Mappings: Primary Keys

When you use Hibernate with a MySQL database, you should use the *IDENTITY* strategy to generate primary key values. This strategy uses an auto incremented database column and is the most efficient approach support by MySQL.

```
@Entity
public class Author {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    @Column(name = "id", updatable = false, nullable = false)
    private Long id;

    ...
}
```

Mappings: Problems with GenerationType.AUTO in Hibernate 5

When you use the *GenerationType.AUTO*, Hibernate selects the generation strategy based on the Hibernate dialect. That's a common approach if you need to support multiple databases.

In older versions, Hibernate selected the *GenerationType.IDENTITY* for MySQL databases. But that changed in Hibernate 5. It now selects the *GenerationType.TABLE* which has huge scalability issues.

You can avoid that by defining a @GenericGenerator which tells Hibernate to use the native strategy to generate the primary key values.

```
@Entity
public class Author {


@Id
    @GeneratedValue(strategy = GenerationType.AUTO,
                                generator = "native")
    @GenericGenerator(name = "native", strategy = "native")
    @Column(name = "id", updatable = false, nullable = false)
    private Long id;
    ...
}
```

Mappings: Read-only Views

With JPA and Hibernate, you can map views in the same way as any database table. If the view is read-only, you should tell Hibernate about it with an *@Immutable* annotation. It will then ignore all changes to this entity.

```
@Entity
@Immutable
public class BookView { … }
```

Queries: MySQL-specific functions and data types

As every other database, MySQL extends the SQL standard with a set of custom functions and data types. Examples for that are the JSON data type and the sysdate function.

These are not supported by JPA but thanks to Hibernate's MySQL dialect, you can use them anyways.

```
Query q = em.createQuery(
        "SELECT a, sysdate() FROM Author a ");
List<Object[]> results = q.getResultList();
```

If you find a function or data type that are not supported by Hibernate's MySQL dialect, you can use an AttributeConverter to convert the data type to a supported one and the JPQL function *function* to call any function within a JPQL query.

Queries: Stored Procedures

Since JPA 2.1, you can use a *@NamedStoredProcedureQuery* or a *StoredProcedureQuery* to define a stored procedure call.

@NamedStoredProcedureQuery

The @NamedStoredProcedureQuery annotation allows you to define the stored procedure call once and to reference it by its name in your business code.

```
@NamedStoredProcedureQuery(
    name = "calculate",
    procedureName = "calculate",
    parameters = {@StoredProcedureParameter(
                mode = ParameterMode.IN,
                type = Double.class, name = "x"),
            @StoredProcedureParameter(
                mode = ParameterMode.IN,
                type = Double.class, name = "y"),
            @StoredProcedureParameter(
                mode = ParameterMode.OUT,
                type = Double.class, name = "sum") })
```

```
StoredProcedureQuery query =
        em.createNamedStoredProcedureQuery("calculate");
query.setParameter("x", 1.23d);
query.setParameter("y", 4d);
query.execute();
Double sum =
        (Double) query.getOutputParameterValue("sum");
```

StoredProcedureQuery

The programmatic definition of a stored procedure call is very similar to the annotation based approach I showed you in the previous example.

```
StoredProcedureQuery query =
em.createStoredProcedureQuery("calculate");

query.registerStoredProcedureParameter("x", Double.class,
ParameterMode.IN);

query.registerStoredProcedureParameter("y", Double.class,
ParameterMode.IN);

query.registerStoredProcedureParameter("sum",
Double.class, ParameterMode.OUT);
```

That's all you need to do define the stored procedure call. You can then use it in the same way as the @NamedStoredProcedureQuery.

```
query.setParameter("x", 1.23d);

query.setParameter("y", 4d);

query.execute();
```