Hibernate 6.0 brought us many internal improvements. And for a moment, it seemed like we wouldn't get much out of it for our day-to-day projects. But with versions 6.1 and 6.2, we start to see the benefits of those changes. Both releases included a bunch of interesting improvements and new features.

Based on JPA 3

Hibernate 6.0 updates the supported JPA version to 3.0.

JPA 3.0 renamed all packages and configuration parameters from *javax.persistence* to *jakarta.persistence*. The easiest way to adapt your project is to use the search and replace command in your IDE or on the command line to replace *javax.persistence* with *jakarta.persistence*.

I explained this and other required changes in more details in my Migrating guide to Hibernate 6.

Proprietary Criteria API got removed

Hibernate's proprietary Criteria API got deprecated in version 5 and removed in version 6.

If you're still using the old API, you need to reimplement it using JPA's Criteria API. I explained this in my Migrating from Hibernate's to JPA's Criteria API article.

New default sequences

Starting with version 6, Hibernate uses a separate default sequence for every entity class. It follows the naming pattern *<table name>_SEQ*.

If you want to keep using the old default naming strategy, you need to configure the ID_DB_STRUCTURE_NAMING_STRATEGY property in your *persistence.xml* file. If you set it to legacy, Hibernate will use the same default strategy as in the Hibernate versions >= 5.3 but <6. And if you set it to single, you get the same default strategy as in Hibernate versions < 5.3.

```
<persistence>
    <persistence-unit name="my-persistence-unit">
        ...
        <properties>
            <property name="hibernate.id.db_structure_naming_strategy"
value="standard" />
            ...
        </properties>
    </persistence-unit>
</persistence>
```

You can learn more about Hibernate's different naming strategies in my guide to Hibernate 6's sequence naming strategies.

Incubating features

The new *@Incubating* annotation tells you that a new API or configuration parameter might still change.

The Hibernate team wants to use it when they release a feature for which they're looking for further user feedback or if they're releasing a subset of a set of interconnected features. In that case, some of the missing features might require additional changes on the already released APIs.

You can find out more about it in my guide to @Incubating features in Hibernate 6.

ResultTransformer replaced by TupleTransformer and ResultListTransformer

Hibernate 6 split the *ResultTransformer* interface into the *TupleTransformer* and the *ResultListTransformer* interfaces. These are functional interfaces that separate the transformation of a single tuple from the transformation of the list of transformed tuples.

```
PersonDTO person = (PersonDTO) session.createQuery(
          "select id as personId, first_name as firstName, "
          + "last_name as lastName, city from Person p", Object[].class)
     .setTupleTransformer((tuples, aliases) -> {
                              log.info("Transform tuple");
                              PersonDTO personDTO = new PersonDTO();
                              personDTO.setPersonId((int)tuples[0]);
                              personDTO.setFirstName((String)tuples[1]);
                              personDTO.setLastName((String)tuples[2]);
                              return personDTO;
     }).getSingleResult();
```

The Hibernate team converted all their transformer implementations to the new interfaces. So, the required changes to your code base are minimal.

I explained the ResultTransformer feature in more details in Hibernate's ResultTransformer in Hibernate 4, 5 & 6.

Improved JSON mapping

Hibernate 6 can map a JSON document to an *@Embeddable*.

You define an embeddable by implementing a Java class and annotating it with *@Embeddable*.

```
@Embeddable
public class MyJson implements Serializable {

    private String stringProp;

    private Long longProp;

    ...
}
```

To use it as an entity attribute and map it to a JSON document, you need to annotate it with @Embedded and @JdbcTypeCode(SqlTypes.JSON) and include a JSON mapping library in your classpath.

```
@Entity
public class MyEntity {

    @Embedded
    @JdbcTypeCode(SqlTypes.JSON)
    private MyJson jsonProperty;

    ...
}
```

Java Records can be embeddables

Records are an obvious match for any immutable set of information, but JPA doesn't support records as embeddables or entity classes.

```
@Embeddable
@EmbeddableInstantiator(AddressInstantiator.class)
public record Address (String street, String city, String postalCode) {}
```

Using Hibernate ORM 6.0 or 6.1, you need to implement an *EmbeddableInstantiator* for each record you want to use as an embeddable.

```
public class AddressInstantiator implements EmbeddableInstantiator {

    Logger log = LogManager.getLogger(this.getClass().getName());

    public boolean isInstance(Object object, SessionFactoryImplementor
sessionFactory) {
        return object instanceof Address;
    }

    public boolean isSameClass(Object object, SessionFactoryImplementor
sessionFactory) {
        return object.getClass().equals( Address.class );
    }

    public Object instantiate(ValueAccess valuesAccess,
SessionFactoryImplementor sessionFactory) {
        // valuesAccess contains attribute values in alphabetical order
        final String city = valuesAccess.getValue(0, String.class);
        final String postalCode = valuesAccess.getValue(1,
String.class);
        final String street = valuesAccess.getValue(2, String.class);
        log.info("Instantiate Address embeddable for "+street+"
"+postalCode+" "+city);
        return new Address( street, city, postalCode );
    }
}
```

The *EmbeddableInstantiator* is a proprietary Hibernate interface that tells Hibernate how to instantiate an embeddable object. This removes JPA's requirement of a no-argument constructor and enables you to model an embeddable as a record.

Starting with version 6.2, Hibernate provides its own *EmbeddableInstantiator* for all embeddable records and you no longer need to provide any extra code or mapping definition.

Improved OffsetDateTime and ZonedDateTime mapping

Hibernate 5 converts objects of type *OffsetDateTime* and *ZonedDateTime* into the local timezone of your application before persisting them. It then adds the local timezone when reading the timestamp.

Hibernate 6 improves this mapping by introducing the *@TimeZoneStorage* annotating and supporting 5 different mappings:

- *TimeZoneStorageType.NATIVE* stores the timestamp in a column of type *TIMESTAMP_WITH_TIMEZONE*,
- *TimeZoneStorageType.NORMALIZE* uses the mapping introduced in Hibernate 5. It normalizes the timestamp to your JDBC driver's local timezone and persists it without timezone information,
- *TimeZoneStorageType.NORMALIZE_UTC* normalizes the timestamp to UTC and persists it without timezone information,
- *TimeZoneStorageType.COLUMN* stores the timestamp without timezone information in one column and the difference between the provided timezone and UTC in another column,
- *TimeZoneStorageType.AUTO* lets Hibernate choose based on the capabilities of your database. It either uses *TimeZoneStorageType.NATIVE* or *TimeZoneStorageType.COLUMN*.