The Hibernate DSL User Guide

Outline

1 What is HEDL?

HEDL is a Domain-Specific Language (DSL) that can be used to create Object Relational Mappings (ORMs) based on the Java Persistence API (JPA). HEDL can be used in conjunction with Hibernate or any other JPA implementation (e.g., EclipseLink). HEDL supports entities, properties (read-only, read-write, unique), enumerations and uniqueness constraints over multiple properties. One can define 1:1, 1:N and N:M references. Inverses are also available. In addition, documenting entities, properties and enumerations in Javadoc style is supported by HEDL.

The HEDL DSL comes with a builder that automatically transforms the DSL document to Java entity classes and Data Access Object (DAO) classes. The DAOs contain many default methods for the creation, deletion and lookup of entity objects. HEDL supports to add custom DAO methods by subclassing the generated files. Code that is generated by HEDL can be used both in managed environments (e.g., within an application server) or in unmanaged environments.

The average LOC ratio between the DSL and Java code is 1:100, which means that for every line in the .hedl file an average of 100 lines of Java code is generated.

This documentation contains many sections, each describing a particular aspect on how to use HEDL. If you find yourself in a situation where some information is missing, don't hesitate to contact our support team at .

support@devboost.de

2 How do I XYZ?

2.1 How do I initialize a new HEDL file?

To create a new HEDL model, create a file with the file extension `.hedl` inside a Java source folder. You can do that inside any existing Java project. It is good practice to place the `.hedl` file in a Java package, because this package is used as base package for the generated code.

Within Eclipse, you can also use existing wizards to create an empty or an example `.hedl` file (`File -> New -> Other... -> Empty HEDL Entity Model` , or `HEDL Example Entity Model`).

2.2 How do I create a new entity?

To create a new entity, just enter the name of the entity followed by a `{...}` block:

```
Producer {
    String name;
}
```

Inside the block, you can add the properties of the entity by stating the property's type followed by its name and a semicolon. Property types can be primitive or can refer to other properties. Currently the following primitive types are supported:

- Char
- String
- LongString
- Int
- Long
- Bool
- BigDecimal
- Date
- Double
- Byte
- Blob (byte array)
- Clob (very long strings)

2.3 How do I extend an entity?

You may use the keyword `extends` to inherit properties from another entity.

```
SupplierItem extends Item {
}
```

2.4 How do I set details like readonly or nullable of a property?

Properties can be augmented with a modifier such as `readonly` or `nullable` . You can use code completion (`Ctrl + Space`) to select from a list of available modifiers.

```
Supplier {
    readonly String name;
    nullable Warehouse warehouse;
}
```

The following modifiers are currently supported:

Modifier	Description
`readonly`	For read only properties. Properties which are read only must be passed when constructing a new entity.
`nullable`	For properties that can be null.
`unique`	For properties that must be unique.
`all`	Defines that all operations (detach, merge, persist, refresh, remove) on this element are applied to the objects referenced by this property.
`detach`	Defines that objects referenced by this property are detached when this object is detached.
`merge`	Defines that objects referenced by this property are merged when this object is merged.
`persist`	Defines that objects referenced by this property are saved when this object is saved.
`refresh`	Defines that objects referenced by this property are refreshed when this object is refreshed.
`remove`	Defines that objects referenced by this property are removed when this object is removed.
`eager`	Defines that objects referenced by this property are fetched when this object is fetched from the database.
`lazy`	Defines that objects referenced by this property are fetched when this property is accessed rather than when the object is fetched from the database.

If neither `detach` nor `merge` nor `persist` nor `refresh` nor `remove` is given, the cascade type `all` is used. This implies that objects referenced by the property are deleted when the entity itself is deleted.

2.5 How do I define one-to-many, many-to-one, and many-to-many relationships?

You can use the `*` symbol to define a many-to-one relationship by putting the `*` symbol before the property name:

```
OrderedItem {
    Item *item;
}
```

You can also define a one-to-many relationship by putting the `*` symbol after the property name:

```
OrderedItem {
    Item item*;
}
```

Many-to-many relationships are defined by using a `*` symbol both before and after the properties name:

```
OrderedItem {
    Item *item*;
}
```

2.6 How do I define uniqueness constraints over multiple properties?

You can use the keyword `unique` followed by a list of property names inside an entity definition to define that combinations of these properties must be unique.

```
Customer {
    String firstName;
    String lastName;
    String address;
    unique (firstName lastName address)
}
```

2.7 How do I define enumerations?

Use the keyword `enum` to define an enumeration and list the literals in its body.

```
enum WarehouseType {
    TYPE1
    TYPE2
}
```

2.8 How do I document entities?

You can add documentation to an entity in Javadoc style (starting with `/**` and ending with `*/`). The same applies to documenting properties, enumerations and enumeration literals.

```
/** A Producer is someone who creates items. */
Producer {
    /** This is the producers name. */
    String name;
}
```

2.9 How do I add comments and TODO items?

HEDL supports single line comments (Java style, starting with `//`) to add notes or `TODO` items. `TODO` items will be automatically recognized by Eclipse and appear in the tasks view.

2.10 How do I configure source folder locations and other global options?

A set of options can be set in the beginning of a `.hedl` file. For example:

```
testSourceFolder = "/my.company.project/src-test"
```

configures HEDL to put generated unit test classes into the folder `src-test` of the project `my.company.project` .

The following options are currently supported:

Option	Description
`daoSourceFolder`	The folder where to store the generated DAO classes.
`entitySourceFolder`	The folder where to store the generated entity classes.
`testSourceFolder`	The folder where to store the generated unit tests.
`customSourceFolder`	The folder where to store classes that can be customized.
`preserveTableNames`	If true, table names will not be converted to lower case (which is default).
`preserveColumnNames`	If true, column names will not be converted to lower case (which is default).
`defaultCascadeType`	A list of cascade types that apply to all properties where no explicit cascade type is set (e.g., 'persist merge').

2.11 How can I let entities implement custom interfaces?

You can let an entity implement existing Java interfaces by using the keyword `implements` . Use code completion (`Ctrl + Space`) to obtain a list of all interfaces available in the class path of your project.

```
Producer implements my.company.commons.Named {
    String name;
}
```

2.12 How can I implement pre/post update/persist methods?

Entities that require code to be executed before or after an the entity is updated or persisted can be annotated with the following annotations:

- @PrePersist
- @PreUpdate
- @PostPersist
- @PostUpdate

The code that shall be executed must then be placed in a class called EntityUpdater which is generated if it does not exist, but which is not overridden and can therefore be customized. For each entity and each annotation mentioned above an empty implementation is provided by an abstract super class that is extended by the EntityUpdater class. These methods need to be overridden in the EntityUpdate class.

2.13 How can I refer to existing entities defined with plain Hibernate/JPA?

If Hibernate or another JPA implementation is already used in your project, you can use HEDL to extend your existing data model. To do so, refer to Java classes that represent entities as type of a property. Use code completion (`Ctrl + Space`) to obtain a list of all classes available in the class path.

```
Customer {
    my.company.project.legacy.Account account;
}
```

Currently, fully qualified class names must be used to reference existing entity classes.

2.14 How can I customize the generated DAO code?

All classes inside the package `my.company.project.custom` are available for modification. They are neither deleted nor overwritten when the `.hedl` file is changed.

2.15 How can I generate Hibernate/JPA code?

The code is generated automatically each time you save a `.hedl` file after a modification. Make sure that `Project -> Build Automatically` is enabled.

2.16 How can I generate JUnit test code?

How to configure source folder locations Test code (JUnit classes) is generated along with the production code. You can specify alternative locations for test code by setting the `testSourceFolder` . Consult the section to learn how to specify source folders in HEDL.

2.17 How do I use HEDL in a managed environment?

To use the code generated by HEDL in a managed environment (e.g., an application server), just create a new `OperationProvider` instance and pass the `EntityManager` that is provided by your application server.

2.18 How do I use HEDL in an unmanaged environment?

To use the code generated by HEDL in an unmanaged environment, just use the main `DAO` class. This class is named according to your `.hedl` file. The main `DAO` will automatically create an `EntityManager` and encapsulate calls to data access methods in transactions. Since the `DAO` creates an `EntityManagerFactory' which is a heavy-weight` object, make sure to create only one main `DAO` in your application.

2.19 How can I use HEDL if I have an existing (legacy) database schema?

HEDL allows to adjust the code that is generated from the entity model to fit to an existing database schema. This mainly requires to replace the default table and column names that are derived automatically by HEDL. To replace these names with the names used in an existing schema, the following annotations be added to entities in the `.hedl` file:

Annotation	**Description**
`@Table("legacy_table_name")`	Specifies the name of the table where the entity must be stored.
`@IdColumn("legacy_id_column")`	Specifies the name of the identity column (must be of type INT)

For example, if instances of `LegacyEntity` are stored in a table called `le_table` which contains the IDs in column `le_id` , use the following annotations:

```
@Table("le_table")
@IdColumn("le_id")
LegacyEntity {
}
```

The following annotations be added to properties in `.hedl` files to specify table and column names for relations:

Annotation	**Description**
`@Column("legacy_column")}`	Specifies the name of the column to store the property in.
`@JoinTable("other_legacy_table")`	Specifies the name for a join table.
`@JoinColumn("other_entity")`	Specifies the name for a join column.
`@InverseJoinColumn("inverse_column")`	Specifies the name for an inverse join column.

```
LegacyEntity {
    @Column("leg_fld")
    nullable Int legacyField;
    @JoinTable("leg_ent_1n_other")
    @JoinColumn("ent_1")
    @InverseJoinColumn("other_n")
    OtherEntity oneToManyRef*;
    @JoinTable("leg_ent_mn_other")
    @JoinColumn("ent_m")
    @InverseJoinColumn("other_n")
    OtherEntity *manyToManyRef*;
}
```

The following annotations be added to properties in `.hedl` files to specify table and column names for collection properties that have a primitive type:

Annotation	Description
`@Table("legacy_collection_table")`	Specifies the name for a collection table.
`@JoinColumn("legacy_join_column")`	Specifies the name for the join column.
`@Column("value")`	Specifies the columns name for primitive collection properties.

```
LegacyEntity {
    @Table("leg_other_col")
    @JoinColumn("ent")
    @Column("value")
    String collection*;
}
```

2.20 How can I add hooks to be notified when entities are created or modified?

HEDL can notify the application code when an entity is created or modified. To enable such notifications, the following annotations be added to entities in the `.hedl` file:

Annotation	Description
`@OnCreate`	Modifies the generated code to call class EntityUpdater.onCreateEntityName() when an instance of the entity is created.
`@OnChange`	Modifies the generated code to call class EntityUpdater.onChange() when a property of the entity is modified.

2.21 How do I use lists of primitive types?

To create a new property that can hold a list of primitive values, just enter a star after the name of property:

```
Document {
    String names*;
}
```

You can use lists for properties properties of the following types:

- Char
- String
- LongString
- Int
- Long
- Bool
- BigDecimal
- Date
- Double

You cannot use lists of the bytes, because a list of bytes needs to be represented by a Blob. Also lists of Blobs are not supported.