Implicit Naming Strategy

Hibernate uses a logical name to map an entity or attribute name to a table or column name. This name can be customized in two ways: it can be derived automatically by using an ImplicitNamingStrategy or it can be defined explicitly by using annotations.

The ImplicitNamingStrategy governs how Hibernate derives a logical name from our Java class and property names. We can select from four built-in strategies, or we can create our own.
For this example, we'll use the default strategy, ImplicitNamingStrategyJpaCompliantImpl. Using this strategy, the logical names will be the same as our Java class and property names.
If we want to deviate from this strategy for a specific entity, we can use annotations to make those customizations. We can use the @Table annotation to customize the name of an @Entity. For a property, we can use the @Column annotation:

```
@Entity
@Table(name = "Customers")
public class Customer {

    @Id
    @GeneratedValue
    private Long id;

    private String firstName;

    private String lastName;

    @Column(name = "email")
    private String emailAddress;

    // getters and setters

}
```

Using this configuration, the logical names for the Customer entity and its properties would be:

```
Customer -> Customers
firstName -> firstName
lastName -> lastName
```

```
emailAddress -> email
```

4. Physical Naming Strategy

Now that we configured our logical names, let's have a look at our physical names.

Hibernate uses the Physical Naming Strategy to map our logical names to a SQL table and its columns.

By default, the physical name will be the same as the logical name that we specified in the previous section. If we want to customize the physical names, we can create a custom PhysicalNamingStrategy class.

For example, we may want to use camel case names in our Java code, but we want to use underscore separated names for our actual table and column names in the database.

Now, we could use a combination of annotations and a custom ImplicitNamingStrategy to map these names correctly, but Hibernate 5 provides the PhysicalNamingStrategy as a way to simplify this process. It takes our logical names from the previous section and allows us to customize them all in one place.

Let's see how this is done.

First, we'll create a strategy that converts our camel case names to use our more standard SQL format:

```
public class CustomPhysicalNamingStrategy implements
PhysicalNamingStrategy {

    @Override
    public Identifier toPhysicalCatalogName(final
Identifier identifier, final JdbcEnvironment jdbcEnv) {
        return convertToSnakeCase(identifier);
    }

    @Override
    public Identifier toPhysicalColumnName(final
Identifier identifier, final JdbcEnvironment jdbcEnv) {
        return convertToSnakeCase(identifier);
    }

    @Override
```

```
    public Identifier toPhysicalSchemaName(final
Identifier identifier, final JdbcEnvironment jdbcEnv) {
        return convertToSnakeCase(identifier);
    }

    @Override
    public Identifier toPhysicalSequenceName(final
Identifier identifier, final JdbcEnvironment jdbcEnv) {
        return convertToSnakeCase(identifier);
    }

    @Override
    public Identifier toPhysicalTableName(final
Identifier identifier, final JdbcEnvironment jdbcEnv) {
        return convertToSnakeCase(identifier);
    }

    private Identifier convertToSnakeCase(final
Identifier identifier) {
        final String regex = "([a-z])([A-Z])";
        final String replacement = "$1_$2";
        final String newName = identifier.getText()
          .replaceAll(regex, replacement)
          .toLowerCase();
        return Identifier.toIdentifier(newName);
    }
}
```

Copy

Finally, we can tell Hibernate to use our new strategy:

```
hibernate.physical_naming_strategy=com.baeldung.hibernate
.namingstrategy.CustomPhysicalNamingStrategy
```

Copy

Using our new strategy against the Customer entity, the physical names would be:

```
Customer -> customers
firstName -> first_name
lastName -> last_name
emailAddress -> email
```

Copy

5. Quoted Identifiers

Because SQL is a declarative language, the keywords that form the grammar of the language are reserved for internal use, and they cannot be employed when defining a database identifier (e.g., catalog, schema, table, column name).

Manual Escaping Using Double Quotes

The first option to escape a database identifier is to wrap the table or column name using the double quotes:

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

    @Id
    @GeneratedValue
    private Long id;

    @Column(name = "\"catalog\"")
    private String catalog;

    @Column(name = "\"schema\"")
    private String schema;

    private String name;

    //Getters and setters
}
```

Copy

Manual Escaping Using the Hibernate-specific Backtick Character

Alternatively, we can also escape a given database identifier using the backtick character:

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

    @Id
    @GeneratedValue
    private Long id;

    @Column(name = "`catalog`")
    private String catalog;

    @Column(name = "`schema`")
    private String schema;

    @Column(name = "`name`")
    private String name;

    //Getters and setters
}
```

Copy

Global Escaping Using Hibernate Configuration

Another option is to set the hibernate.globally_quoted_identifiers property to true. This way, Hibernate is going to escape all database identifiers. As a result, we don't have to quote them manually.
In order to use quoted identifiers in CustomPhysicalNamingStrategy, we need to explicitly use the isQuoted() method when creating a new Identifier object:
`Identifier.toIdentifier(newName, identifier.isQuoted());`