Date and Time Mappings with Hibernate and JPA

Databases support various data types to store date and time information. The most commonly used ones are:

- DATE to save a date without time information,
- TIME to store a time without a date, and
- TIMESTAMP to store date and time information.

You can map all of them with JPA and Hibernate.

But you need to decide to which Java type you want to map your database column. The Java language supports a bunch of classes to represent date and time information, like:

- java.util.Date and java.util.Calendar
- java.sql.Date, java.sql.Time and java.sql.Timestamp
- java.time.LocalDate, java.time.LocalDateTime, java.time.OffsetTime, java.time.OffsetDateTime, java.time.ZonedDateTime, java.time.Duration

JPA supports most of them. In addition to that, Hibernate provides proprietary support for almost all of the remaining ones.

Mapping java.util classes

Before the release of Java 8, java.util.Date and java.util.Calendar were the most commonly used classes to represent dates with and without time information.

You can, of course, map both of them with JPA and Hibernate. But the mapping requires a few additional information. You need to define if you want to map the java.util.Date or java.util.Calendar to a column of type DATE, TIME or TIMESTAMP.

You can do that by annotating the entity attribute with @Temporal and providing a TemporalType enum value as a value. You can choose between:

- TemporalType.DATE to map it to a SQL DATE column
- TemporalType.TIME to map it to a SQL TIME column

- TemporalType.TIMESTAMP to map it to a SQL TIMESTAMP column

I'm using the @Temporal annotation in the following code snippet to map an attribute of type java.util.Date to a TIMESTAMP column and an attribute of type java.util.Calendar to a DATE column.

```
@Entity
public class MyEntity {

    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private Long id;

    @Temporal(TemporalType.TIMESTAMP)
    private Date utilDate;

    @Temporal(TemporalType.DATE)
    private Calendar utilCalendar;

    ...
}
```

You can then use these attributes in the same way you use any other entity attributes.

```
MyEntity e = new MyEntity();
e.setUtilDate(new Date(119, 6, 18));
e.setUtilCalendar(new GregorianCalendar(2019, 6, 18));
em.persist(e);
```

If you activate the logging of SQL statements, you can find the following messages in your log file.

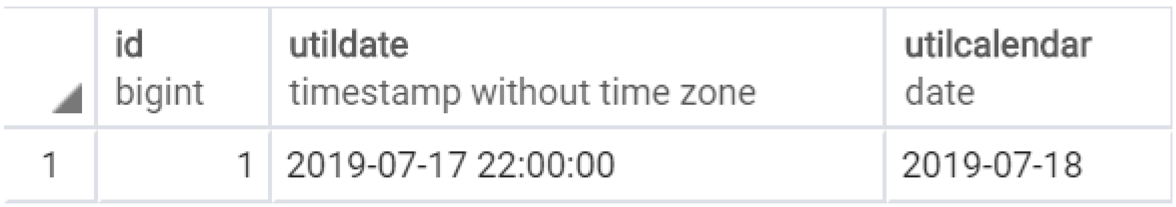
```
16:04:07,185 DEBUG SQL:92 - insert into MyEntity (utilCalendar, utilDate, id)
values (?, ?, ?)

16:04:07,202 TRACE BasicBinder:65 - binding parameter [8] as [DATE] -
[java.util.GregorianCalendar[time=1563400800000,areFieldsSet=true,areAllFields
Set=true,lenient=true,zone=sun.util.calendar.ZoneInfo[id="Europe/Berlin",offset=3
600000,dstSavings=3600000,useDaylight=true,transitions=143,lastRule=java.util.
SimpleTimeZone[id=Europe/Berlin,offset=3600000,dstSavings=3600000,useDayli
ght=true,startYear=0,startMode=2,startMonth=2,startDay=-
1,startDayOfWeek=1,startTime=3600000,startTimeMode=2,endMode=2,endMont
h=9,endDay=-
1,endDayOfWeek=1,endTime=3600000,endTimeMode=2]],firstDayOfWeek=2,min
imalDaysInFirstWeek=4,ERA=1,YEAR=2019,MONTH=6,WEEK_OF_YEAR=29,W
EEK_OF_MONTH=3,DAY_OF_MONTH=18,DAY_OF_YEAR=199,DAY_OF_WE
EK=5,DAY_OF_WEEK_IN_MONTH=3,AM_PM=0,HOUR=0,HOUR_OF_DAY=0,
MINUTE=0,SECOND=0,MILLISECOND=0,ZONE_OFFSET=3600000,DST_OFF
SET=3600000]]

16:04:07,207 TRACE BasicBinder:65 - binding parameter [2] as [TIMESTAMP] -
[Thu Jul 18 00:00:00 CEST 2019]

16:04:07,208 TRACE BasicBinder:65 - binding parameter [3] as [BIGINT] - [1]
```

The 2nd message about the binding of the GregorianCalendar might surprise you. That's Hibernate's very complicated way to show you which Calendar object gets bound to a parameter of type DATE. But don't worry, if you take a look at the database, you can see that Hibernate wrote the date to a column of type DATE.



	id bigint	utildate timestamp without time zone	utilcalendar date
1	1	2019-07-17 22:00:00	2019-07-18

Mapping java.sql classes

The mappings of the java.sql classes Date, Time, and Timestamp are easier than the previously shown mappings of the java.util classes.

That's because the classes in the java.sql package match the SQL data types.

That enables your persistence provider, e.g., Hibernate, to identify the mapping automatically. So, without providing any additional annotations:

- java.sql.Date gets mapped to SQL DATE,
- java.sql.TIME gets mapped to SQL TIME and
- java.sql.TIMESTAMP gets mapped to SQL TIMESTAMP.

```
@Entity
public class MyEntity {

    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private Long id;

    private java.sql.Date sqlDate;
    private Time sqlTime;
    private Timestamp sqlTimestamp;

    ...
}
```

You can then use these attributes in your business code to store date and time information in your database.

```
MyEntity e = new MyEntity();
e.setSqlDate(new java.sql.Date(119, 6, 18));
e.setSqlTime(new Time(15, 05, 30));
e.setSqlTimestamp(new Timestamp(119, 6, 18, 15, 05, 30, 0));
em.persist(e);
```

And after you activated the logging of SQL statements, you can see that Hibernate maps the entity attributes to the corresponding SQL types.

```
06:33:09,139 DEBUG SQL:92 - insert into MyEntity (sqlDate, sqlTime,
sqlTimestamp, id) values (?, ?, ?, ?)
06:33:09,147 TRACE BasicBinder:65 - binding parameter [1] as [DATE] - [2019-
07-18]
06:33:09,147 TRACE BasicBinder:65 - binding parameter [2] as [TIME] -
[15:05:30]
06:33:09,147 TRACE BasicBinder:65 - binding parameter [3] as [TIMESTAMP] -
[2019-07-18 15:05:30.0]
06:33:09,154 TRACE BasicBinder:65 - binding parameter [4] as [BIGINT] - [1]
```

Mapping java.time classes

Java 8 introduced the Date and Time API to fix the flaws of the java.util.Date class. Using the new API is clean and concise, and you can finally distinguish between date and time information.

Since Hibernate 5 and JPA 2.2, you can use the following classes as attribute types.

Java Type	JPA	Hibernate	JDBC Type
java.time.LocalDate	X	X	DATE
java.time.LocalTime	X	X	TIME
java.time.LocalDateTime	X	X	TIMESTAMP
java.time.OffsetTime	X	X	TIME_WITH_TIMEZONE
java.time.OffsetDateTime	X	X	TIMESTAMP_WITH_TIMEZONE
java.time.Duration	-	X	BIGINT
java.time.Instant	-	X	TIMESTAMP
java.time.ZonedDateTime	-	X	TIMESTAMP

As you can see in the table, Hibernate supports a few more Date and Time classes than JPA. You can easily add support for additional classes by implementing an AttributeConverter. I used it in a previous article to map an attribute of type Duration with JPA.

Hibernate's proprietary support for Duration and Instant works fine. But you should be careful if you want to use Hibernate's mapping of ZonedDateTime. The handling of timezones and the mapping to a TIMESTAMP column presents a few pitfalls. I get into more details about that in the Working with ZonedDateTime section.

Let's first take a look at a basic entity mapping.

```
@Entity
public class MyEntity {

    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private Long id;

    private LocalDate localDate;
    private LocalDateTime localDateTime;
    private OffsetTime offsetTime;
    private OffsetDateTime offsetDateTime;

    // Hibernate-specific - not supported by JPA
    private Duration duration;
    private Instant instant;
    private ZonedDateTime zonedDateTime;
    ...
}
```

In this example, I map entity attributes of the 7 types supported by JPA. As you can see, these mappings don't require any annotations.

```
MyEntity e = new MyEntity();

e.setLocalDate(LocalDate.of(2019, 7, 19));

e.setLocalDateTime(LocalDateTime.of(2019, 7, 19, 15, 05, 30));

e.setOffsetTime(OffsetTime.of(15, 05, 30, 0, ZoneOffset.ofHours(+2)));

e.setOffsetDateTime(OffsetDateTime.of(2019, 7, 19, 15, 05, 30, 0,
ZoneOffset.ofHours(+2)));


// Hibernate-specific - not supported by JPA

e.setDuration(Duration.ofHours(2));

e.setInstant(Instant.now());

e.setZonedDateTime(ZonedDateTime.of(2019, 7, 18, 15, 05, 30, 0,
ZoneId.of("UTC-4")));


em.persist(e);
```

The classes of the Date and Time API clearly define if they store date and/or time information. So, the JPA specification and all implementing frameworks can map them to the correct SQL types.

```
11:52:26,305 DEBUG SQL:94 - insert into MyEntity (duration, instant, localDate,
localDateTime, offsetDateTime, offsetTime, sqlDate, sqlTime, sqlTimestamp,
utilCalendar, utilDate, zonedDateTime, id) values (?, ?, ?, ?, ?, ?, ?, ?)

11:52:26,306 TRACE BasicBinder:65 - binding parameter [1] as [BIGINT] - [PT2H]

11:52:26,307 TRACE BasicBinder:65 - binding parameter [2] as [TIMESTAMP] -
[2019-07-22T09:52:26.284946300Z]

11:52:26,308 TRACE BasicBinder:65 - binding parameter [3] as [DATE] - [2019-
07-19]

11:52:26,308 TRACE BasicBinder:65 - binding parameter [4] as [TIMESTAMP] -
[2019-07-19T15:05:30]

11:52:26,312 TRACE BasicBinder:65 - binding parameter [5] as [TIMESTAMP] -
[2019-07-19T15:05:30+02:00]

11:52:26,313 TRACE BasicBinder:65 - binding parameter [6] as [TIME] -
[15:05:30+02:00]

11:52:26,324 TRACE BasicBinder:65 - binding parameter [7] as [TIMESTAMP] -
[2019-07-18T15:05:30-04:00[UTC-04:00]]

11:52:26,324 TRACE BasicBinder:65 - binding parameter [8] as [BIGINT] - [1]
```

Working with ZonedDateTime

As I mentioned early, using Hibernate’s support for ZonedDateTime is risky. And let’s be honest, I don’t recommend using it.

Hibernate maps a ZonedDateTime to an SQL TIMESTAMP without time zone information. It converts the ZonedDateTime into the local time zone of the JVM and then stores it in the database. And when it reads the TIMESTAMP, it adds the local time zone information to it.

```
MyEntity e = new MyEntity();

e.setZonedDateTime(ZonedDateTime.of(2019, 7, 18, 15, 05, 30, 0,
ZoneId.of("UTC-4")));

em.persist(e);
```

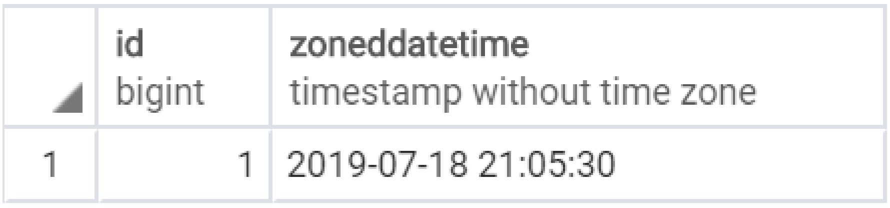
Hibernate shows the time zone information in the log message.

```
09:57:08,918 DEBUG SQL:92 - insert into MyEntity (zonedDateTime, id) values
(?, ?)

09:57:08,959 TRACE BasicBinder:65 - binding parameter [1] as [TIMESTAMP] -
[2019-07-18T15:05:30-04:00[UTC-04:00]]

09:57:08,961 TRACE BasicBinder:65 - binding parameter [2] as [BIGINT] - [1]
```

But you can see in the database, that it converted the time zone from UTC-4 to UTC+2, which is my local time zone.



	id bigint	zoneddatetime timestamp without time zone
1	1	2019-07-18 21:05:30

This mapping works as long as:

- you use a time zone without daylight saving time,
- all instances of your application use the same time zone, and
- you never have to change this time zone.

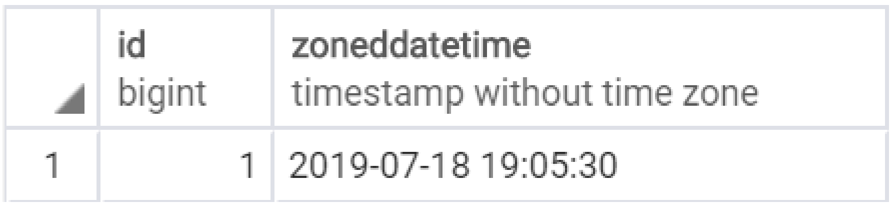
You can avoid these problems by configuring a time zone without daylight saving time in your persistence.xml configuration. Hibernate will then use the configured time zone instead of the one used by your local JVM. I recommend using the UTC time zone.

```
<persistence>
    <persistence-unit name="my-persistence-unit">
        ...
        <properties>
            <property name="hibernate.dialect"
value="org.hibernate.dialect.PostgreSQLDialect" />
            <property name="hibernate.jdbc.time_zone" value="UTC"/>
            ...
        </properties>
    </persistence-unit>
</persistence>
```

When you now rerun the test, you will not see any difference in the log file.

```
10:06:41,070 DEBUG SQL:92 - insert into MyEntity (zonedDateTime, id) values
(?, ?)
10:06:41,107 TRACE BasicBinder:65 - binding parameter [1] as [TIMESTAMP] -
[2019-07-18T15:05:30-04:00[UTC-04:00]]
10:06:41,108 TRACE BasicBinder:65 - binding parameter [2] as [BIGINT] - [1]
```

But if you look at the database table, you can see, that Hibernate now converted the ZonedDateTime to the UTC time zone.



	id bigint	zoneddatetime timestamp without time zone
1	1	2019-07-18 19:05:30

Conclusion

JPA and Hibernate can map database columns of type DATE, TIME and TIMESTAMP to various Java classes. You can map them to:

- java.util.Date and java.util.Calendar
- java.sql.Date, java.sql.Time and java.sql.Timestamp
- java.time.LocalDate, java.time.LocalDateTime, java.time.OffsetTime, java.time.OffsetDateTime, java.time.ZonedDateTime, java.time.Duration

You just need to decide which Java type you want to use in your code. I recommend using the classes in the java.time package. They are part of the Date and Time API, which was introduced in Java 8. These classes are a lot easier to use in your mapping and your business code.