Getting Started with the Hibernate DSL

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.

This tutorial introduces into HEDL and shows, how HEDL can be installed and used with a running example. For further hints on what HEDL is good for and how it can be used and customized, feel to also consult the HEDL documentation. 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 to install HEDL

This section explains how HEDL is installed within a distribution of the Eclipse IDE. If you installed HEDL already, you can skip this section.

2.1 Required Software to Run HEDL

As HEDL is implemented as an extension (plugin) of the Eclipse IDE, Eclipse is required for the usage of HEDL. You can download Eclipse from http://www.eclipse.org/ or download a distribution having HEDL installed . Feel free to use any Eclipse distribution you like. However, as HEDL bases on Java, make sure that your distribution includes at least the Java Development Tools (JDT).

2.2 Installing HEDL in Eclipse

If you downloaded a complete Eclipse distribution including HEDL from http://www.hibernate-dsl.com/download.html , you don't need to install further software at all. However, most of the times you want to install HEDL within your personalized, existing Eclipse IDE. Generally, two possibilities to install HEDL into your Eclipse exist:

2.2.1 Installing HEDL via Eclipse Market Place

Within Eclipse, select the menu option `Help > Eclipse Marketplace...`

Enter `HEDL` into the search dialog and click on the `Go` button.

HEDL should appear in the search results:

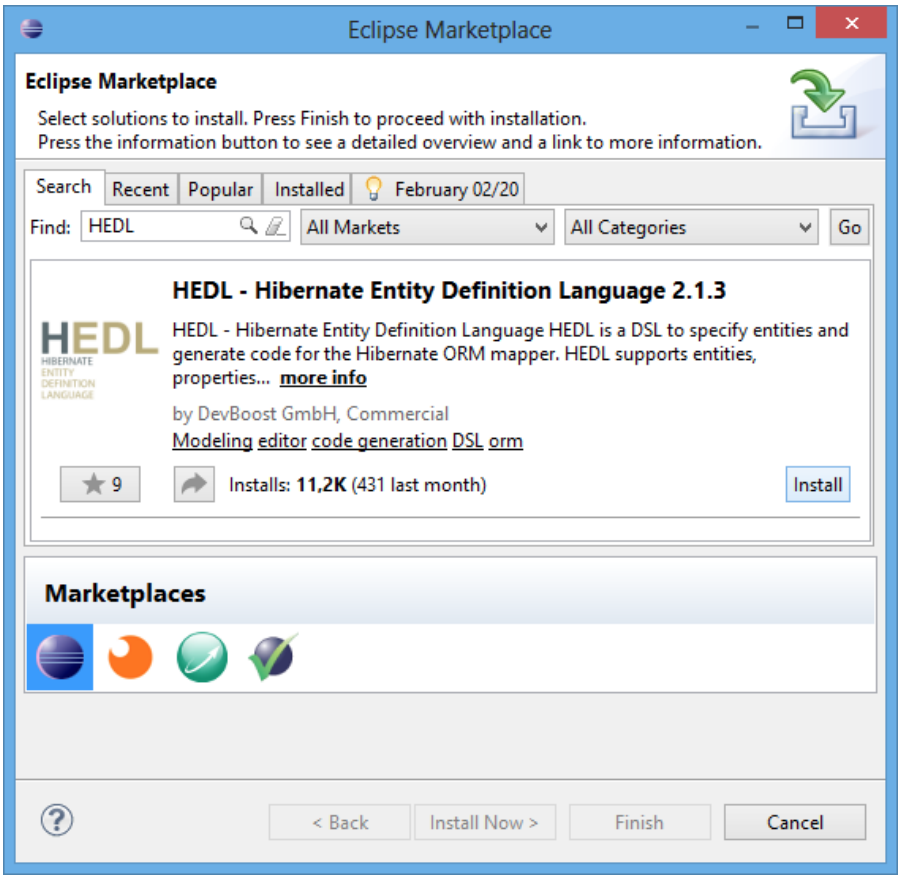

Figure 1 - Dialog to install HEDL via Eclipse Marketplace

Click on the `Install` button and follow the instructions to install HEDL.

2.2.2 Installing HEDL via HEDL Update Site

Within Eclipse, select the menu option `Help > Install New Software...`

Enter the link to the HEDL Update Site ( ) into the URI text field and type `ENTER` .

http://www.hibernate-dsl.com/update/

Select `Hibernate Entity Definition Language (HEDL)` from the items to install, click on the `Next` button and follow the instructions to install HEDL:

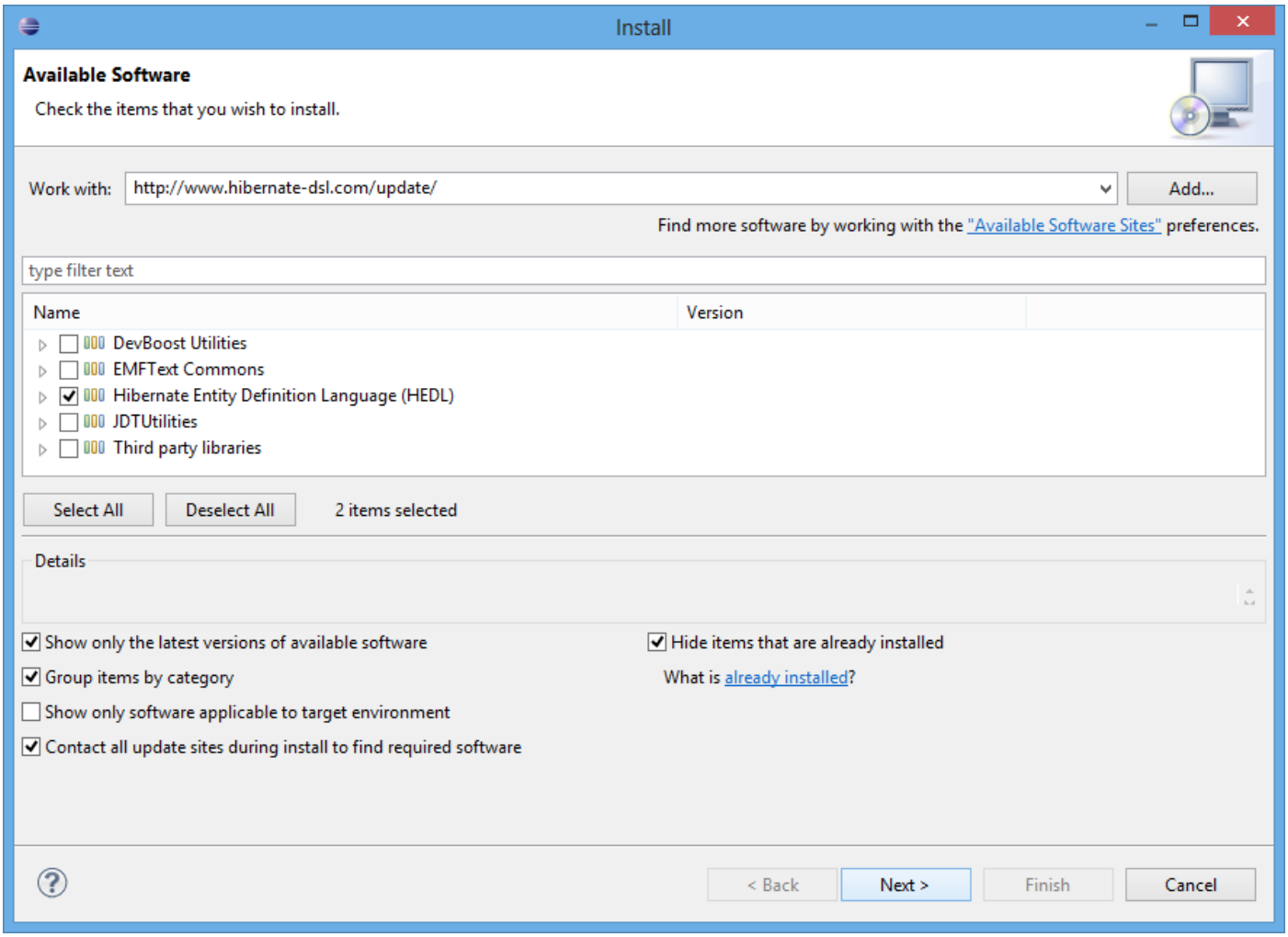

Figure 2 - Dialog to install HEDL via HEDL Update Site

2.3 Getting Your Personal HEDL License

To use HEDL you will need a HEDL license. Once you have installed HEDL and restarted your Eclipse distribution, a respective dialog will appear to enter a HEDL license.

You can either apply for a trial license or buy a regular license. Students and research institutions can apply for a HEDL license for free. Visit to learn more about HEDL licenses.

the HEDL website

To apply for a trial license you can use the opened dialog in Eclipse: Select the option `Request free trial license` and click on the `Next` button.

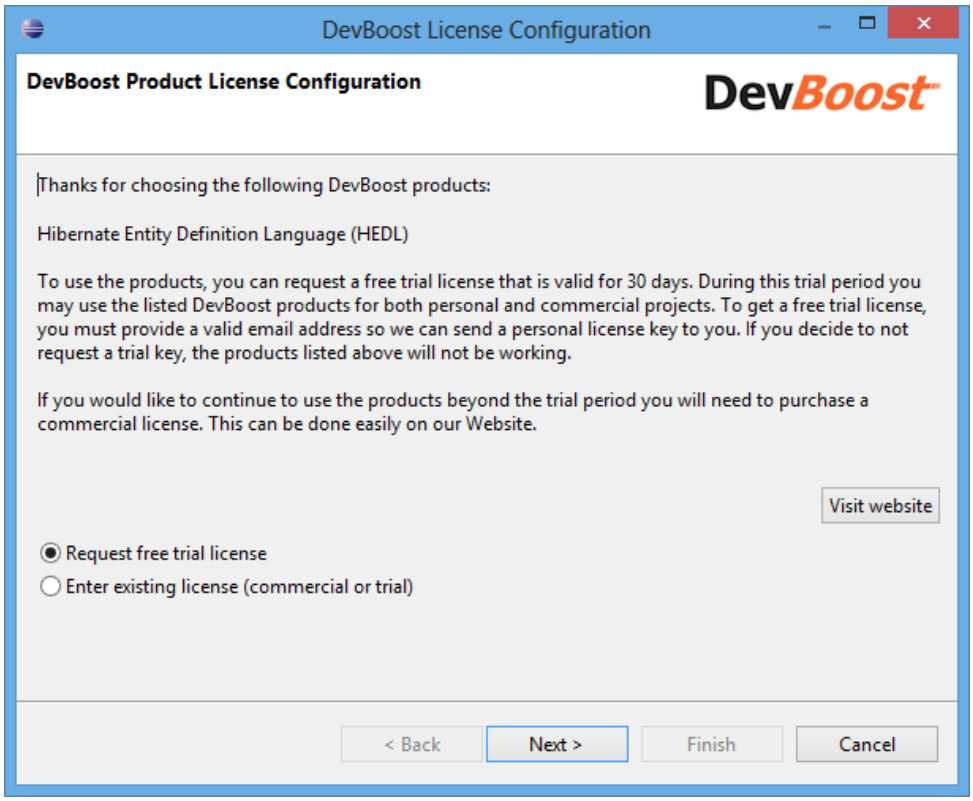

Figure 3 - Dialog to request a HEDL trial license (part 1)

Enter your name and your email address and click on the `Next` button.

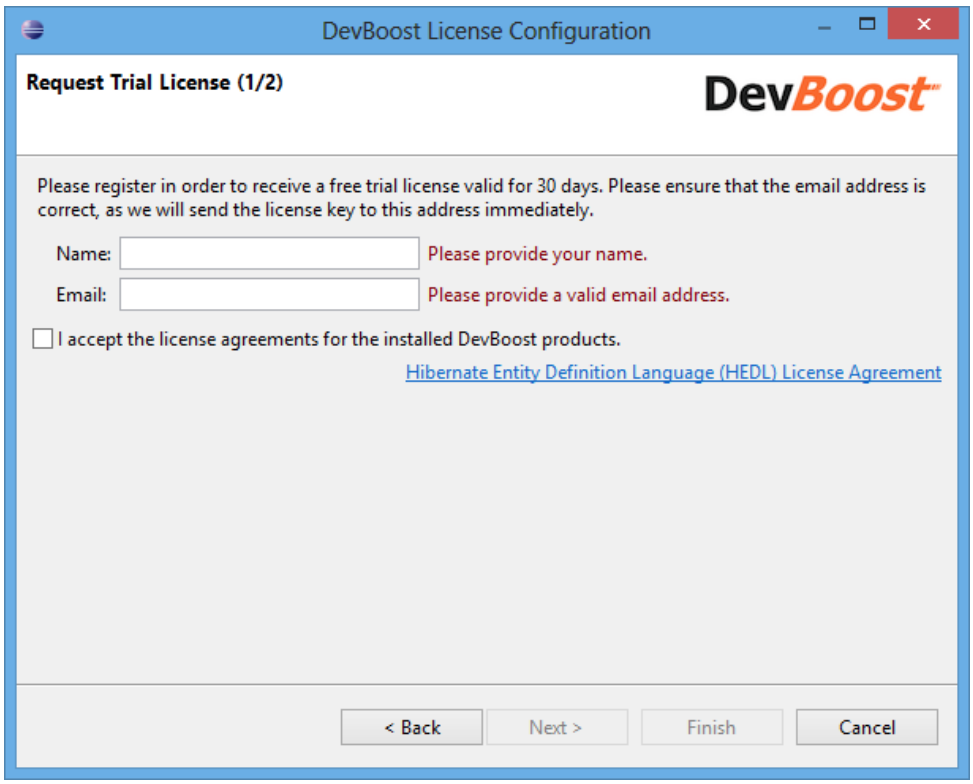

Figure 4 - Dialog to request a HEDL trial license (part 2)

You should receive an email containing a trial license key you can enter into the following screen:

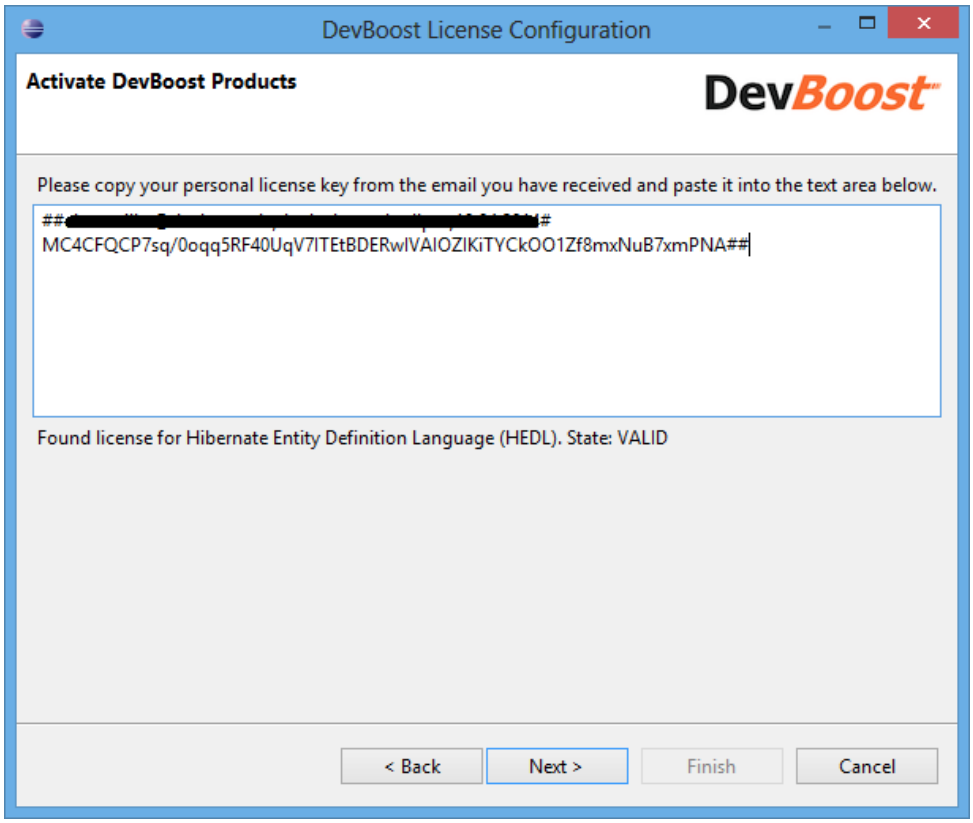

Figure 5 - Dialog to enter HEDL license

Proceed with the dialog by clicking on the `Next` button. Now you should be able to use an try your HEDL distribution.

3 A Running Example

This tutorial uses a running example for a web shop application to explain functionalities of HEDL. The entities of the example comprise `Items` having `Producers` and `Prices` (called `PriceSet` in the example). We will create these entity classes in the following using HEDL instead of implementing them manually.

To get used to the example and get an overview of the features and possibilities of HEDL you can create the complete example using the wizard `File > New > Other... > HEDL > HEDL Example Project`. However, in the following we will create the same example step by step to get used to all the features and generated classes in a sensible order. Besides, our example also differs a bit from the generated example, as we explain only the entity classes required to explain the features of HEDL within this tutorial.

3.1 Creating a HEDL Project

First, we create a new project for our web shop application. We use the wizard `File > New > Java Project`. You might also use a plugin project instead if you like to manage dependencies between the projects of your web shop application with OSGI dependencies. However, to keep things simple, we use a Java project here.

Enter a project name for your project (e.g., `org.hibernate_dsl.example`) and click on the `Finish` button. Afterwards, your project explorer should look as follows.

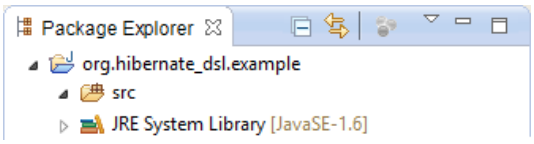

Figure 6 - The newly created empty Java project

Next, we create create our HEDL Entity Model, the textual description of our entity classes that shall be generated by HEDL. The HEDL developer guidelines propose to place this `.HEDL` file within the root package of our project's source folder. Thus, we select the `src` folder first and create a new package via the context menu first (`New > Package`). You may use the same name as for the project for the package (e.g., `org.hibernate_dsl.example`).

Next, we create the HEDL Entity Model (the `.HEDL` file). Select the newly created package in the package explorer and use the wizard `File > New > File`:

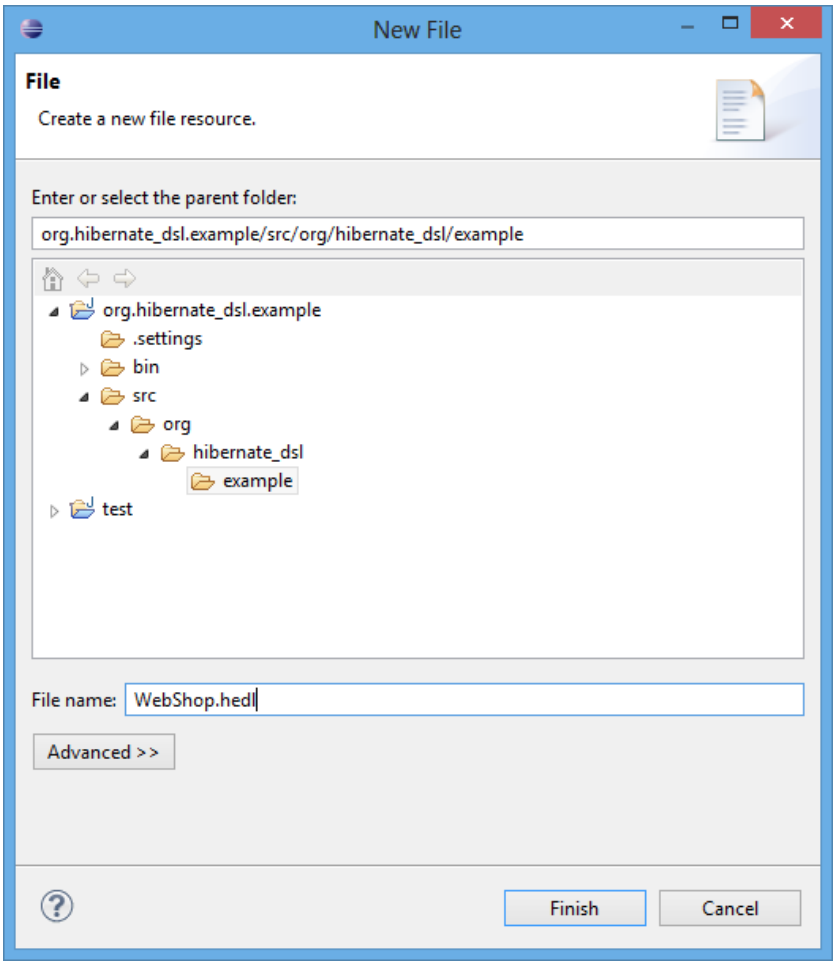

Figure 7 - Dialog to create a new HEDL Entity Model

Enter the name of the HEDL Entity Model (e.g., `WebShop.hedl`), and click on the `Finish` button:

Afterwards, your example project should look as follows:

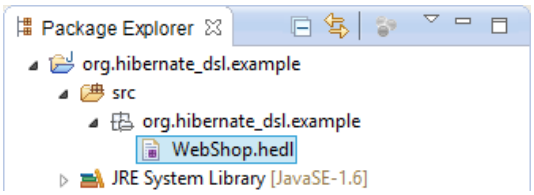

Figure 8 - The example project containing the created HEDL Entity Model

3.2 Creating an Entity using HEDL

Now, we can start with creating our first entity. We start with a very simple entity class, the `Producer` entity, having only one property, a `name` . Open the `WebShop.hedl` file and enter the following code:

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

That's all. Once you safe the file, you may notice that HEDL generates a set of Java files being contained in several packages. After saving the `WebShop.hedl` file, you project should look as follows:

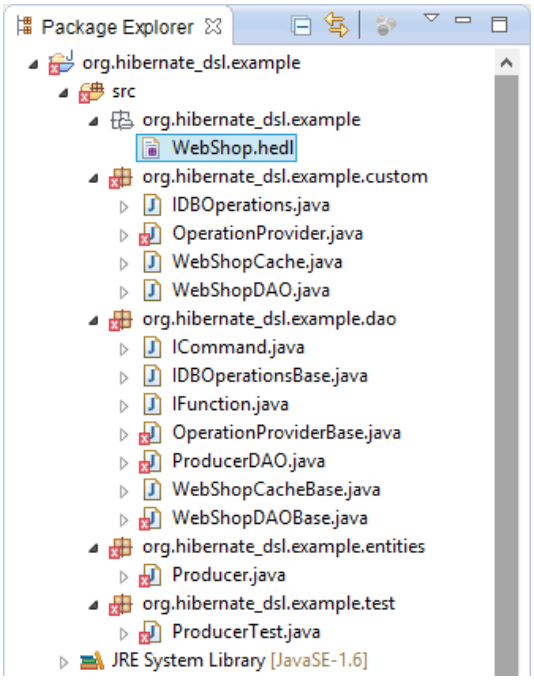

Figure 9 - The example project after creating our first entity

3.3 Having a Look at the Packages and Classes generated by HEDL

Lets look at these packages and their generated content:

3.3.1 The Entities Package

First, of course, HEDL generates entity classes for the entities specified in the `.HEDL` file. Thus, HEDL generates a package `org.hibernate_dsl.example.entities` , containing the class `Producer.java` , using JPA annotations for the entity definition, as well as providing getter and setter methods for the entity's fields.

```
@Entity
@Table(name = "producer")
public class Producer {
    @Id
    @GeneratedValue(strategy=GenerationType.AUTO)
    private int id;
    @Column(name="name")
    private String name;
    /* ... */
    /**
    * Returns the value of property 'name'.
    */
    public String getName() {
        return name;
    }
    /**
    * Sets the value of property 'name'.
    */
    public void setName(String newValue) {
        this.name = newValue;
    }
    /* ... */
}
```

Once generated for the first time, the class may contain errors, as required dependencies to respective JAR files are not available (e.g., the javax.persistence package). If you add the required libraries to the example project, these errors will go away.

3.3.2 The DAO package

Next, HEDL generates a package called `org.hibernate_dsl.example.dao` , containing database access objects (DAO) for all entities defined in the `.hedl` file. Right now, the package contains the class `ProducerDAO` , being the DAO for `Producer` entities.

Besides, the `dao` package will contain the interface `IDBOperationsBase` that eases the daily work with entity classes, by encapsulating the DAO functionality behind more usable methods such as `createProducer(...)` and `deleteProducer(...)` :

```
public interface IDBOperationsBase {
    public void flush();
    public Producer createProducer(final String name);
    public Producer getProducer(final int id);
    public List<Producer> getAllProducers();
    public List<Producer> searchProducers(final String _searchString, final int _maxResults);
    public void delete(final Producer entity);
    public void deleteProducers(final List<Producer> entities);
    public int countProducers();
    public void merge(Producer entity);
}
```

Apart from the interface, the `dao` package provides three implementations for this interface. The first one is called `OperationProviderBase` and implements the interface `IDBOperationsBase` for usage within a managed environment (e.g., within an application server), where the `EntityManager` and its transactions are provided by the application server environment.

The second implementation of the interface `IDBOperationsBase` is called `WebShopBase` and implements the `IDBOperationsBase` interface for usage in an unmanaged environment. Thus, it creates its own `EntityManager` based on a given context class and encapsulates all DAO operations within transactions.

The Section on how to integrate HEDL entities into real applications explains the differences between the different `IDBOperationsBase` implementations in more detail.

Besides the managed and unmanaged `IDBOperationsBase` implementation, HEDL provides a third implementation called `WebShopCacheBase` . This implementation can be used to encapsulate either of the two other implementations with an additional in memory cache that caches entities within memory, besides storing them in a relational database. Again, consult the Section on how to integrate HEDL entities into real applications for further details on this caching mechanism.

3.3.3 The Custom Package

Although the implementations of `IOperationsBase` provide many usable entity access methods, it is likely that some of these methods must be adapted for your project (e.g., to adapt the entity properties being considered in the generated `searchXYZ(...)` methods).

Thus, the generated `org.hibernate_dsl.example.custom` package provides extensions of the `IDBOperationsBase` interface and the existing instantiating classes that can be modified manually and will not be overwritten once the `.hedl` file will be modified or extended.

The interface `IDBOperations` extends the interface `IDBOperationsBase` , whereas the classes `OperationsProvider` , `WebShopCache` , and `WebShopDAO` extend the classes `OperationsProviderBase` , `WebShopCacheBase` , and `WebShopDaoBase` . The Section on how to integrate HEDL entities into real applications contains more details about how the different `IDBOperations` implementations can be instantiated for their usage within enterprise applications.

3.3.4 The Test Package

Finally, HEDL also generates a test package (`org.hibernate_dsl.example.test` , containing a JUnit test class for each entity DAO being generated into the `org.hibernate_dsl.example.dao` package. Right now, this package should contain exactly one test class, called `ProducerTest` . As this class has dependencies to the JUnit 4 annotations, you must add the JUnit JAR to your projects classpath to avoid compilation errors.

3.3.5 Configuring the Destination for all generated Packages

Right now, HEDL stores all generated classes in your `src` folder, which works, but might interfere with your development guidelines (e.g., it is common to separate code into folders called `src` and `src-gen`). Thus, HEDL allows to configure the destinations for all generated packages and classes. To do so, delete the yet generated HEDL code and add the following statements to the beginning of your `.hedl` file:

```
daoSourceFolder = "src-gen"
entitySourceFolder = "src-gen"
testSourceFolder = "src-test"
```

Now, your example project should look as follows:

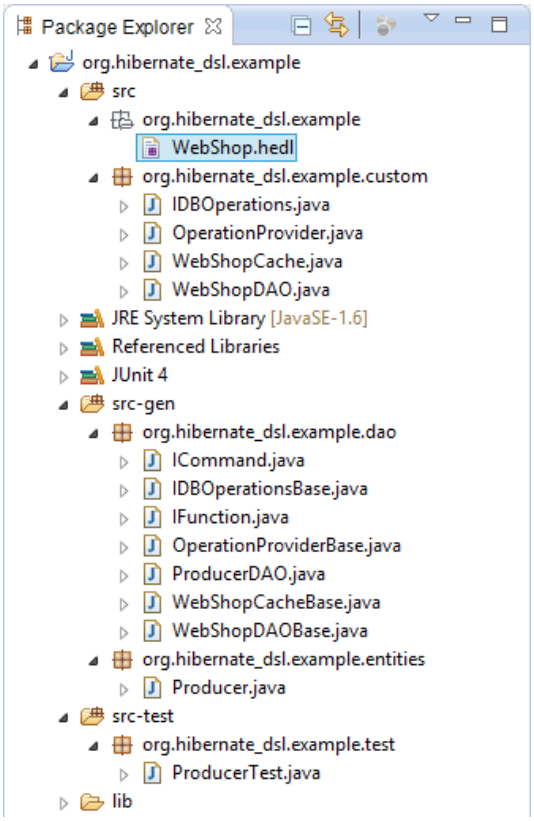

Figure 10 - The example project after configuring source folder destinations

3.4 Use of Primitive Types

Now, it is time to define further entities for our web shop application. First, we define a second entity using properties having primitive types. We define the `PriceSet` entity, that defines the `sellingPrice` of an `Item` as well as the last `Date` when this price has been updated:

```
PriceSet {
    Double sellingPrice;
    Date lastUpdate;
}
```

Besides `Strings` , `Doubles` , and `Dates` , you can also use integers (`Int`), Booleans (`Bool`), very large Strings (`Clobs`), and byte arrays (`Blobs`). Please consult the HEDL documentation for a complete list of all supported primitive types in HEDL.

3.5 One-to-One Relationships

Next, we define the `Item` entity:

```
Item {
    String name;
    PriceSet priceSet;
}
```

Initially, `Items` get a `name` and a `priceSet` . Please note that one-to-one relationships to other entities are defined similar to primitive type properties. The code generated for this one-to-one relationship by HEDL looks as follows:

```
public class Item {
    /* ... */
    @OneToOne(cascade={CascadeType.ALL})
    @JoinColumn(name="priceset", nullable=false)
    private PriceSet priceSet;
    /* ... */
}
```

3.6 Nullable, Readonly and Unique Properties

Next, we use modifiers to declare special constraints on respective properties. For the `priceSet` property of an `Item`, we declare that the `priceSet` can be set only once, when an `Item` entity is created (`readonly`) and that the `priceSet` of an `Item` can be `null` (`nullable`). HEDL will take care of this modifiers by generating the respective annotations for the `Item` entity and its properties:

```
public class Item {
    /* ... */
    @Column(name="name")
    private String name;
    @OneToOne(cascade={CascadeType.ALL})
    @JoinColumn(name="priceset", updatable=false, nullable=true)
    private PriceSet priceSet;
    /* ... */
}
```

For the `name` of a `Producer` we define a uniqueness constraint using the modifier `unique` :

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

Resulting in the following HEDL-generated annotations:

```
public class Producer {
    /* ... */
    @Column(name="name", unique=true)
    private String name;
    /* ... */
}
```

Besides `nullable` , `readonly` , and `unique` further property modifiers in HEDL exist (e.g., to declare the fetch mode as eager or lazy). Please consult the HEDL documentation for a complete list of all property modifiers supported by HEDL.

3.7 One-to-Many and Many-to-One Relationships

As `Producers` produce `Items` , we want to introduce a relationship between these entities. However, now we need a one-to-many or a many-to-one relationship respectively. One-to-many relationships are denoted by a `*` at the end of the property name, many-to-one-relationships are denoted by a `*` at the beginning of the property name:

```
Producer {
    String name;
    Item items*;
}
Item {
    String name;
    readonly nullable PriceSet priceSet;
    nullable Producer *producer;
}
```

However, this configuration does not ensure that the `Producer.items` property is the opposite of the `Item.producer` relation.

To do so, we declare the `Producer.items` as the opposite of the `Item.producer` relation.

```
Producer {
    String name;
    Item items* <= producer;
}
```

Again, HEDL will take care of generating the right JPA annotations for each one-to-many and many-to-one relationship:

```
public class Item {
    /* ... */
    @ManyToOne(cascade={CascadeType.ALL})
    @JoinColumn(name="producer", nullable=true)
    private Producer producer;
    /* ... */
}
public class Producer {
    /* ... */
    @OneToMany(cascade={CascadeType.ALL}, mappedBy="producer")
    private List<Item> items;
    /* ... */
}
```

3.8 Using Enumerations

Besides the use of primitive type properties and entity relationships, HEDL allows for the definition of enumeration types and respective properties within entities. For example, we can define an entity `ItemOrder` having a state of the enumeration type `OrderState:`

```
ItemOrder {
    Item *item;
    OrderState state;
}
enum OrderState {
    ORDERED SHIPPED DELIVERED
}
```

HEDL will generate the respective Java `enum` , as well as the correct annotations for an enumeration property:

```
public enum OrderState {
    ORDERED,
    SHIPPED,
    DELIVERED,
}
public class ItemOrder {
    /* ... */
    @Enumerated(EnumType.STRING)
    @Column(name="state")
    private OrderState state;
    /* ... */
}
```

3.9 Annotating entities to be archivable

In some applications it's needed to mark entities as not used anymore instead of deleting them physically. HEDL offers an automated way to provide such archiving capabilities via the `@Archivable` annotation. If an entity is annotated in such way, HEDL adds another parameter `boolean includeArchivedEntities` to each of its getters. If this parameter is set to `true` it also returns the archived entities, while it doesn't if the parameter is set to `false` . In addition, methods for getting and setting the archived property are generated.

An example for an archivable entity can be defined as follows:

```
@Archivable
Producer {
    unique String name;
    Item items* <= producer;
}
```

Optionally, one can specify the name of the property that is used to indicate whether an entity was logically deleted or not:

```
@Archivable("deleted")
Producer {
    unique String name;
    Item items* <= producer;
}
```

If no name for the property is given, a default name ("archived") is used.

3.10 Entities implementing specific Interfaces

Sometimes its required to let your entities implement certain interfaces. For example, it can be usable to declare your entities as being serializable. You can do so, by using the `implements` keyword in HEDL:

```
Producer implements java.io.Serializable {
    unique String name;
    Item items* <= producer;
}
```

HEDL will copy the respective implementations to the generated entity classes:

```
public class Producer implements java.io.Serializable {
    /* ... */
}
```

3.11 Documenting your Entities with JavaDoc

HEDL also allows for the documentation of your entity classes using JavaDoc:

```
/** Provides the price of an {@link Item} and the Date of its last modification. */
PriceSet {
    /** The selling price of the {@link Item}. */
    Double sellingPrice;
    /** The Date of its last modification. */
    Date lastUpdate;
}
```

The respective JavaDoc comments will be copied to the right places within the generated entity classes:

```
@Entity
@Table(name = "priceset")
/**
* Provides the price of an {@link Item} and the Date of its last modification.
*/
public class PriceSet {
    /* ... */
    /**
    * The selling price of the {@link Item}.
    */
    @Column(name="sellingprice")
    private double sellingPrice;
    @Temporal(TemporalType.TIMESTAMP)
    /**
    * The Date of its last modification.
    */
    @Column(name="lastupdate")
    private Date lastUpdate;
    /* ... */
}
```

4 Using HEDL-Generated Entities within your Application

Generally speaking, HEDL generates the complete persistence layer of your enterprise application. Thus, all you have to do is to implement the logic (or service) and possibly the UI layer of your application. But how to integrate the HEDL-generated entities and their DAO objects into your application?

Generally, HEDL provides two different data access object implementations encapsulating the DAO functionality for all entity classes: one class for a managed environment (e.g., an application server) and one class for an unmanaged environment.

4.1 Using HEDL in a managed Environment

see Subsection explaining the generated custom package The DAO for the managed environment is called `OperationProvider` and located within the `de.hibernate_dsl.example.custom` package as it can be extended with customized functionality ( ). To use the `OperationProvider` in a managed environment, you have to retrieve the `EntityManager` of your application server and pass it to the `OperationProvider` via its constructor:

```
public class OperationProvider extends OperationProviderBase implements IDBOperations {
    public OperationProvider(EntityManager em) {
        super(em);
    }
}
```

Afterwards, you can delegate to the `OperationProvider` within all methods of your service layer that require database/entity access.

4.2 Using HEDL in an unmanaged Environment

The DAO for using HEDL in an unmanaged environment is called `WebShopDAO` and also located within the `de.hibernate_dsl.example.custom` package as it can be extended with customized functionality. The `WebShopDAO` encapsulates all DAO methods within transactions and creates its own `EntityManager` internally, based on a context class that allows to access the `persistence.properties` of a JPA environment:

```
public class WebShopDAO extends WebShopDAOBase {
    public WebShopDAO(Class<?> contextClass) {
        super(contextClass);
    }
    /* ... */
}
```

4.3 Using a HEDL DAO with a cache

Besides the managed and unmanaged DAO, HEDL provides a third implementation of the `IDBOperations` interface. This class is called `WebShopCache` and also located within the `de.hibernate_dsl.example.custom` package. The Code `WebShopCache` can be used to extend each of the other DAOs with an in memory cache. Therefore, it provides a constructor taking another `IDBOperations` instance that can be used to delegate all entity modifying operations, besides their modification in the `WebShopCache` :

```
public class WebShopCache extends WebShopCacheBase implements IDBOperations {
    public WebShopCache(IDBOperations delegate) {
        super(delegate);
    }
}
```

The `WebShopCache` can also be applied for testing purposes. Just pass a `null` value to the `WebShopCache` constructor and all entities will only be stored in memory. This is an ideal setup to execute JUnit and other functional tests for your logic and UI layers.

5 Some Further Hints

For the daily use of HEDL you might also know some further hints and tricks that ease the usage of HEDL.

5.1 HEDL Help

During the usage of HEDL it can be useful to have its documentation directly available within Eclipse. Thus, a HEDL Help view exists that provides the HEDL documentation at any time you need it.

The HEDL help can be opened by using the menu option `Window > Show View > Other... > HEDL Help > HEDL Help`

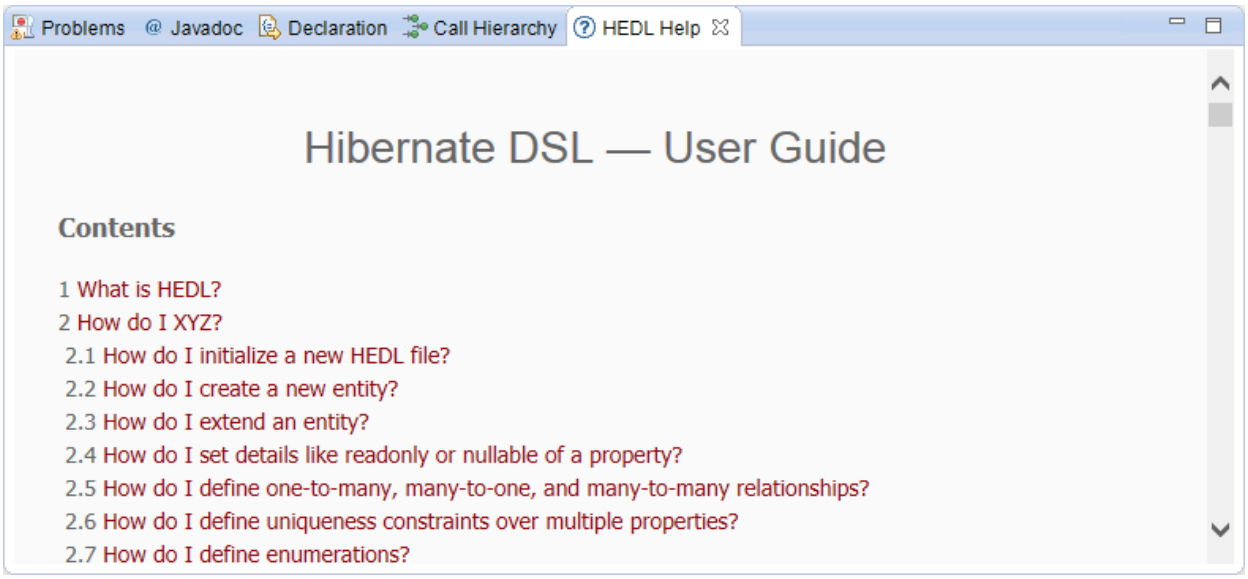

Figure 11 - The NatSpec Help View

6 Summary

This tutorial showed, how HEDL can be installed in Eclipse and how HEDL can be applied to define enterprise entities in a textual manner and generate the entity, as well as their data access object (DAO) and testing code in an automated manner. Besides the functionalities shown in this tutorial, HEDL has features that would increase the size of this tutorial even more. Thus, feel free to explore the HEDL documentation for all functions and features supported by HEDL.

Besides, your feedback is highly appreciated in order to improve HEDL. Feel free to suggest new features, report encountered problems or simply share your HEDL experience with us. Just . We love to hear from you.

send an e-mail to the HEDL support team

7 Table of Figures