {
    this.dueDate = dueDate;
}
```

اگر دقت کنید، برای فیلد id، setter تعریف نکردیم. می‌تونستیم بکنیم راستش، ولی همون‌طور که در ادامه می‌بینید، دیتابیسه که id موجودیت‌های ما رو مشخص می‌کنه و به همین خاطر، نیاز مبرمی به اضافه کردن setter برای این فیلد نداریم.

حالا، دوتا constructor هم برای این کلاس تعریف می‌کنیم:

```
public Task() {
}

public Task(String name, LocalDate dueDate) {
    this.name = name;
    this.dueDate = dueDate;
}
```

تمام موجودیت‌هایی که Hibernate ازشون استفاده می‌کنه، **باید یک constructor پابلیک داشته باشن که هیچ ورودی‌ای نداره**. همون‌طور که می‌دونید، اگر برای کلاس‌تون constructor جدیدی تعریف نکنید خود جاوا یه constructor خالی و پابلیک براش می‌ذاره. اما این‌جا چون ما می‌خواستیم یه constructor دیگه هم برای کلاس‌مون تعریف کنیم، مجبور شدیم که خودمون یه constructor خالی برای این کلاس بذاریم.

نهایتا هم، برای این که بتونیم تسک‌هامون رو پرینت کنیم، یه متد toString هم براش تعریف می‌کنیم:

```
@Override
public String toString() {
    return "Task{" +
            "id=" + id +
            ", name='" + name + '\'' +
            ", dueDate=" + dueDate +
            '}';
}
```

بیاین برای این که مطمئن شیم همه چی درست کار می‌کنه، یه تسک درست کنیم و پرینتش کنیم:

```
public static void main(String[] args) {
    Task t = new Task("Read Hibernate Document", LocalDate.now());
    System.out.println(t);
}
```

خروجی این کد به شکل زیره:

```
Task{id=null, name='Read Hibernate Document', dueDate=2025-05-24}
```

همون‌طور که می‌بینید، id تسک‌مون مقدار دهی نشده و nullئه. اگر id، به جای Integer، int بود این مقدار 0 می‌شد.[6]

بیاید یه جدول هم برای تسک‌هامون بسازیم. به DataGrip برگردین و توی دایرکتوری sql، اسکریپت tasks.sql رو ایجاد کنید. توی اون، کوئری‌های زیر رو بنویسید:

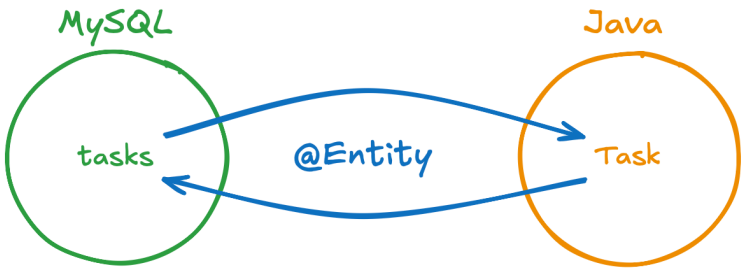
```sql
use learn_hibernate_db;

create table tasks(
    id int primary key auto_increment,
    name nvarchar(255) not null,
    due_date date not null
)
```

این اسکریپت رو اجرا کنید تا جدول tasks هم به وجود بیاد.

حالا که هم کلاس‌مون رو درست کردیم، هم جدول‌مون رو، بیاید اون‌ها رو به هم وصل کنیم!

تعریف Entity جدید

هدف Hibernate، پیوند دادن کلاس‌های جاوا و جداول دیتابیس بود. Hibernate این کار رو از طریق annotationهایی که بالای کلاس‌ها و فیلدهاتون می‌نویسید انجام می‌ده.



[6] اگر یادتون باشه، مقدار دیفالت reference typeها null بود و مقدار دیفالت primitive typeها، بسته به نوعشون، یا صفر بود، یا false یا '\0'.

یکی از اصلی‌ترین annotationهای Hibernate، Entityئه. این annotation باید بالای هر کلاسی که به یه جدول توی دیتابیس وصله حضور داشته باشه. به کلاس Task برگردین و این annotation رو بالای اون بنویسید:

```
@Entity
public class Task {
    // code code code
}
```

دومین چیزی که باید به Hibernate بگین، اینه که این کلاس به چه جدولی وصل می‌شه. برای این کار از annotationای به اسم Table استفاده می‌کنیم و اون رو هم بالای کلاس‌مون می‌نویسیم:

```
@Entity
@Table(name = "tasks")
public class Task {

    // code code code
}
```

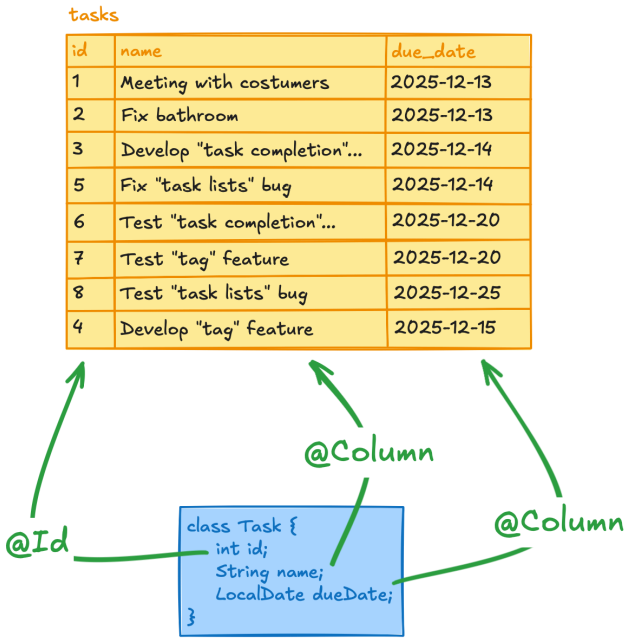
توی این annotation، فیلد name، اسم جدول‌تون رو نشون می‌ده. Hibernate از این به بعد کلاس Task رو به جدول tasks وابسته می‌دونه.

نهایتا، باید به فایل hibernate.cfg.xml برگردین و اون‌جا هم، اسم کامل کلاس‌تون رو (به همراه اسم پکیجش) برای hibernate بنویسید. برای این کار، تکه کد زیر رو بعد از بخش Hibernate Settings و قبل از ‹session-factory\› کپی کنید:

```
<!-- Mappings -->
<mapping class="aut.ap.model.Task"/>
```

تعریف اطلاعات ستون‌ها

حالا، Hibernate می‌دونه که کلاس Task و جدول tasks به هم دیگه مرتبطن. ولی فیلدهای Task و ستون‌های tasks چطور؟ مثلا Hibernate از کجا بدونه که فیلد dueDate و ستون due_date به هم مرتبطن؟ یا این که فیلد id، در واقع primary key جدول tasksئه. ما باید با annotationهای جدید فیلدهای کلاس Task هم برای Hibernate توضیح بدیم.



از فیلد id شروع می‌کنیم. ستون id توی جدول tasks دو خاصیت اصلی داره:

1. این ستون، primary keyئه.
2. این ستون، auto_incrementئه. به همین خاطر لازم نیست که توی هر insert، مقدار اون رو به دیتابیس ورودی بدیم.

هر دوی این خواص رو می‌تونیم به Hibernate بفهمونیم. برای این که بهش بگیم فیلد id توی جدول‌مون primary keyئه از annotationای به اسم Id استفاده می‌کنیم:

```
@Id
private Integer id;
```

و برای این که به Hibernate بگیم که این ستون auto_incrementئه و اتوماتیک مقداردهی می‌شه، از annotationای به اسم GeneratedValue استفاده می‌کنیم. این annotation به Hibernate می‌گه که id توسط دیتابیس داده می‌شه و ما لازم نیست مقداردهی‌ش کنیم:

```
@Id
@GeneratedValue(strategy = GenerationType.IDENTITY)
private Integer id;
```

خوبه بدونید که ممکنه ما شیوه‌های مختلفی به جز auto_increment برای مقداردهی ستون id داشته باشیم. مثلا شاید بخوایم به جای این که idی رکوردهای جدید، یکی بیشتر از آخرین رکورد باشه، دو برابر اون باشه! این استراتژی برای مشخص کردن idی رکوردهای جدید خیلی احمقانه‌ست، ولی

شدنیه و به ما اثبات می‌کنه که تنها راه مقداردهی اتوماتیک id، auto_increment نیست. به خاطر همین موضوع، ما توی GeneratedValue مشخص کردیم که استراتژی‌مون برای idهای جدید، GenerationType.IDENTITYئه. این enum برای Hibernate، معادل همون auto_incrementاینه که توی MySQL نوشتیم، به خاطر همین هم شما، اگر کار عجیب غریبی نکنید، احتمالا از همین مقدار برای strategy استفاده بکنید.

حالا لازمه که باقی فیلدهامون هم به ستون‌های متناظرشون متصل کنیم. از name شروع می‌کنیم. این فیلد یک خاصیت توی دیتابیس‌مون داره، این که not nullئه. با استفاده از annotationای به اسم Basic می‌تونیم این رو به Hibernate بفهمونیم:

```
@Basic(optional = false)
private String name;
```

توی این annotation، مقدار optional رو false تعیین کردیم. این یعنی فیلد name برای Taskهامون اختیاری نیست و نمی‌تونه null باشه.

به سراغ فیلد dueDate می‌ریم. ستون متناظر این فیلد، یعنی due_date، not nullئه و مجددا لازمه Basic رو بالای این فیلد بنویسیم:

```
@Basic(optional = false)
private LocalDate dueDate;
```

علاوه بر اون، اسم این فیلد هم یه مقدار با ستون due_date متفاوته. به خاطر همین موضوع، ما باید به Hibernate بگیم که این فیلد رو، متناظر با ستون due_date در نظر بگیره. برای این کار از annotationای به اسم Column استفاده می‌کنیم:

```
@Basic(optional = false)
@Column(name = "due_date")
private LocalDate dueDate;
```

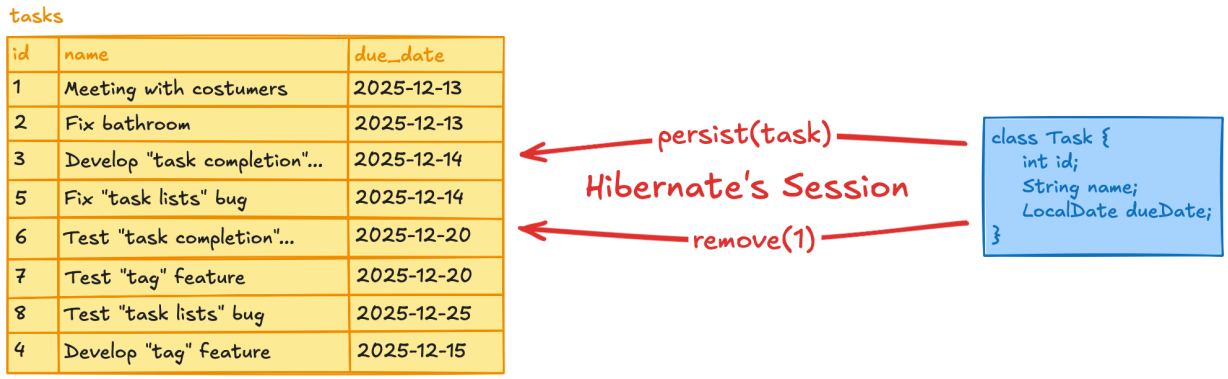
مقدار فیلد name این annotation رو، با اسم ستونِ متناظرِ فیلدمون پر می‌کنیم. اگر دقت کنید، برای فیلدهای id و name کلاسمون نیازی به این کار نداشتیم، چون که این فیلدها کاملا هم‌اسم ستون‌های متناظرشون، یعنی id و name بودن.

Sessionها و اجرای دستورات SQL

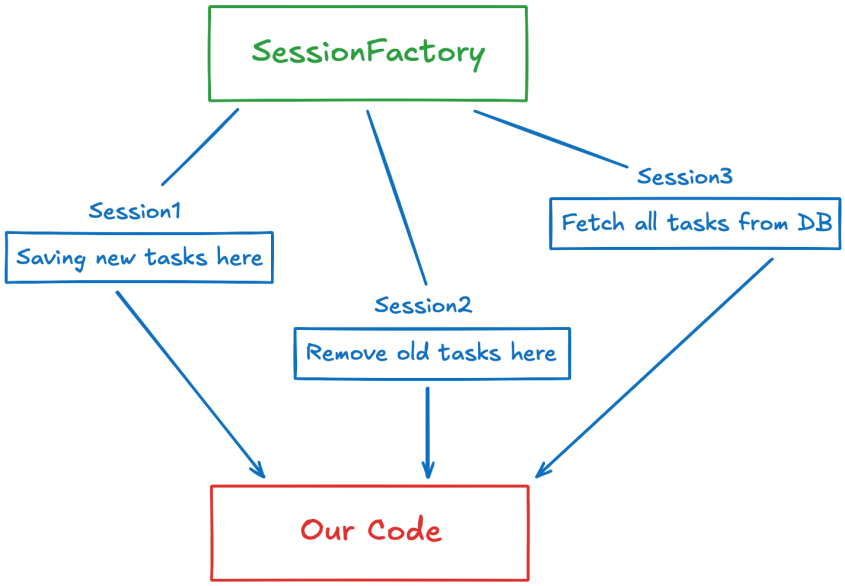
کم کم داریم به بخش اصلی Hibernate می‌رسیم! وقتشه که با استفاده از کدهای جاوامون، دیتابیس رو تغییر بدیم. قبل از اون اما، لازمه که با مفهوم Session توی Hibernate آشنا بشیم.

Sessionها

توی Hibernate، Sessionها هستن که ارتباط ما رو با دیتابیس برقرار می‌کنن. Sessionها به ما اجازه می‌دن که رکوردهای جدید به دیتابیس اضافه کنیم، اون‌ها رو پاک کنیم، آپدیتشون کنیم، یا اون‌ها رو از دیتابیس بخونیم. هر کوئری‌ای که تا الآن توی دیتابیس‌مون اجرا می‌کردیم رو می‌تونیم با استفاده از sessionها توی جاوا بنویسیم:



Hibernate، از طریق کلاسی به اسم SessionFactory به ما اجازهٔ ساخت Session جدید رو می‌ده. ممکنه ما توی برنامه‌مون Sessionهای مختلفی برای کارهای متفاوت‌مون داشته باشیم، مثلا یه Session برای ذخیرهٔ Taskهامون باشه، یه Session برای حذف Taskهامون، و یه Session هم برای دیدن کل Taskهای دیتابیس. برای این کارها، می‌تونیم توی برنامه‌مون سه Session مختلف از SessionFactory بگیریم و توی هر کدوم یه کار متفاوت انجام بدیم:



امیدوارم که متوجه کاربرد Sessionها شده باشین. اما اگر نشدین، نگران نباشید. وقتی کدش رو ببینید خیلی راحت‌تر می‌فهمید که چطور از اون‌ها استفاده می‌کنیم.

ایجاد SessionFactory

به کلاس Main جاواتون برگردین. اولین چیزی که نیاز داریم، یه SessionFactoryئه که بتونیم ازش Sessionهای جدید بگیریم. برای این کار، یه فیلد static از جنس SessionFactory به Main اضافه کنید:

```
public class Main {
    private static SessionFactory sessionFactory;

    // code code code
}
```

یه متد به اسم setUpSessionFactory() به Main اضافه می‌کنیم که توی اون sessionFactory رو مقداردهی کنیم. SessionFactoryمون باید اطلاعات دیتابیسی که بهش وصل می‌شیم رو داشته باشه، و ما این اطلاعات رو توی hibernate.cfg.xml مشخص کردیم. به همین خاطر، برای مقداردهی اون از کد زیر استفاده می‌کنیم:

```
private static void setUpSessionFactory() {
    sessionFactory = new Configuration()
            .configure("hibernate.cfg.xml")
            .buildSessionFactory();
}
```

لازم نیست کد بالا رو خیلی دقیق بفهمین. صرفا بدونید که این تکه کد، فایل hibernate.cfg.xml رو می‌بینه، از توی اون اطلاعات دیتابیس رو می‌خونه، و یه SessionFactory برای ارتباط با اون دیتابیس بهمون می‌ده.

همچنین، sessionFactory باید بعد از اتمام استفاده close بشه.[7] برای این کار، متد closeSessionFactory() رو هم به Main اضافه می‌کنیم:

```
private static void closeSessionFactory() {
    sessionFactory.close();
}
```

حالا این متدها رو در ابتدا و انتهای متد main صدا می‌زنیم:

```
public static void main(String[] args) {
    setUpSessionFactory();

    // our code is here

    closeSessionFactory();
}
```

بین این دو method call، می‌تونیم از SessionFactoryمون استفاده کنیم. توی کد زیر، یه Session جدید از sessionFactory می‌گیریم. این Session هم باید بعد از اتمام استفاده، close بشه:

```
public static void main(String[] args) {
    setUpSessionFactory();

    Session session = sessionFactory.openSession();

    // the code that uses the session goes here
```

[7] ما عادت کردیم که بعد از اتمام کارمون با یه object، Garbage Collector جاوا اون رو از موری برامون حذف کنه. اما بعضی از زبان‌ها، مثل C، Garbage Collector خاصی ندارن. توی این زبان‌ها، اگر بخشی از موری رو اشغال کردین، بعد از اتمام کار باهاشون، خودتون هم باید اون بخش رو خالی کنید.
نداشتن Garbage Collector باعث می‌شه که این زبان‌ها سریع‌تر از باقی زبان‌ها باشن؛ ولی از طرفی، اگر یادتون بره که بخشی از موری رو آزاد کنید، ممکنه برنامه‌تون کل موری رو اشغال کنه و به اصطلاح، Memory Leak داشته باشه. راجع به این مشکل می‌تونید توی این صفحهٔ ویکی‌پدیا بیشتر بخونین.
ولی حتی Garbage Collector هم نمی‌تونه تمام resourceهایی که بعضی از objectهامون گرفتن رو آزاد کنه. به عنوان مثال، آبجکت‌های file از این دسته‌ن. کلا Garbage Collector نمی‌تونه آبجکت‌هایی که با هر نوع I/O کار می‌کنن رو مدیریت کنه. این objectها باید بعد از استفاده‌شون، close بشن. آبجکت‌های SessionFactory و Session هم، هر دو از این دسته‌ن. این کلاس‌ها همگی اینترفیس Closeable رو پیاده‌سازی می‌کنن. برای اطلاعات بیشتر راجع به Closeable، این صفحهٔ GeeksForGeeks رو بخونید.

```
    session.close();

    closeSessionFactory();
}
```

حالا که Sessionمون رو درست کردیم، می‌تونیم ازش استفاده کنیم. برای استفاده از دیتابیس‌مون، از کد زیر استفاده می‌کنیم:

```
public static void main(String[] args) {
    setUpSessionFactory();

    Session session = sessionFactory.openSession();

    try {
        // We're going to use the session
        Transaction tx = session.beginTransaction();

        // We're using the database...
        // CODE CODE CODE

        // We're done using the session. Save everything to the database.
        tx.commit();
    } catch (Exception e) {
        System.out.println("Exception in the database: " +
                e.getMessage());
    }

    session.close();

    closeSessionFactory();
}
```

همون‌طور که می‌بینید، برای استفاده از sessionمون اول یه transaction شروع می‌کنیم. هر transaction، یک بلوک از کدهاییه که توشون از دیتابیس می‌خونیم یا اون رو تغییر می‌دیم. نهایتا، با صدا زدن متد commit، تغییراتمون رو توی دیتابیس ذخیره می‌کنیم.

ممکنه تغییرات دیتابیسی‌مون با خطا مواجه بشن، برای همین هم کل کدهای مرتبط با transactionمون رو، از شروع تا پایان، توی یه try-catch گذاشتیم که این خطاها منجر به crash کردن برنامه‌مون نشن.[8]

[8] try-catchای که نوشتیم، بهترین کد Hibernate جهان نیست! بهتره که session.close() رو توی بخش finallyی tryمون صدا می‌زدیم و همچنین، بهتر بود که توی catchمون، tx.rollback() هم صدا کنیم. توی صفحات بعدی، ما به کل راه راحت‌تری برای استفاده از Sessionها بهتون ارائه می‌دیم، و نمی‌خواستیم این‌جا خیلی گیج‌تون کنیم. ولی اگر دوست دارین، کد درست‌تر این بخش رو از این‌جا بخونید.

همه چی برای تغییر دیتابیس آماده‌ست. اولین تغییراتمون رو، با ذخیرهٔ یک Task شروع می‌کنیم.

ذخیره کردن در دیتابیس

بیاین یه Task بسازیم و اون رو توی جدول tasks ذخیره کنیم. با توجه به annotationهایی که برای کلاس Task نوشتیم، Hibernate ارتباط بین Task و tasks رو به طور کامل متوجه شده. پس ذخیرهٔ یه تسک جدید نباید کار سختی باشه.

برای ذخیرهٔ یک object توی دیتابیس، از متد persist استفاده می‌کنیم. کد زیر رو بخونید:

```
public static void main(String[] args) {
    setUpSessionFactory();

    Session session = sessionFactory.openSession();

    try {
        Transaction tx = session.beginTransaction();

        Task t = new Task("Read 'Hibernate' Document", LocalDate.now());
        session.persist(t);

        tx.commit();
    } catch (Exception e) {
        System.out.println("Exception in the database: " +
                e.getMessage());
    }

    session.close();

    closeSessionFactory();
}
```

کد بالا، یه Task جدید ایجاد می‌کنه و اون رو توی دیتابیس ذخیره می‌کنه. بیاین اجراش کنیم:

```
Run  Main
    Isolation level: undefined/unknown
    Minimum pool size: 1
    Maximum pool size: 20
[INFO] HHH000489: No JTA platform available (set 'hibernate.transaction.jta.platform' to enable JTA platform integration)
[Hibernate]
    insert
    into
        tasks
        (due_date, name)
    values
        (?, ?)
[TRACE] binding parameter (1:DATE) <- [2025-05-24]
[TRACE] binding parameter (2:VARCHAR) <- [Read 'Hibernate' Document]
```

طبق معمول، Hibernate یه سری log بهمون می‌ده. لاگ آخر رو می‌بینید؟ این لاگ منظورمه:

```
[Hibernate]
    insert
    into
        tasks
        (due_date, name)
    values
        (?, ?)
[TRACE] binding parameter (1:DATE) <- [2025-05-24]
[TRACE] binding parameter (2:VARCHAR) <- [Read 'Hibernate' Document]
```

اگر شبیه کوئری SQLئه، به خاطر اینه که کوئری SQLئه! وقتی session.pertsist(t) اجرا شد، Hibernate این کوئری رو توی دیتابیس‌تون اجرا کرد. این کوئری، یه insert intoی ساده‌ست که توی بخش values، به جای مقادیری ستون‌های due_date و name، دوتا علامت سوال داره. Hibernate قبل از اجرای این کوئری، علامت‌های سوال رو با پارامترهایی که توی لاگ‌های TRACE اومدن پر می‌کنه. این یعنی کوئری نهاییِ اجرا شده روی دیتابیس، همچین چیزیه:

```
insert
into
    tasks
    (due_date, name)
values
    ('2025-05-24', 'Read \'Hibernate\' Document')
```

این که Hibernate کوئری‌هایی دیتابیسی‌ش رو به ما نشون می‌ده اتفاقی نیست. اگر یادتون باشه، توی فایل hibernate.cfg.xml ما Hibernate رو به شکل زیر تنظیم کرده بودیم:

```
<!-- Hibernate settings -->
<property name="hibernate.show_sql">true</property>
<property name="hibernate.format_sql">true</property>
<property name="hibernate.highlight_sql">true</property>
```

این تنظیمات باعث می‌شن که Hibernate تمام کوئری‌هایی که روی دیتابیس می‌زنه رو برای ما چاپ کنه. اگر حذف‌شون کنید و دوباره برنامه‌تون رو اجرا کنید، می‌بینید که خبری از این کوئری‌ها نیست.

اگر الآن برین توی DataGrip و روی جدول tasks، select بزنید خروجی زیر رو می‌بینید:

	id	name	due_date
1	1	Read 'Hibernate' Document	2025-05-24

می‌بینید؟ کار کرد! برنامهٔ جاواتون الآن کاملا به دیتابیس متصله و می‌تونه توش تغییر ایجاد کنه!

حالا به کد جاواتون برگردین. می‌خوایم فیلد idی تسک‌مون رو، قبل و بعد از ذخیره شدن اون توی دیتابیس بررسی کنیم. کد زیر رو به جای کد قبلی بذارین:

```
Task t = new Task("Read 'Hibernate' Document", LocalDate.now());
System.err.println(t);

session.persist(t);
System.err.println(t);
```

ما به جای System.out، توی System.err تسک‌مون رو چاپ کردیم که رنگ قرمز خروجی، اون رو از logهامون متمایز کنه. این کد رو اجرا کنید و خروجی‌ش رو ببینید:

```
Task{id=null, name='Read 'Hibernate' Document', dueDate=2025-05-25}
[Hibernate]
    insert
    into
        tasks
        (due_date, name)
    values
        (?, ?)
[TRACE] binding parameter (1:DATE) <- [2025-05-24]
[TRACE] binding parameter (2:VARCHAR) <- [Read 'Hibernate' Document]
Task{id=2, name='Read 'Hibernate' Document', dueDate=2025-05-25}
```

به فیلد id، قبل و بعد از ذخیرهٔ تسک‌مون توی دیتابیس دقت کنید. قبل از ذخیره، مقدار اون nullئه. ولی بعد از ذخیره، مقدارش به ۲ آپدیت می‌شه! همون‌طور که قبلا گفتیم، توی جدول tasks تعیین id بر عهدهٔ دیتابیس بوده و Hibernate، نه تنها تسک‌مون رو ذخیره کرده، که id اون توی دیتابیس هم بهمون داده! اگر دوباره به جدول tasks نگاه کنید، می‌تونید این تسک جدید رو توش ببینید:

	id	name	due_date
1	1	Read 'Hibernate' Document	2025-05-24
2	2	Read 'Hibernate' Document	2025-05-24

یادتونه که ما idی کلاس Task رو به جای int، Integer تعریف کردیم؟ بیاین اون رو int بذاریم و ببینیم چی می‌شه. این فیلد رو توی کلاس Task به شکل زیر تغییر بدین:

```
@Id
@GeneratedValue(strategy = GenerationType.IDENTITY)
private int id;
```

حالا، دوباره کدی که زدیم رو اجرا کنید. خروجی به شکل زیره:

```
Task{id=0, name='Read 'Hibernate' Document', dueDate=2025-05-25}
[Hibernate]
    insert
    into
        tasks
        (due_date, name)
    values
        (?, ?)
[TRACE] binding parameter (1:DATE) <- [2025-05-24]
```

```
[TRACE] binding parameter (2:VARCHAR) <- [Read 'Hibernate' Document]
Task{id=3, name='Read 'Hibernate' Document', dueDate=2025-05-25}
```

کدمون باز هم درست کار می‌کنه. ولی مقدار اولیهٔ id قبل از ذخیره، به جای null، صفره.

ولی Integer گذاشتن id به جای int، یه خوبی داره. فرض کنید فیلد id رو int نگه داریم. بعد از یه مدت، کد جاوامون خیلی خیلی بزرگ‌تر شده و یه جایی توی برنامه‌مون، به تسک زیر برخوردیم:

Task

id	0
name	AP Project
dueDate	2025 - 06 - 30

دقت کنید که id این Task صفره. دو حالت داریم:

1. این Task، توی دیتابیس ذخیره نشده. به خاطر همین، idش صفره و بعد از ذخیره، تغییر می‌کنه.
2. این Task، توی دیتابیس ذخیره شده و دیتابیس بوده که بهش idی صفر داده. الآن هم از دیتابیس خوندیمش[9].

هر دو سناریو می‌تونه درست باشه و شما نمی‌تونید بین این دو حالت تمایز قائل بشین. ولی فرض کنید که Task idی به جای int، Integer بود. در این صورت، شما می‌دونستید که Task بالا قطعا توی دیتابیس ذخیره شده، چون idش null نیست. در مقابل اون، می‌دونید که Task زیر قطعا توی دیتابیس ذخیره نشده:

Task

id	null
name	AP Project
dueDate	2025 - 06 - 30

[9] در ادامه یاد می‌گیریم که چطور می‌تونیم با Hibernate، از دیتابیس اطلاعات مختلف رو بخونیم.

این تمایز کوچیک، خیلی جاها به ما کمک می‌کنه. به خاطر همین توصیه می‌شه که فیلد primary keyتون رو از جنس reference typeها بذارین.[10] برگردین و جنس فیلد id رو Integer کنید:

```
@Id
@GeneratedValue(strategy = GenerationType.IDENTITY)
private Integer id;
```

حواستون باشه که شما می‌تونید باقی فیلدهای موجودیت‌های دیتابیس‌تون رو primitive type بذارین. فقط طبیعتا ستون نظیر اون‌ها توی دیتابیس هم باید not null باشه.

خوندن رکوردهای دیتابیس

این بخش، نسبتا راحته. sessionها یه متد به اسم get دارن که با استفاده از اون، می‌تونید رکوردهای دیتابیس‌تون رو بخونید. کد زیر رو ببینید:

```
public static void main(String[] args) {
    setUpSessionFactory();

    Session session = sessionFactory.openSession();

    try {
        Transaction tx = session.beginTransaction();

        Task t = session.get(Task.class, 1);
        System.err.println("db: " + t);

        tx.commit();
    } catch (Exception e) {
        System.out.println("Exception in the database: " +
                e.getMessage());
    }

    session.close();

    closeSessionFactory();
}
```

بخش جدید این کد، دو خطِ زیره. توی این دو خط، ما به Hibernate می‌گیم که از جدول tasks، رکوردی که idش یکه رو پیدا کنه و به ما بده. بعدش هم اون رو چاپ می‌کنیم:

```
Task t = session.get(Task.class, 1);
System.err.println("task from db: " + t);
```

[10] منبع این توصیه، این‌جاست.

به ورودی‌های متد get توجه کنید. دومین ورودی اون، idی رکوردیه که می‌خوایم از دیتابیس بخونیم. همچنین، ما باید به Hibernate بگیم که این رکورد رو از کدوم جدول بخونه. اگر یادتون باشه، ما اسم جدول taskها رو توی یه annotation، بالای کلاس Task، نوشته بودیم:

```
@Entity
@Table(name = "tasks")
public class Task {

    // CODE CODE CODE

}
```

با ورودی دادنِ Task.class به متد get، Hibernate می‌فهمه که باید توی جدول tasks دنبال رکوردِ دلخواه ما بگرده. ورودی دادن Task.class یه خوبیِ دیگه هم داره. اگر به declaration متد get برین، می‌بینید که این متد جنریکه:

پس توی کد زیر، چطور جاوا می‌فهمه که خروجی این متد، Taskئه؟ کجای این call شما جنس T رو مشخص کردین؟

```
session.get(Task.class, 1);
```

آبجکت Task.class، از جنس Class<Task>ئه، و ورودی اول get هم از جنس Class<T>ئه. از روی همین ورودی، جاوا می‌فهمه که توی این متد جنریک، T همون Taskئه و در نتیجه، خروجیِ get هم یه Taskئه.

کد رو اجرا کنید و نتیجه رو ببینید:

```
db: Task{id=1, name='Read 'Hibernate' Document', dueDate=2025-05-24}
```

تونستیم با موفقیت، تسک ۱ رو از دیتابیس بخونیم. یه نگاه به لاگ‌هاتون بندازین تا ببینید که Hibernate برای خوندن این Task، چه کوئری‌ای روی دیتابیس زد:

```
[Hibernate]
    select
        t1_0.id,
        t1_0.due_date,
        t1_0.name
    from
        tasks t1_0
    where
        t1_0.id=?
[TRACE] binding parameter (1:INTEGER) <- [1]
```

می‌بینید؟ Hibernate فقط یه select و where ساده توی دیتابیس زد. اگر پارامتری که توی بخش TRACE اومده رو توی کوئریِ لاگ شده بذاریم، می‌بینیم که کوئری نهایی Hibernate روی دیتابیس دقیقا چی بوده:

```
select
    t1_0.id,
    t1_0.due_date,
    t1_0.name
from
    tasks t1_0
where
    t1_0.id=1
```

خیلی هم خوب، حالا وقتشه که یاد بگیریم رکوردهای دیتابیس‌مون رو حذف کنیم.

حذف رکوردها از دیتابیس

حذف رکوردها، حتی از get کردنشون هم راحت‌تره. کد زیر رو ببینید:

```
public static void main(String[] args) {
    setUpSessionFactory();

    Session session = sessionFactory.openSession();

    try {
        Transaction tx = session.beginTransaction();

        Task t = session.get(Task.class, 1);
        System.err.println("task from the db: " + t);

        session.remove(t);

        tx.commit();
    } catch (Exception e) {
        System.out.println("Exception in the database: " +
                e.getMessage());
    }

    session.close();
```

```
    closeSessionFactory();
}
```

تکهٔ جدید این کد، خط زیره. با دادن آبجکت t به متد remove، می‌تونیم اون رو از دیتابیس پاک کنیم:

```
session.remove(t);
```

این کد رو اجرا کنید. بعدش به DataGrip برین و روی جدول tasks، select بزنید. همون‌طور که می‌بینید، تسکِ ۱ از دیتابیس حذف شده:

	id	name	due_date
1	2	Read 'Hibernate' Document	2025-05-24
2	3	Read 'Hibernate' Document	2025-05-25

بیاین کوئری‌هایی که Hibernate روی دیتابیس زد رو ببینیم. به لاگ‌های Hibernate توجه کنید:

```
[Hibernate]
    select
        t1_0.id,
        t1_0.due_date,
        t1_0.name
    from
        tasks t1_0
    where
        t1_0.id=?
[TRACE] binding parameter (1:INTEGER) <- [1]

[Hibernate]
    delete
    from
        tasks
    where
        id=?
[TRACE] binding parameter (1:INTEGER) <- [1]
```

Hibernate، این بار دو کوئری روی دیتابیس زد. کوئری اول برای get کردن تسک‌مونه و اون رو توی بخش قبلی دیده بودیم. اما کوئری دومه که رکورد ما رو از دیتابیس پاک می‌کنه. این کوئری هم یک delete و where ساده‌ست. اگر پارامتر لاگ TRACE رو توی کوئری بذارین، می‌بینید که کوئریِ نهاییِ اجرا شده روی دیتابیس به این شکل بوده:

```
delete
from
    tasks
where
    id=1
```

می‌تونیم این‌جا متوقف شیم. ولی یه چیزی راجع به برنامهٔ بالا من رو اذیت می‌کنه. چرا برای حذف تسک ۱، مجبور شدیم دوتا کوئری روی دیتابیس بزنیم؟ توی کوئریِ اول، ما تسک ۱ رو get کردیم:

```
Task t = session.get(Task.class, 1);
```

و توی کوئری دوم، این تسک رو remove کردیم:

```
session.remove(t);
```

برای یه لحظه فرض کنید که Session، یه overload از متد remove داشت که ورودی‌هاش شبیه متد get بود:

```
session.remove(Task.class, 1);
```

این‌طوری، لازم نبود با صدا زدن متد get، یه select و where روی دیتابیس بزنیم و بعدش با یه delete و where تسک‌مون رو پاک کنیم؛ به جاش می‌تونستیم همهٔ کارهامون رو توی یه کوئری انجام بدیم.

شاید این موضوع، براتون مسئلهٔ مهمی به نظر نرسه. ولی کوئری‌های Hibernate روی دیتابیس عملیات‌های نسبتا پرهزینه‌ای هستن. برای اجرای هر کوئری، Hibernate باید به دیتابیس وصل بشه، کوئری رو بهش ارسال کنه، منتظرِ جواب دیتابیس بمونه و نهایتا اون جواب رو به ما بده. این کارها، همگی شامل عملیات‌های کند I/O هستن. به خاطر همین، ما تلاش می‌کنیم جوری برنامه‌نویسی کنیم که Hibernate حداقل کوئری ممکن رو روی دیتابیس بزنه.

متاسفانه، session اون overload دوست‌داشتنی متد remove که بالاتر نوشتیم رو نداره. ولی این مسئله هم بدون راه حل نیست. جلوتر اون رو بررسی می‌کنیم و بهتون توضیح می‌دیم که چطور می‌تونید با فقط یه کوئری رکوردهای دیتابیس رو حذف کنید.

متدهای inTransaction و fromTransaction

تا این‌جا، کدهای Hibernate ما همچین ظاهری داشتن:

```
Session session = sessionFactory.openSession();

try {
    Transaction tx = session.beginTransaction();

    // our actual code is here

    tx.commit();
} catch (Exception e) {
    System.out.println("Exception in the database: " +
```

```
            e.getMessage());
}

session.close();
```

برای هر عملیات دیتابیسی، ما یه session و transaction باز می‌کردیم، با دیتابیس کار می‌کردیم، بعدش transaction رو کامیت می‌کردیم و نهایتا هم session رو می‌بستیم. وقتی برنامه‌مون بزرگ‌تر می‌شه و با دیتابیس بیشتر کار می‌کنیم، باید دائما sessionها و transactionهای مختلف رو باز و بسته کنیم، و به همین خاطر این تکه کد دائما تکرار می‌شه.[11]

بیاین این تکه کد رو به یه متد جداگانه منتقل کنیم. متد runInTransaction رو به Main اضافه کنید:

```
private static void runInTransaction(Consumer<Session> consumer) {
    Session session = sessionFactory.openSession();

    try {
        Transaction tx = session.beginTransaction();

        consumer.accept(session);

        tx.commit();
    } catch (Exception e) {
        System.out.println("Exception in the database: " +
                e.getMessage());
    }

    session.close();
}
```

این متد، یه consumer می‌گیره و بعد از ایجاد session و transaction، اون رو اجرا می‌کنه. نهایتا هم session و transaction رو می‌بنده. حالا می‌تونیم این متد رو با consumerهای مختلف صدا کنیم و دیگه لزومی به تکرار کد قدیمی‌مون نیست. به عنوان مثال، تکه کد زیر معادل کدیه که توی بخش «خوندن رکوردهای دیتابیس» نوشته بودیم:

```
public static void main(String[] args) {
    setUpSessionFactory();

    runInTransaction(session -> {
        Task t = new Task("AP Project", LocalDate.now());
        session.persist(t);

        System.out.println("Successfully saved task.");
    });
```

[11] علاوه بر این، همون‌طور که توی پانویس‌های قبلی گفتیم، این کد هم کاملا درست نیست و بستن sessionها و rollback کردن transactionها رو به درستی انجام نمی‌ده. نوشتن کدی که به درستی این کارها رو بکنه یه خورده قلق داره.

```
    closeSessionFactory();
}
```

کدمون کوتاه‌تر و تمیزتر شده، مگه نه؟ اگر اون رو اجرا کنید، می‌بینید که تسک AP Project توی دیتابیس ذخیره می‌شه.

خوبیِ متد runInTransaction اینه که می‌تونه هر کدی رو توی دیتابیس اجرا کنه. مثلا تکه کد زیر معادل کدیه که قبلا برای حذف تسک‌هامون نوشته بودیم:

```
public static void main(String[] args) {
    setUpSessionFactory();

    runInTransaction(session -> {
        Task t = session.get(Task.class, 3);
        session.remove(t);
    });

    closeSessionFactory();
}
```

و اگر اون رو اجرا کنید، می‌بینید که به درستی تسک ۳ رو پاک می‌کنه.

خوشبختانه، کلاس SessionFactory هم یه متد شبیه به متد runInTransaction ما داره. اسمش inTransactionئه و به شکل مشابه استفاده می‌شه. کد زیر، معادل کدیه که بالاتر برای ذخیرهٔ تسک AP Project نوشته بودیم:

```
public static void main(String[] args) {
    setUpSessionFactory();

    sessionFactory.inTransaction(session -> {
        Task t = new Task("AP Project", LocalDate.now());
        session.persist(t);
    });

    closeSessionFactory();
}
```

می‌تونید متد runInTransaction رو پاک کنید. دیگه نیازی بهش نداریم!

گاهی وقت‌ها، می‌خواید از دیتابیس یه چیزی رو بخونید و توی برنامه‌تون استفاده کنید. برای این کار از متد مشابه‌ای به اسم fromTransaction استفاده می‌شه. این متد به جای Consumer، یه Function ازتون ورودی می‌گیره و خروجی اون رو بهتون برمی‌گردونه. به عنوان مثال، کد زیر تسک ۲ رو از دیتابیس می‌خونه و اون رو بهتون خروجی می‌ده:

```
public static void main(String[] args) {
    setUpSessionFactory();

    Task t = sessionFactory.fromTransaction(session -> {
        return session.get(Task.class, 2);
    });

    System.err.println(t);

    closeSessionFactory();
}
```

اگر lambda expressionها رو به خوبی یادتون باشه، می‌دونید که این کد رو می‌شه کوتاه‌تر هم نوشت:

```
public static void main(String[] args) {
    setUpSessionFactory();

    Task t = sessionFactory.fromTransaction(session ->
            session.get(Task.class, 2));

    System.err.println(t);

    closeSessionFactory();
}
```

متدهای inTransaction و fromTransaction، کارهای دیتابیسی ما رو خیلی خیلی راحت‌تر می‌کنن و با استفاده از اون‌ها، دیگه لازم نیست خودمون رو درگیر باز کردن و بستن sessionها و transactionها کنیم.

متدهای createNativeQuery و createNativeMutationQuery

تا به این‌جای کار، ما متدهای persist، get و remove رو دیدیم و فهمیدیم چطور می‌شه با اون‌ها کار کرد. ولی این متدها، در عین قدرمندی‌شون، نمی‌تونن تمام عملیات‌های دیتابیسی ما رو انجام بدن. مثلا، ما هنوز نمی‌دونیم که چطور می‌شه تمام رکوردهای یه جدول رو خوند.

توی همهٔ این متدها، Hibernate بود که کوئری‌های SQL رو generate می‌کرد. مثلا وقتی ما کد زیر رو می‌نوشتیم، این وظیفهٔ Hibernate بود که کوئریِ معادل SQLش رو تولید کنه:

```
session.get(Task.class, 2);
```

کوئریِ نهایی Hibernate به شکل زیر بود:

```
select
    t1_0.id,
    t1_0.due_date,
    t1_0.name
from
```

```
    tasks t1_0
where
    t1_0.id=3
```

متدهای createNativeQuery و createNativeMutationQuery، به ما اجازه می‌دن که کوئری‌های دست‌ساز خودمون رو توی دیتابیس اجرا کنیم. مثال زیر رو ببینید:

```
public static void main(String[] args) {
    setUpSessionFactory();

    List<Task> allTasks = sessionFactory.fromTransaction(session ->
            session.createNativeQuery("select * from tasks", Task.class)
                    .getResultList());


    System.out.println("All tasks: ");

    for (Task t : allTasks) {
        System.out.println(t);
    }

    closeSessionFactory();
}
```

توی این کد، تمام تسک‌های دیتابیس‌مون رو خوندیم و اون‌ها رو چاپ کردیم. اگر به خروجی برنامه‌تون نگاه کنید، همچین چیزی می‌بینید:

```
[Hibernate]
    select
        *
    from
        tasks

All tasks:
Task{id=2, name='Read 'Hibernate' Document', dueDate=2025-05-24}
Task{id=3, name='Read 'Hibernate' Document', dueDate=2025-05-25}
```

توی متد createNativeQuery، ما کوئری دلخواه خودمون رو به شکل یه String به Hibernate دادیم تا توی دیتابیس اجراش کنه:

```
session.createNativeQuery("select * from tasks", Task.class).getResultList()
```

با توجه به لاگ Hibernate هم می‌تونید ببینید که دقیقا همین کوئری توی دیتابیس اجرا شده:

```
[Hibernate]
    select
        *
    from
        tasks
```

علاوه بر این، با ورودی دادن Task.class مشخص کردیم که خروجی این کوئری، از جنس Taskئه. نهایتا هم با صدا زدن متد getResultList از Hibernate خواستیم که بعد از اجرای کوئریِ بالا، نتیجه رو به شکل یه List‹Task› به ما برگردونه.

با استفاده از متد createNativeQuery، ما می‌تونیم هر کوئریِ selectای روی دیتابیس بزنیم. توی مثال زیر، با استفاده از این متد یه کوئری select و where زدیم:

```
List<Task> tasks = sessionFactory.fromTransaction(session ->
        session.createNativeQuery("select * from tasks " +
                        "where id = 3", Task.class)
                .getResultList());

System.out.println("Tasks: ");
for (Task t : tasks) {
    System.out.println(t);
}
```

خروجی این کد به شکل زیره:

```
[Hibernate]
    select
        *
    from
        tasks
    where
        id = 3

All tasks:
Task{id=3, name='Read 'Hibernate' Document', dueDate=2025-05-25}
```

کد بالا رو می‌شد به شکل دیگه‌ای هم نوشت:

```
List<Task> allTasks = sessionFactory.fromTransaction(session ->
        session.createNativeQuery("select * from tasks " +
                        "where id = :given_id", Task.class)
                .setParameter("given_id", 3)
                .getResultList());

System.out.println("All tasks: ");
for (Task t : allTasks) {
    System.out.println(t);
}
```

توی کد بالا، بر خلاف کد قبلی‌مون، عدد ۳ رو به شکل مستقیم توی متنِ کوئری نیاوردیم. در عوض به جای اون، پارامترِ given_id رو مشخص کردیم. وقتی توی کوئری‌های SQLای که به Hibernate می‌دیم، قبل عبارتی از ":" استفاده می‌کنیم، به Hibernate می‌گیم که «این تیکهٔ کوئری‌مون، یه پارامتره و مقدارش رو بعدا مشخص می‌کنیم». به خاطر همین، قبل از این که متد getResultList رو صدا بزنیم،

با استفاده از متد setParameter به Hibernate گفتیم که «پارامترِ given_id، ۳ هست». خروجی این کد هم، مشابه کد قبلیه:

```
[Hibernate]
    select
        *
    from
        tasks
    where
        id = ?
[TRACE] binding parameter (1:INTEGER) <- [3]

All tasks:
Task{id=3, name='Read 'Hibernate' Document', dueDate=2025-05-25}
```

تنها تفاوت کد بالا با کد قبلی‌مون، اینه که Hibernate هم توی کوئری لاگ شده، به جای given_id یه علامت سوال چاپ کرده. توی لاگِ TRACE می‌تونید ببینید که توی کوئری نهایی، به جای این علامت سوال مقدار ۳ داده شده.

با استفاده از createNativeQuery، ما می‌تونیم اطلاعات مختلف رو از دیتابیس بخونیم. ولی چطور می‌تونیم رکوردهای جدید به دیتابیس اضافه کنیم، اون‌ها رو تغییر بدیم یا به کل حذف کنیم؟ اجرای کوئری‌هایی که منجر به تغییرات دیتابیسی می‌شن با createNativeMutationQueryئه. کد زیر رو ببینید:

```
Scanner sc = new Scanner(System.in);

System.out.println("New task's name: ");
String taskName = sc.nextLine();

System.out.println("New task's due date: ");
String taskDueDate = sc.nextLine();

sessionFactory.inTransaction(session -> {
    session.createNativeMutationQuery("insert into tasks(name, due_date) " +
            "values (:name, :dueDate)")
            .setParameter("name", taskName)
            .setParameter("dueDate", taskDueDate)
            .executeUpdate();
});
```

توی کد بالا، یه task جدید بر اساس ورودی‌های کاربر توی دیتابیس ذخیره کردیم. بیاین متد createNativeMutationQuery رو بررسی کنیم:

```
session.createNativeMutationQuery("insert into tasks(name, due_date) " +
        "values (:name, :dueDate)")
        .setParameter("name", taskName)
```

```
        .setParameter("dueDate", taskDueDate)
        .executeUpdate();
```

کوئری SQLمون، یه insert intoی ساده‌ست که دو پارامتر name و dueDate توی اون تعریف شده. برای مقداردهی این دو پارامتر، ما دو بار متد setParameter رو صدا زدیم و بعد از اون، با استفاده از متد executeUpdate کوئری نهایی رو توی دیتابیس اجرا کردیم. این برنامه رو اجرا کنید و ورودی‌های زیر رو به اون بدین:

```
New task's name:
Read "The Pragmatic Programmer"

New task's due date:
2025-12-02
```

بعد از دادن ورودی‌ها، می‌بینید که Hibernate لاگ زیر رو براتون چاپ می‌کنه:

```
[Hibernate]
    insert
    into
        tasks
        (name, due_date)
    values
        (?, ?)
[TRACE] binding parameter (1:VARCHAR) <- [Read "The Pragmatic Programmer"]
[TRACE] binding parameter (2:VARCHAR) <- [2025-12-02]
```

اگر الآن به جدول tasks نگاه کنید، می‌بینید که تسک جدید کاربر با موفقیت به دیتابیس اضافه شده:

	id	name	due_date
1	2	Read 'Hibernate' Document	2025-05-24
2	3	Read 'Hibernate' Document	2025-05-25
3	4	Read "The Pragmatic Programmer"	2025-12-02

ما می‌تونیم کوئری‌های update و delete هم با استفاده از createNativeMutationQuery اجرا کنیم. به عنوان مثال، کد زیر تمامی تسک‌هایی که ورودی کاربر بخشی از اسم اون‌هاست رو پاک می‌کنه:

```
Scanner sc = new Scanner(System.in);

System.out.println("task's name: ");
String taskName = sc.nextLine();

sessionFactory.inTransaction(session -> {
    session.createNativeMutationQuery("delete from tasks " +
                    "where name like :taskName")
            .setParameter("taskName", "%" + taskName + "%")
            .executeUpdate();
});
```

همون‌طور که می‌بینید، کوئری‌مون این بار یه deleteئه که پارامتر taskName توی اون پدیدار شده. از اون‌جایی که می‌خواستیم تمام تسک‌هایی که ورودی کاربر بخشی از اسمشونه رو پاک کنیم، موقع مقداردهی این پارامتر قبل و بعد اون % گذاشتیم. نهایتا هم با متد executeUpdate کوئری‌مون رو توی دیتابیس اجرا کردیم.

این برنامه رو اجرا کنید و ورودی زیر رو بهش بدین:

```
task's name:
Hibernate
```

می‌بینید که Hibernate، کوئری زیر رو براتون لاگ می‌کنه:

```
[Hibernate]
    delete
    from
        tasks
    where
        name like ?
[TRACE] binding parameter (1:VARCHAR) <- [%Hibernate%]
```

و اگر الآن جدول tasks رو ببینید، تسک‌هایی که توی اسم‌شون Hibernate داشتن پاک شدن:

	id	name	due_date
1	4	Read "The Pragmatic Programmer"	2025-12-02

متدهای createNativeQuery و createNativeMutationQuery، قدرت اجرای هر کوئری SQLای که دلتون می‌خواد رو بهتون می‌دن.[12]

[12] Hibernate متدهای مشابه دیگه‌ای هم در اختیارتون می‌ذاره که با استفاده از اون‌ها هم می‌تونید کنترل بیشتری روی کوئری اجرا شده توی دیتابیس داشته باشین. متدهایی مثل createSelectionQuery و createMutationQuery به شما اجازه می‌دن که توی نوشتن کوئری‌هاتون از Hibernate کمک بیشتری بگیرین و استفاده از اون‌ها، نسبت به createNativeQuery و createNativeMutationQuery بهتره. ولی این داکیومنت بدون معرفی این متدها هم طولانیه و به خاطر همین موضوع، این‌جا بهشون اشاره‌ای نکردیم. اگر دوست داشتین، حتما با خوندن داک رسمی Hibernate با این متدها بیشتر آشنا بشین.

Foreign keyها و Lazy Fetching

به دایرکتوری src/main/sql برین و فایل steps.sql رو درست کنید. توی این فایل، کوئریِ ایجاد جدول steps رو بنویسید و اون رو اجرا کنید:

```
use learn_hibernate_db;

create table steps(
    id int primary key auto_increment,
    task_id int not null,
    name nvarchar(255) not null,
    is_completed boolean not null default false,

    foreign key (task_id) references tasks(id)
);
```

حالا به کد جاواتون برگردین و کلاس Step رو توی پکیج aut.ap.model ایجاد کنید:

```
package aut.ap.model;


public class Step {
    private Integer id;

    private Task task;

    private String name;

    private boolean isCompleted;

}
```

قراره که از این کلاس، برای ارتباط با جدول steps توی برنامه‌مون استفاده کنیم. نکتهٔ جالبی که در مورد این کلاس هست، اینه که فیلد نظیر ستون task_id، یعنی فیلد task، از جنس int نیست و از جنس خود Taskئه! در ادامه، می‌بینید که Hibernate با ستون‌های foreign key چطور برخورد می‌کنه و علت Task بودن این فیلد براتون شفاف می‌شه.

به کلاس Step، کانستراکتورهای لازم رو اضافه کنید. اگر یادتون باشه، هر کلاسی که Hibernate باهاش کار می‌کنه باید یک کانستراکتور خالی داشته باشه. پس دو کانستراکتور زیر رو به کلاس Step اضافه کنید:

```
public Step() {
}

public Step(Task task, String name) {
    this.task = task;
```

```
    this.name = name;
    this.isCompleted = false;
}
```

حالا، getter و setterهای لازم هم به کلاس Step اضافه کنید:

```
public Integer getId() {
    return id;
}

public Task getTask() {
    return task;
}

public void setTask(Task task) {
    this.task = task;
}

public String getName() {
    return name;
}

public void setName(String name) {
    this.name = name;
}

public boolean isCompleted() {
    return isCompleted;
}

public void setCompleted(boolean completed) {
    isCompleted = completed;
}
```

نهایتا، متد toString هم override کنید تا بتونیم Stepهامون رو به راحتی چاپ کنیم:

```
@Override
public String toString() {
    return "Step{\n" +
            "\tid=" + id + ", \n" +
            "\ttask=" + task + ", \n" +
            "\tname='" + name + "', \n" +
            "\tisCompleted=" + isCompleted + "\n" +
            '}';
}
```

وقتشه که با annotate کردن کلاس Step، اون رو به Hibernate معرفی کنیم. بیشتر این annotationها، شبیه همون‌هایی‌ان که برای کلاس Task هم استفاده کردیم. از annotationهای Entity و Table شروع می‌کنیم:

```
@Entity
@Table(name = "steps")
```

```
public class Step {

    // CODE CODE CODE

}
```

حالا، فیلد id هم مثل کلاس Task که قبلا دیدیمش annotate می‌کنیم:

```
@Id
@GeneratedValue(strategy = GenerationType.IDENTITY)
private Integer id;
```

فیلدهای name و isCompleted هم، مثل قبل annotate می‌کنیم:

```
@Basic(optional = false)
private String name;

@Basic(optional = false)
@Column(name = "is_completed")
private boolean isCompleted;
```

تنها فیلد annotate نشده، فیلد taskئه. annotationهای این فیلد، با اون چیزی که قبلا دیده بودیم متفاوته. ستون task_id، سه ویژگی مهم داره که Hibernate باید از اون‌ها مطلع باشه:

- این ستون، نگه‌دارندهٔ یک foreign key به جدول tasksئه.
- این ستون، not nullئه و حتما باید مقداردهی بشه.
- هر task، ممکنه چند step متفاوت داشته باشه. این یعنی مقدار ستون task_id منحصر به فرد نیست و ممکنه stepهای مختلف، task_idهای یکسانی داشته باشن. از اون‌جا که چند step می‌تونن به یه task مرتبط باشن، به رابطهٔ بین این دو جدول ManyToOne می‌گن.[13]

از اولین ویژگی شروع می‌کنیم. با استفاده از annotationای به اسم JoinColumn، به Hibernate می‌گیم که ستونِ نظیر فیلد task، یه foreign keyئه:

[13] جدول‌های مختلف می‌تونن روابط متفاوتی با هم داشته باشن. رابطهٔ بین جدول tasks و steps، رابطهٔ ManyToOneئه. ولی رابطهٔ جداول people و students یک به یک یا OneToOneئه، چون دو student مختلف نمی‌تونن person_idهای یکسانی داشته باشن. ارتباط دو جدول ممکنه چند به چند یا ManyToMany هم باشه؛ مثلا جداول students و courses ارتباط ManyToMany با هم دیگه دارن، چون هر student می‌تونه چند course داشته باشه و هر course هم می‌تونه چند student رو شامل بشه. این ارتباط‌ها رو با جدولی مثل taken_courses مدیریت می‌کنیم، جدولی که یه foreign key به students داره و یه foreign key هم به courses. از این‌جا در مورد انواع روابط جداول توی دیتابیس‌های relational بیشتر بخونید.

```
@JoinColumn(name = "task_id")
private Task task;
```

همچنین، از اون‌جایی که Hibernate اسم ستون task_id رو نمی‌دونه، توی این annotation اون رو مشخص کردیم. از اون‌جایی که فیلد task از جنس Taskئه، Hibernate خودش می‌دونه که این foreign key به جدول tasks ارتباط داره.

حالا، باید دو ویژگیِ دوم این ستون رو به Hibernate معرفی کنیم. برای این که نشون بدیم ارتباط step و task از جنس ارتباط‌های ManyToOneئه، از annotationهای به همین اسم استفاده می‌کنیم[14]:

```
@ManyToOne(optional = false)
@JoinColumn(name = "task_id")
private Task task;
```

همچنین، با false گذاشتن optional، به Hibernate می‌گیم که این ستون not nullئه و مقداردهی اون اجباریه.[15] نهایتا، یادتون نره که توی فایل hibernate.cfg.xml هم این mapping رو مشخص کنید[16]:

```
<!-- Mappings -->
<mapping class="aut.ap.model.Task"/>
<mapping class="aut.ap.model.Step"/>
```

خیلی خب، حالا که ارتباط کلاس Step و جدول steps رو کاملا به Hibernate توضیح دادیم، می‌تونیم عملیات‌های دیتابیسی مختلف رو به کمک این کلاس انجام بدیم.

به متد main برگردین، تسک ۴ رو از دیتابیس بخونید و یه Step برای اون ایجاد کنید:

```
public static void main(String[] args) {
    setUpSessionFactory();
```

[14] همون‌طور که احتمالا حدس زدید، annotationهای OneToOne، ManyToMany یا حتی OneToMany هم توی Hibernate داریم. برای این که بیشتر راجع به اون‌ها یاد بگیرید، به داک رسمی Hibernate مراجعه کنید.

[15] کلا هر چی بهتر ساختار جداول‌تون رو به Hibernate توضیح بدین، اون هم می‌تونه کارش رو بهتر انجام بده. مثلا، ستون‌های nvarchar شما سایزهای مختلفی دارن، بعضی‌هاشون ممکنه ۱۰۰ کاراکتر نگه دارن و بعضی‌هاشون، ۸۰۰۰ کاراکتر! یکی از annotationهایی که قبلا دیدین، یعنی Column، یک فیلد length داره که اگر اون رو مقداردهی کنید Hibernate از طول رشته‌های ستون‌هاتون مطلع می‌شه. متاسفانه ما نمی‌تونیم توی این داک، همهٔ annotationهای Hibernate رو تا این حد بررسی کنیم؛ ولی خوبه که هر وقت از این annotationها استفاده می‌کنید به این موضوع فکر کنید که چطور می‌تونید جدول‌تون رو دقیق‌تر به Hibernate توضیح بدین و در رابطه باهاش سرچ کنید.

[16] من دائما این موضوع از یادم می‌ره و باعث می‌شه به Exceptionهای عجیب غریب بخورم!

```
    // Fetch 'Read "The Pragmatic Programmer"' task from DB
    Task t = sessionFactory.fromTransaction(session ->
            session.get(Task.class, 4));

    Step s = new Step(t, "Read chapter 1");

    closeSessionFactory();
}
```

حالا، با استفاده از تکه کد زیر، این Step رو توی دیتابیس ذخیره کنید:

```
sessionFactory.inTransaction(session -> {
    session.persist(s);
});
```

کد نهایی رو اجرا کنید و خروجی‌ش رو ببینید. توی این کد، Hibernate دوتا کوئری روی دیتابیس می‌زنه و اون‌ها رو لاگ می‌کنه. کوئری اولش برای خوندن تسک ۴ از دیتابیسه:

```
[Hibernate]
    select
        t1_0.id,
        t1_0.due_date,
        t1_0.name
    from
        tasks t1_0
    where
        t1_0.id=?
[TRACE] binding parameter (1:INTEGER) <- [4]
```

این کوئری رو قبلا هم دیدیم و آشناست. اما کوئری دوم Hibernate، که برای ذخیرهٔ Step جدیدمونه، جالب‌تره:

```
[Hibernate]
    insert
    into
        steps
        (is_completed, name, task_id)
    values
        (?, ?, ?)
[TRACE] binding parameter (1:BOOLEAN) <- [false]
[TRACE] binding parameter (2:VARCHAR) <- [Read chapter 1]
[TRACE] binding parameter (3:INTEGER) <- [4]
```

توی این لاگ، می‌بینید که Hibernate به درستی فهمیده که مقدار ستون task_id، ۴ئه! Hibernate می‌دونست که:

- ستون task_id به فیلد task مرتبطه
- فیلد task، از جنس Taskئه

- توی کلاس Task، فیلد id نمایندهٔ ستون primary keyی جدول tasksئه

با کنار هم گذاشتن این اطلاعات، Hibernate فهمید که مقدار درست ستون task_id توی Step جدیدمون چیه! اگر الآن به دیتابیس‌تون برین و روی جدول steps، select بزنید می‌بینید که Stepتون به درستی ذخیره شده:

	id	task_id	name	is_completed
1	1	4	Read chapter 1	0

حالا وقتشه این Step رو از دیتابیس بخونیم و اون رو چاپ کنیم. کد زیر رو به جای کد قبلی main بنویسید:

```
Step s = sessionFactory.fromTransaction(session ->
        session.get(Step.class, 1));
System.err.println(s);
```

کد بالا رو اجرا کنید و خروجی رو ببینید:

```
Step{
    id=1,
    task=Task{id=4, name='Read "The Pragmatic Programmer"', dueDate=2025-…},
    name='Read chapter 1',
    isCompleted=false
}
```

می‌بینید که Hibernate، نه تنها قدم ۱ رو به درستی از دیتابیس خوند، که تونست فیلد task هم به درستی با تسک ۴ پر کنه! foreign keyها و نحوهٔ مدیریت اون‌ها توی Hibernate، از جالب‌ترین بخش‌های این فریم‌ورکه. اگر به کوئریِ‌ای که Hibernate روی دیتابیس زد توجه کنید، می‌بینید که برای پر کردن فیلد task، Hibernate مجبور شد جداول steps و tasks رو با هم‌دیگه join کنه:

```
[Hibernate]
    select
        s1_0.id,
        s1_0.is_completed,
        s1_0.name,
        s1_0.task_id,
        t1_0.id,
        t1_0.due_date,
        t1_0.name
    from
        steps s1_0
    join
        tasks t1_0
            on t1_0.id=s1_0.task_id
    where
```

```
        s1_0.id=?
[TRACE] binding parameter (1:INTEGER) <- [1]
```

همون‌طور که توی بخش select می‌بینید، خروجی این کوئری ۷ ستون مختلف داره که Hibernate، با موفقیت، مقادیر اون‌ها رو به فیلدهای مختلف کلاس‌های ما داد.[17]

Lazy Fetching

joinای که بالا دیدیم، با این که خیلی به درد ما می‌خورد، ولی برای دیتابیس عملیات خیلی سختی بود. فرض کنید جداول steps و tasks، به جای یکی-دوتا رکورد تستی ما، هر کدوم ده میلیون رکورد داشتن.[18] در این صورت، اجرای عملیات join بین این دو جدول برای دیتابیس کار خیلی سختیه. ولی Hibernate با هر بار get کردن Stepها از دیتابیس، برای پر کردن فیلد task این join رو اجرا می‌کنه.

ما می‌تونیم از Hibernate بخوایم که این joinها رو بهینه‌تر انجام بده. به کلاس Step برگردین و بالای فیلد task، توی ManyToOne، عبارت fetch = FetchType.LAZY رو اضافه کنید[19]:

```
@ManyToOne(optional = false, fetch = FetchType.LAZY)
@JoinColumn(name = "task_id")
private Task task;
```

با اضافه کردن این مورد به annotationتون، شما به Hibernate می‌گید «تا زمانی که توی session، از فیلد task استفاده نکردم، اون رو مقداردهی نکن.». بیاین با یه کد ببینیم که این موضوع دقیقا یعنی چی. به متد main برگردین و کد زیر زیر رو بنویسید:

```
sessionFactory.inTransaction(session -> {
    Step s = session.get(Step.class, 1);

    System.err.println("id: " + s.getId());
    System.err.println("name: " + s.getName());
```

[17] اگر یادتون باشه، تعریف کردن List<Step> توی کلاس Task گاهی برای ما مشکل‌ساز می‌شد، ولی عدم تعریف اون هم برنامه‌نویسی رو برای ما سخت‌تر می‌کرد. خوشبختانه، با استفاده از روابط OneToManyی Hibernate می‌تونید به راحتی این فیلد رو توی کلاس Task تعریف کنید. برای آشنایی بیشتر با این روابط و مدیریت‌شون توی Hibernate، اینجا رو بخونید.

[18] این سناریو، به هیچ وجه دور از ذهن نیست. توی دنیای واقعی، دیتابیس‌ها از حجیم‌ترین بخش‌های یک برنامه هستن. تعداد رکوردهای اون‌ها گاهی انقدر زیاده که شرکت‌ها مجبور به استخدام database engineerها می‌شن تا دیتابیس‌شون رو بهتر مدیریت کنن.

[19] مقدار دیفالت fetch برای OneToMany و ManyToMany، FetchType.LAZYئه و برای ManyToOne و OneToOne، FetchType.EAGERئه. طبق داک رسمی Hibernate، توصیه می‌شه که شما تا جای ممکن از FetchType.LAZY استفاده کنید.

```
    System.err.println("is completed: " + s.isCompleted());
});
```

توی این کد، قدم ۱ رو از دیتابیس خوندیم، ولی هیچ‌وقت از فیلد task اون استفاده‌ای نکردیم. این کد رو اجرا کنید و خروجیِ اون رو ببینید:

```
[Hibernate]
    select
        s1_0.id,
        s1_0.is_completed,
        s1_0.name,
        s1_0.task_id
    from
        steps s1_0
    where
        s1_0.id=?
[TRACE] binding parameter (1:INTEGER) <- [1]

id: 1
name: Read chapter 1
is completed: false
```

همون‌طور که می‌بینید، از اون‌جایی که توی کد بالا از فیلد task استفاده نکردیم، Hibernate هم روی دیتابیس joinای نزد. حالا بیاین از این فیلد هم استفاده کنیم. کد زیر رو توی main بنویسید:

```
sessionFactory.inTransaction(session -> {
    Step s = session.get(Step.class, 1);

    System.err.println("id: " + s.getId());
    System.err.println("name: " + s.getName());
    System.err.println("is completed: " + s.isCompleted());
    System.err.println("task: " + s.getTask());
});
```

همون‌طور که می‌بینید، توی خط آخر lambdaمون، task رو هم چاپ کردیم. این کد رو اجرا کنید و خروجی رو ببینید:

```
[Hibernate]
    select
        s1_0.id,
        s1_0.is_completed,
        s1_0.name,
        s1_0.task_id
    from
        steps s1_0
    where
        s1_0.id=?
[TRACE] binding parameter (1:INTEGER) <- [1]

id: 1
name: Read chapter 1
```

```
is completed: false

[Hibernate]
    select
        t1_0.id,
        t1_0.due_date,
        t1_0.name
    from
        tasks t1_0
    where
        t1_0.id=?
[TRACE] binding parameter (1:INTEGER) <- [4]

task: Task{id=4, name='Read "The Pragmatic Programmer"', dueDate=2025-12-02}
```

کوئریِ اول Hibernate، همون کوئری‌ایه که قبلا دیده بودیم و برای get کردن قدم ائه. اما همون‌طور که می‌بینید، بلافاصله موقعی که خواستیم از فیلد task توی printمون استفاده کنیم، Hibernate به کوئری روی جدول tasks می‌زنه تا تسک ۴ رو پیدا کنه و فیلد task رو پر کنه. به خاطر همین هم، قبل از print ما فیلد task پر شده و مقدار اون آمادهٔ استفاده‌ست.

به این تکنیک توی Hibernate، Lazy Fetching می‌گن. این که Hibernate، تا زمانی که به یه object نیاز پیدا نکردیم، اون رو از دیتابیس نخونه خیلی جاها می‌تونه کمک کنه. مثلا اگر یادتون باشه، توی بخش «حذف رکوردها از دیتابیس» ما این مشکل رو مطرح کردیم که برای حذف یک رکورد از دیتابیس مجبوریم دوتا کوئری بزنیم، یکی برای get کردن اون رکورد و دیگری برای حذفش:

```
sessionFactory.inTransaction(session -> {
    Step s = session.get(Step.class, 1);
    session.remove(s);
});
```

یه متد، مشابه متد get، به اسم getReference داریم که می‌تونه توی این مشکل به ما کمک کنه. اگر به جای get، از getReference استفاده کنیم، Hibernate روی جدول steps کوئریِ select نمی‌زنه تا زمانی که به فیلدهای s نیاز داشته باشه:

```
sessionFactory.inTransaction(session -> {
    Step s = session.getReference(Step.class, 1);
    session.remove(s);
});
```

بعد از اجرای خط اول lambdaمون، تنها چیزی که Hibernate از s می‌دونه اینه که از جنس Stepئه و idش یکه. متد remove هم دقیقا به همین دو مورد نیاز داره تا بفهمه که s رو چجوری باید از دیتابیس

پاک کنه! به خاطر همین، اگر این کوئری رو اجرا کنید می‌بینید که هیچ وقت روی جدول steps، select زده نمی‌شه و حذف قدم ۱ توی تنها یک کوئری انجام می‌شه:

```
[Hibernate]
    delete
    from
        steps
    where
        id=?
[TRACE] binding parameter (1:INTEGER) <- [1]
```

استفاده از Lazy Fetching، چندان بی‌دردسر هم نیست. اگر بعد از بسته شدن sessionای که توی اون رکوردتون رو از دیتابیس get کرده بودین، بخواین به فیلدهایی که Hibernate اون‌ها رو از دیتابیس نخونده دسترسی پیدا کنید به خطای LazyInitializationException می‌خورین.

برای این که این exception رو با هم ببینیم، با استفاده از کد زیر یه قدم جدید توی دیتابیس ذخیره کنید:

```
sessionFactory.inTransaction(session -> {
    Task t = session.getReference(Task.class, 4);
    Step s = new Step(t, "read chapter 2");

    session.persist(s);
});
```

اگر دقت کنید برای select نزدن روی جدول tasks، توی این کد از getReference استفاده کردیم. بعد از اجرای این کد، روی جدول steps، select بزنید و مطمئن بشید که قدم جدیدتون به درستی توی دتیابیس ذخیره شده:

	id	task_id	name	is_completed
1	2	4	read chapter 2	0

حالا، اون رو به شکل زیر get و چاپ کنید:

```
Step s = sessionFactory.fromTransaction(session ->
        session.get(Step.class, 2));

System.err.println("id: " + s.getId());
System.err.println("name: " + s.getName());
System.err.println("is completed: " + s.isCompleted());
```

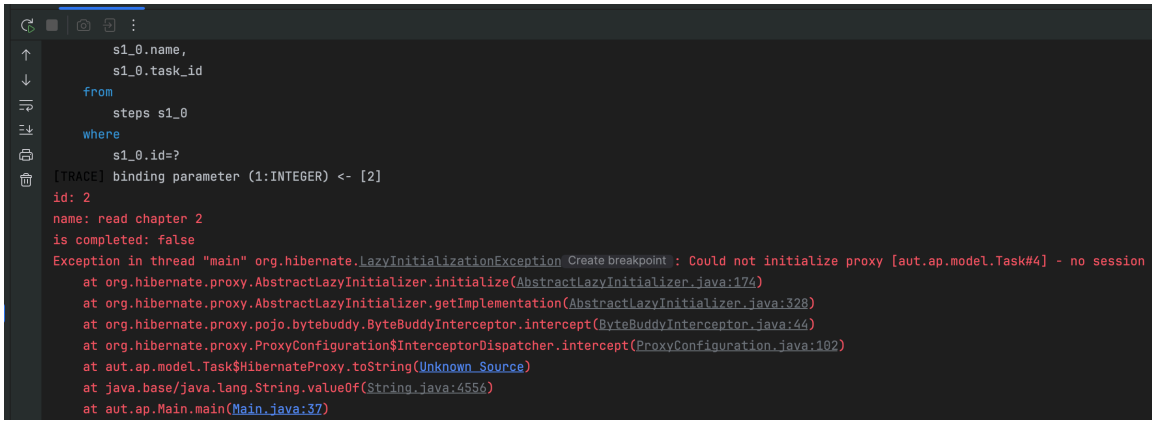
توی این کد، بعد از بسته شدن sessionمون، هیچ نیازی به فیلد task نداریم، به خاطر همین هم این کد به درستی اجرا می‌شه:

```
[Hibernate]
    select
        s1_0.id,
        s1_0.is_completed,
        s1_0.name,
        s1_0.task_id
    from
        steps s1_0
    where
        s1_0.id=?
[TRACE] binding parameter (1:INTEGER) <- [2]


id: 2
name: read chapter 2
is completed: false
```

ولی اگر بخوایم، بعد از بسته شدن sessionمون به فیلد task دسترسی پیدا کنیم، به خطای LazyInitializationException برمی‌خوریم. کد زیر رو اجرا کنید:

```
Step s = sessionFactory.fromTransaction(session ->
        session.get(Step.class, 2));

System.err.println("id: " + s.getId());
System.err.println("name: " + s.getName());
System.err.println("is completed: " + s.isCompleted());
System.err.println("task: " + s.getTask());
```

توی خط انتهایی این کد، task رو هم چاپ می‌کنیم. ولی اگر این کد رو اجرا کنید به خطا می‌خورید:



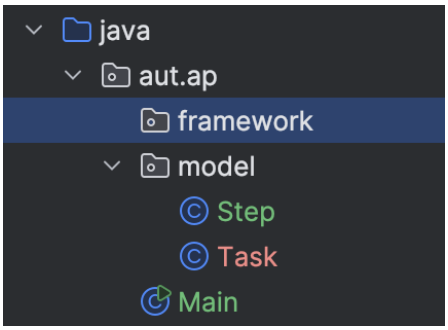
علت این خطا اینه که بعد از اتمام sessionتون، خواستین به فیلدی دسترسی پیدا کنید که Hibernate اون رو از دیتابیس نخونده. اگر به هر شکلی، توی sessionتون Hibernate رو مجبور کنید که فیلد task رو از دیتابیس بخونه، به این خطا نمی‌خورید.[20]

[20] برای این کار، می‌شه از EntityGraphها استفاده کرد. از این‌جا می‌تونید بیشتر راجع به اون‌ها بخونید.

سینگلتون کردن SessionFactory

همون‌طور که توی کدهای بالا دیدین، ما فقط به یک SessionFactory توی کل کدهامون نیاز داریم. ممکنه توی یک کد، چندین Session مختلف درست کنیم، ولی همهٔ اون‌ها، فقط از یک SessionFactory گرفته می‌شن. علاوه بر این، ساخت آبجکت‌های SessionFactory برای Hibernate کار سختیه و به همین خاطر، بهتره که توی برنامه‌مون از فقط یک SessionFacroty استفاده کنیم.

برای این کار، از دیزاین پترنی به اسم Singleton استفاده می‌کنیم. اصلا دیزاین پترن پیچیده‌ای نیست، ولی به ما اجازه می‌ده که فقط یک SessionFactory توی کل برنامه‌مون استفاده کنیم. برای استفاده از اون، پکیج aut.ap.framework رو ایجاد کنید:



حالا، توی این پکیج کلاس SingletonSessionFactory رو ایجاد کنید و توی اون، کدهای زیر رو بنویسید:

```
package aut.ap.framework;

import org.hibernate.SessionFactory;
import org.hibernate.cfg.Configuration;

public class SingletonSessionFactory {
    private static SessionFactory sessionFactory = null;

    public static SessionFactory get() {
        if (sessionFactory == null) {
            sessionFactory = new Configuration()
                    .configure("hibernate.cfg.xml")
                    .buildSessionFactory();
        }

        return sessionFactory;
    }

    public static void close() {
        if (sessionFactory == null) {
```

```
            return;
        }

        sessionFactory.close();
    }
}
```

فهم این کد، کار نسبتا راحتیه. توی کلاس SingletonSessionFactory یه فیلد از جنس SessionFactory داریم که مقدار اولیهٔ اون، nullئه. توی متد get، اول چک می‌کنیم که اگر sessionFactoryمون null بود، کانفیگ hibernate.cfg.xml رو بخونیم و اون رو مقداردهی کنیم. این باعث می‌شه توی اولین مرتبه‌ای که متد get صدا زده شد، sessionFactory مقداردهی بشه و بعد از اون، همیشه همون sessionFactory قدیمی‌مون خروجی داده بشه. به خاطر همین موضوع، کل برنامهٔ ما از دقیقا یک SessionFactory استفاده می‌کنه. نهایتا توی متد close هم SessionFactoryمون رو بستیم.

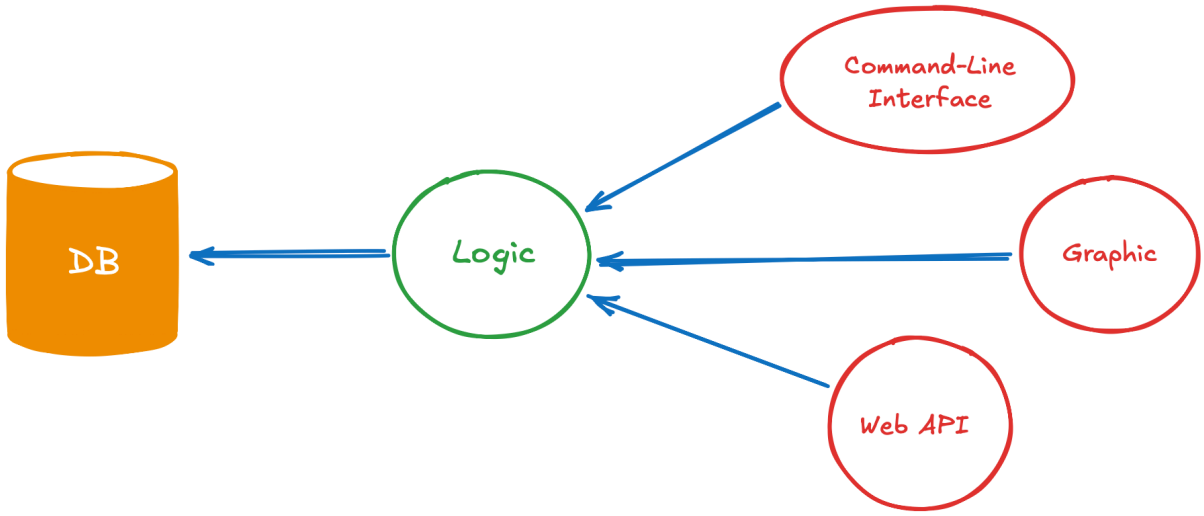
الآن می‌تونیم توی Main، از دست متدهای setUpSessionFactory و closeSessionFactory خلاص بشیم و از SingletonSessionFactory استفاده کنیم:

```
public class Main {
    public static void main(String[] args) {
        SingletonSessionFactory.get()
                .inTransaction(session -> {
                    Task t = session.getReference(Task.class, 4);
                    Step s = new Step(t, "read chapter 3");

                    session.persist(s);
                });

        SingletonSessionFactory.close();
    }
}
```

با اجرای کد بالا، یک قدم جدید برای تسک ۴ توی دیتابیس‌تون ذخیره می‌شه.

افزودن Serviceها

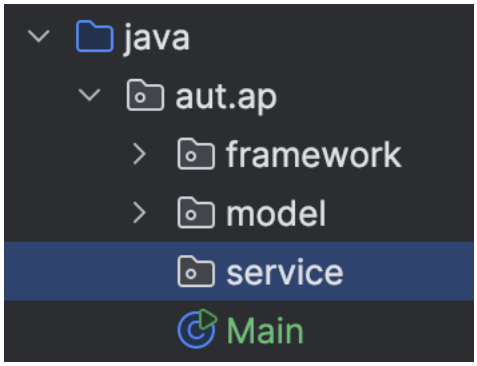
همون‌طور که می‌دونید، خوبه که که لایه‌های UI و Logic برنامه‌مون جدا باشن.[21] به عبارتی، ما منتقل برنامه‌هامون رو، کنار کدهای ورودی و خروجی نمی‌نویسیم. ساختار برنامه‌های ما، یه همچین شکلی داره:



توی این ساختار، وظیفهٔ لایهٔ UI صرفا ورودی گرفتن و خروجی دادن به کاربره. لایهٔ UI، بعد از گرفتن ورودی‌های کاربر، اون‌ها رو به لایهٔ Logic می‌ده و توی این لایه، بر اساس منطق برنامه‌مون ورودی‌ها پردازش می‌شه و عملیات دلخواه کاربر انجام می‌شه.

ما لایهٔ Logic رو با استفاده از Serviceها پیاده‌سازی می‌کنیم. پکیج aut.ap.service رو به برنامه‌تون اضافه کنید:

[21] اگر راستش رو بخواید، به نظر خود من ایراد خیلی بزرگی نیست وقتی این دو لایه ترکیب بشن. درسته که دیزاین برنامه‌مون کمی به هم می‌ریزه، ولی راستش همیشه هم لازم نیست که دیزاین برنامه‌های ما بی‌نقص باشه. اگر خیلی دنبال دیزاین‌های بی‌نقص بگردیم و وقت بگذرونیم، ممکنه هیچ‌وقت به کد زدن نرسیم. علاوه بر این موضوع، نظرات دولوپرهای مختلف هم در مورد یک دیزاین خوب گاهی خیلی متفاوته. به همین خاطر، من اصلا شک داشتم که آیا این بخش رو برای شما بنویسم یا نه. این بخش رو بخونید، و سعی کنید بهش عمل کنید، ولی بدونید که راهی که این‌جا توصیف شده، تنها راه ممکن برای نوشتن یک برنامهٔ خوب نیست.



توی این پکیج، دو کلاس TaskService و StepService رو ایجاد کنید. توی این داکیومنت، سرویس‌های این دو کلاس رو به صورت کامل پیاده‌سازی نمی‌کنیم و فقط بخشی از متدهای اون‌ها رو می‌نویسیم. به کلاس TaskService برین:

```
package aut.ap.service;

public class TaskService {
}
```

توی این کلاس، متد persist، که برای ذخیرهٔ یک Task استفاده می‌شه رو پیاده‌سازی می‌کنیم:

```
public static Task persist(String name, LocalDate dueDate) {
    Task t = new Task(name, dueDate);

    SingletonSessionFactory.get()
            .inTransaction(session -> {
                session.persist(t);
            });

    return t;
}
```

حالا، یه متد هم برای گرفتن همهٔ تسک‌ها می‌نویسیم:

```
public static List<Task> getAll() {
    return SingletonSessionFactory.get()
            .fromTransaction(session ->
                 session.createNativeQuery("select * from tasks", Task.class)
                         .getResultList());
}
```

و همچنین، یه متد هم برای حذف تسک‌هامون می‌نویسیم:

```
public static void remove(int id) {
    SingletonSessionFactory.get()
            .inTransaction(session -> {
                Task t = session.getReference(Task.class, id);
                session.remove(t);
```

```
            });
}
```

حالا می‌تونیم به سراغ StepService بریم. برای اون هم سه متد مشابه تعریف می‌کنیم:

```
public class StepService {
    public static Step persist(Task task, String name) {
        Step s = new Step(task, name);

        SingletonSessionFactory.get()
                .inTransaction(session -> {
                    session.persist(s);
                });

        return s;
    }

    public static List<Step> getAll() {
        return SingletonSessionFactory.get()
                .fromTransaction(session ->
                    session.createNativeQuery("select * from steps",
Step.class)
                            .getResultList());
    }

    public static void remove(int id) {
        SingletonSessionFactory.get()
                .inTransaction(session -> {
                    Step s = session.getReference(Step.class, id);
                    session.remove(s);
                });
    }
}
```

حالا، ما می‌تونیم از توی Main، یعنی لایهٔ Command-Line Interfaceمون، با گرفتن ورودی‌های کاربر متد مناسب رو از Serviceها صدا بزنیم و خروجی اون رو هم به کاربر نشون بدیم.

چیزی که یاد گرفتیم

توی این داک، ما دانش‌مون از SQL و جاوا رو کنار هم گذاشتیم و اولین قدم‌هامون رو توی دنیای گستردهٔ Hibernate در کنار هم زدیم. توی این داک فهمیدیم که:

- ORMها چی‌ان و به چه درد می‌خورن.
- چطور می‌شه موجودیت‌های مختلف رو به Hibernate معرفی کرد.
- چطور می‌شه با استفاده از Hibernate، روی دیتابیس کوئری زد.
- تکنیک‌هایی مثل Lazy Fetching توی دیتابیس چطور کار می‌کنن.

منابع بیشتر

Hibernate، خیلی دنیای بزرگی داره و ما به هیچ وجه نمی‌تونستیم تمام اون رو توی این داک بررسی کنیم. این داک، صرفا برای یک آشنایی اولیه با دنیای ORMهاست و خیلی خوبه که شما بعد از خوندنش، از ریسورس‌های آنلاین استفاده کنید و Hibernate رو بهتر یاد بگیرید. by the way، من هم برای اینکه Hibernate رو بهتر یاد بگیرم، یه پروژهٔ «مدیریت دانشگاه» زدم و سورسش رو روی گیت‌هاب گذاشتم، اگر دوست داشتین به این پروژه هم یه نگاه بندازید.

بین کتاب‌های و منابعی که من دیدم، یکی از بهترین منابع برای شروع کار با Hibernate داک رسمی «An Introduction to Hibernate 6»ئه. این داک برای این نوشته شده که دولوپرهای تازه‌کاری مثل شما راحت‌تر به دنیای Hibernate قدم بذارن. اگر اون رو خوندین و سوالی داشتین، حتما از تدریس‌یارهاتون بپرسین.

علاوه بر این، برای این که بفهمید متدهای مختلف توی Hibernate دقیقا چه کار می‌کنن، می‌تونید به داکیومنت Hibernate 6 هم سر بزنید.