Entity Engine Guide

Written By: David E. Jones
Edited by: Les Austin and Pawel H Debski

Table of Contents

Related Documents:

Entity Engine Configuration Guide

Introduction

The Open For Business Entity Engine is a set of tools and patterns used to model and manage entity specific data. In this context an entity is a piece of data defined by a set of fields and a set of relations to other entities. This definition comes from the standard Entity-Relation modeling concepts of Relational Database Management Systems. The goal of the entity engine is to simplify the enterprise wide use of entity data. This includes definition, maintenance, quality assurance, and development of entity related functionality.

The entity engine uses a number of Business Tier and Integration Tier patterns that will be recognized by most enterprise software programmers. Many Presentation Tier patterns are also used in the Open For Business project, but only in the servlet controller, not in the entity engine. The patterns used in the Entity Engine include: Business Delegate, Value Object, Composite Entity (variation), Value Object Assembler, Service Locator, and Data Access Object. Implementations of other patterns and more complete implementations of these Patterns are planned.

Descriptions of these patterns are all available in the book "Core J2EE Patterns" by Alur, Crupi, and Malks, published by Sun. You can also find information on these patterns at the site that contains the early work of the authors before the published book was finished. See http://developer.java.sun.com/developer/restricted/patterns/J2EEPatternsAtAGlance.html. Note that you need a Java Developers Connection account to view this information.

In addition there are a number of patterns used from the book "EJB Design Patterns" by Floyd Marinescu of TheServerSide.com. These include Data Transfer HashMap and Generic Attribute Access. For unique primary key generation we use an original pattern we like to call the "Ethernet Key Generation" pattern because it uses a collision detection mechanism to insure that multiple servers can use a single database to get banks of unique keys in a database independent way.

The primary goal of the entity engine is to eliminate the need for entity specific persistence code in as many areas of a transactional application as possible. Granted that this sort of abstraction is a different issue for reporting and similar systems, but for transactional systems such as are used on a day to day basis in all businesses, an entity engine can save a great deal of development effort and dramatically reduce persistence related bugs in the system. These types of applications include everything from ecommerce to accounting to inventory and warehouse management to human resources and so on. These tools can be useful for reporting and analytical systems, but really aren't meant to allow for the wide variety of custom queries that often take place in such tools.

In order to achieve having as little entity specific code as possible, all value objects are generic, using a map to store the fields values of the entity instance by name. The get and set methods for the fields take a String with the fieldName in it which is used to verify that the field is part of the entity, and then either get or set a value as desired. The danger of this flexibility is curtailed using a contract between the entity engine and the application; this is contained in a special XML file.

Instead of writing entity specific code, entity definitions are read from an XML file and used by the entity engine to enforce a set of rules between the application and the data source, be it a database or some other source. These XML entity definitions specify the names of all of the entities and their fields along with which database tables and columns they correspond to. They are also used to specify a type for each field which is then looked up in the field types file for the given data source to find the Java and SQL data types. Relations between entities are also defined in this XML file. A relation is defined by specifying the related table, the type of relation (one or many) and the keymaps for the relation. A title can also be given to the relation which becomes part of its name to distinguish it from other relations to that specific related entity.

Using the Entity Engine as an abstraction layer, entity specific code can be easily created and modified. Code that uses the Entity Engine APIs to interact with entities can be deployed in various ways so that entity persistence can be done differently without changing the code that interacts with those entities on a higher level. An example of the usefulness of this abstraction is that, by changing a configuration file, an application written using the Entity Engine can switch from hitting a database directly through JDBC to using an EJB server and Entity Beans for persistence. The same code could also be used for custom data sources like legacy systems over HTTP or messaging services through a bit of custom coding within the same framework.

Entity Modeling

Entity Modeling Files & Locations

The first thing to do when starting work with a new entity is to define or model that entity. In the OFBiz Entity Engine this is done through two XML files, one for entity modeling and the other for field type modeling. There are links to the XML DTDs for these documents in the Related Documents section above. The reason that these two files are separated is that field type definitions can be database specific. In other words, for a given entity model definition various field type definitions may exist for different databases. When a persistence server is defined the field type model XML file to be used for that server is specified.

The main entity model XML files for Open For Business can be found in **ofbiz/commonapp/entitydef/**. Originally all of the entities were in the **entitymodel.xml** file, but now they are separated into various files in addition to the **entitymodel.xml** file. They are all named after the following pattern: **entitymodel_*.xml**

The MySQL field type model XML file for Open For Business can be found in **ofbiz/commonapp/entitydef/fieldtypemysql.xml**. There are other database specific field type files for Postgres, Hypersonic, Oracle, et cetera. From the entity model files and the field type files database tables can be created automatically through the checkDataSource routine on the GenericHelper interface. This can be done automatically on startup or through the tools in WebTools.

Loading Seed Data

While tables can be created automatically, data must be loaded from data files. These files can be either SQL scripts or XML Entity Engine files. All of the type information and other pre-loaded information such as statuses, enumerations, geo data, etc., are located in XML Entity Engine files in **ofbiz/commonapp/db/**. These files can be located and loaded automatically by the **install.jsp** page in WebTools. This page looks in the directories specified in the **entityengine.xml** file for a given entity group name and finds all .xml and .sql files. These are listed and confirmation is requested by the page. Clicking on the Yes, Load Now link will cause these files to attempt to be loaded. Error messages will appear in the page as well as on the console or in log files. Data files can also be loaded one at a time by specifying the full path of the .sql or .xml file in the load a single file form. While on the topic, XML Entity Engine files can also be imported and exported through the import & export pages in WebTools.

An Example Entity Definition

As mentioned above an entity consists of a set of fields and a set of relationships to other entities. In the XML entity definitions each of these are specified in addition to attributes of the entity itself such as the entity name, the corresponding table name, the package name for the entity, and meta data about the entity such as the author, a copyright notice, a description, and so forth. Here is an example of an XML entity definition:

```
<entity title="Sample Entity"
        copyright="Copyright (c) 2001 John Doe Enterprises"
        author="John Doe" version="1.0"
        package-name="org.ofbiz.commonapp.sample"
        entity-name="SampleEntity"
        table-name="SAMPLE_ENTITY">
  <field name="primaryKeyFieldOne" col-name="PRIMARY_KEY_FIELD_ONE" type="id"></field>
  <field name="primaryKeyFieldTwo" type="id"></field>
  <field name="fieldOne" type="long-varchar"></field>
  <field name="fieldTwo" type="long-varchar"></field>
  <field name="foreignKeyOne" type="id"></field>
  <prim-key field="primaryKeyFieldOne" />
  <prim-key field="primaryKeyFieldTwo" />
  <relation type="one" rel-entity-name="OtherSampleEntity">
    <key-map field-name="foreignKeyOne" rel-field-name="primaryKeyOne" />
  </relation>
  <relation type="one" title="Self" rel-entity-name="SampleEntity">
    <key-map field-name="primaryKeyFieldOne" />
    <key-map field-name="primaryKeyFieldTwo" />
  </relation>
  <relation type="many" title="AllOne" rel-entity-name="SampleEntity">
    <key-map field-name="primaryKeyFieldOne" />
  </relation>
</entity>
```

This is a pretty simple entity that demonstrates a few small points. The meta data at the top is all optional, and if left unspecified will default to meta data defined for the entire entity model file. For example the "description" element was left out, so the ModelReader will use the description specified at the top of the XML file if one exists. The package-name is used to organize the entities, and specify a default location for any code that would be entity specific. This becomes extremely useful when you have an entity model with hundreds of entities.

Entity and Field Definitions

Notice that while the field primaryKeyFieldOne has a column name specified, none of the other fields do. The col-name and the table-name elements are optional. They can be derived from the field name or entity name through widely used conventions. These conventions dramatically simplify the definition of entities.

Table and column names are written in caps with underscores separating the words like SAMPLE_ENTITY. Entity and field names are written using the Java conventions for class and field names where all letters are lowercase except for the first letter of each word, which is uppercase. Entity names correspond to Java classes so the first letter is upper case but field names correspond to member fields of a class so the first letter is lower case. For example: SampleEntity and fieldOne correspond to SAMPLE_ENTITY and FIELD_ONE.

Multiple primary key columns can be specified using multiple <prim-key> tags specifying the names of the primary key fields.

Field types are specified using a type string, which is defined in a fieldtypemodel XML file specified by the fieldtypemodel.dtd XML Data Type Definition. Each type maps to a Java type and an SQL type. Because of this separation different fieldtypemodel XML files can be specified for different databases allowing an entitymodel XML file to work with the various databases. In addition, validators can be specified for a field type or for any field. This denotes that the named validator should be called when data is input. These validators are defined in the class **org.ofbiz.base.util.UtilValidate** and follow the definition pattern: [`boolean isValid(String in);`].

Entity Relationships (relations)

Multiple relationships can exist for each entity. Each relation is either of type 'one', 'one-nofk', or 'many' depending on the cardinality of the relation and whether a foreign key is desired or not. If the type is 'one' or 'one-nofk' then the key-map elements must fully specify the primary key of the related entity. If the type is 'many', the key-map elements are not restricted by the primary key of the related entity.

Foreign keys and indexes on foreign keys can be created automatically by the Entity Engine. This is only done for type 'one' relations, not for 'one-nofk' or 'many' type relations.

If multiple relations to the same related entity are used for a given entity, a title must be specified to make the relation name unique. By this convention the relation name is defined as [title][rel-table-name]. For the two SampleEntity relations their names are SelfSampleEntity and AllOneSampleEntity. For OtherSampleEntity there is no title, so the relation name is simply OtherSampleEntity, or the name of the related entity.

Key Maps are used to define how the related entity will be looked up. Each key map specified a field name (field-name) and a related field name (rel-field-name). If the two column names are the same, the rel-field-name does not have to be specified.

Entity Element Reference

entity

Attribute Name	Required?	Description
entity-name	Y	The name of the entity as it is referred to when using the Entity Engine Java API and various other places in the Entity Engine.
table-name	N	The name of the database table that corresponds to this entity. This attribute is optional and if not specified the table name will be derived from the entity name.
package-name	Y	The name of the package that this entity is contained in. With hundreds of entities in a large data model this is used to organize and structure the entities definitions.
dependent-on	N	This can be used to specify a parent entity or an entity that this entity is dependent on. This is currently not used for anything automated in the Entity Engine, but can be used to specify an heirarchical entity structure.
enable-lock	N	Specifies whether or not optimistic locking should be used for this entity. The **lastUpdatedStamp** field must exist on the entity and will be used to keep track of the last time the entity instance was updated. If the current instance to be updated does not have a matching lastUpdatedStamp an EntityLockedException will be thrown. Must be true or false. Defaults to false.
never-cache	N	If this is set to true caching of this entity will not be allowed. Automatic cache clearing will not be done to improve efficiency and any attempt to use the cache methods on the entity will result in an exception so that it is easier to find and eliminate where this is being done. Must be true or false. Defaults to false.
title	N	A title for the entity. If not specified defaults to the global setting for the file the entity is in.
copyright	N	The copyright of the entity. If not specified defaults to the global setting for the file the entity is in.
author	N	The author of the entity. If not specified defaults to the global setting for the file the entity is in.
version	N	The version of the entity. If not specified defaults to the global setting for the file the entity is in.

Sub-Element Name	How Many	Description
description	0 or 1	A description of the entity. If not specified defaults to the global setting for the file the entity is in. This element has no attributes and should contain only a simple string of characters.
field	1 to many	Used to declare fields that are part of the entity.
prim-key	0 to many	Used to declare which fields are primary keys
relation	0 to many	Used to declare relationships between entities.

field

Attribute Name	Required?	Description
name	Y	The name of the field that is used to refer to it in Java code and other places.
col-name	N	The name of the corresponding database column. This is not required and if not specified this will be derived from the field name.
type	Y	The type of the field. This is looked up in the field types file for the current datasource at run-time to determine the Java and SQL types for the field and database column.

Sub-Element Name	How Many	Description
validate	0 to many	Each validate element has a single attribute called name which specifies the name of the validation method to call. These methods are not called in all Entity Engine operations and are only used for generic user interfaces like the Entity Data Maintenance pages in WebTools.

prim-key

Attribute Name	Required?	Description
field	Y	The name of the field that will be part of the primary key.

relation

Attribute Name	Required?	Description
type	Y	Specifies the type of the relationship including the cardinality of the relationship (in one direction) and if a foreign key should be created for cardinality one relationships. Must be "one", "one-nofk", or "many".
title	N	Because you may want to have more than one relationship to a single entity this attribute allows you to specify a title that will be prepended to the rel-entity-name to make up the name of the relationship. If not specified the rel-entity-name alone will be used as the relationship name.
rel-entity-name	Y	The name of the related entity. The relationship goes from this entity to the related entity.
fk-name	N	The foreign key name can be created automatically from the relationship name, but this is not recommended for two reasons: many databases have a very small maximum size (like 18 characters) for foreign key and index names, and many databases require that the FK name be unique for the entire database and not just for the table the FK is coming from.

Sub-Element Name	How Many	Description
key-map	1 to many	The key-map is used to specify a field in this entity that corresponds to a field in the related entity. This element has two attributes: **field-name** and **rel-field-name**. These are used to specify the name of the field on this entity and the corresponding name of the field on the related entity.

View Entity Modeling

In addition to entities that map directly to a single relational table you can create "virtual" or "view" entities that map to a set of other entities. The idea of a **`view-entity`** is the same as the idea of a view in Oracle, Access, or other popular database management systems. View Entities allow you to combine, or join, entities to make a new entity. The new entity's fields will be aliases of the fields on the original entities. The member entities of the view will be linked together by creating a set of view-links that contain key-maps, just like relationships do as described above.

The meta data for a **view-entity** is nearly the same as for a normal entity. It has a name, package, description, copyright, author, version, and so forth. It does not have a table name, since it does not map to a single relational table in the data source. A view entity must also be assigned to a group in the entity group XML file, just like any normal entity. All member-entities in a view-entity must be in the same database, but not necessarily in the same entity group.

```
    <view-entity title="Sample View Entity"
            copyright="Copyright (c) 2001 John Doe Enterprises"
            author="John Doe" version="1.0"
            package-name="org.ofbiz.commonapp.sample"
            entity-name="SampleViewEntity">
      <member-entity entity-alias="SE" entity-name="SampleEntity" />
      <member-entity entity-alias="OSE" entity-name="OtherSampleEntity" />
      <alias entity-alias="SE" name="primaryKeyFieldOne" />
      <alias entity-alias="SE" name="primaryKeyFieldTwo" />
      <alias entity-alias="SE" name="fieldOne" />
      <alias entity-alias="SE" name="fieldTwo" />
      <alias entity-alias="OSE" name="primaryKeyOne" />
      <alias entity-alias="OSE" name="otherFieldOne" field="fieldOne" />
      <view-link entity-alias="SE" rel-entity-alias="OSE">
        <key-map field-name="foreignKeyOne" rel-field-name="primaryKeyOne" />
      </view-link>
      <relation type="one" rel-entity-name="OtherSampleEntity">
        <key-map field-name="primaryKeyOne" />
      </relation>
      <relation type="many" title="AllOne" rel-entity-name="SampleEntity">
        <key-map field-name="primaryKeyFieldOne" />
      </relation>
    </view-entity>
```

Member Entities

Normal entities that will become part of the view entity are referred to as member entities. Each member entity is given an alias and is referred to by that alias for the rest of the definition. This allows a single entity to be linked in multiple times.

Field Aliases

Rather than specifying fields with a view entity you specify aliases. Each **alias** is effectively a field in usage and is defined by being mapped to a field on an aliases member entity. If the **field** name is the same as the alias **name**, there is no need to specify it. In other words if no field name is specified the alias name will be used as the field name to map to on the member entity with the specified **entity-alias**.

View Links

View links are used to specify the links between the member-entities of the view. They link one entity-alias to another and use key-maps just like relations. Here the field-name specifies the name of the field on the aliased entity and the rel-field-name the field on the related aliased entity. As with many other things, the rel-field-name is optional if it is the same as the field-name.

To represent an outer join you can specify in a **view-link** element that the related entity is optional using the **rel-optional** attribute, which can be either "true" or "false", and of course defaults to "false". The Entity Engine will generate ANSI, Oracle Theta, or MS SQL Server Theta style join code depending on the setting in the entityengine.xml file. See the Entity Engine Configuration Guide for more information.

Primary Keys

View Entities have primary keys just like normal entities. The Entity Engine automatically determines which aliases are primary keys by looking at the field that they alias.

The primary key for a view entity should include all primary key fields of each member entity of the view. This means that all of the primary key fields must be aliased, but fields that are used to link the entities need only be aliased once. For example, if an OrderHeader entity has an orderId primary key and an OrderLine entity has an orderId primary key and an orderLineSeqId primary key, and the orderId was the mapped key that links the two entities, then the view entity would have two primary keys: orderId and orderLineSeqId. In this case orderId would only be aliased once since by definition the orderIds from each entity will always be the same since that is how the entities are linked (or joined).

Relationships

Relationships are specified the same way with view entities as they are with normal entities. That key-map attributes are still called field-name and rel-field-name but in the case of view entities the field name is actually the alias name that will be looked up.

Grouping and Summary Data

Another useful feature available in a view-entity is the Grouping and Summary Data feature. This is accomplished with two attributes on the **alias** element: **group-by** and **function**. These work the same way conceptually as grouping and functions in SQL.

The **group-by** attribute can be set to either "true" or "false" and naturally defaults to "false". When set to true the results of a query will be grouped by the given aliased field. This means that results where the aliased fields with group-by set to true that have equal values will be grouped together.

The **`function`** attribute is used to summarize data in a result group as described in the group-by description above. The values in the grouped results will be summarized as specified by the function. The functions available include: **min | max | sum | avg | count | count-distinct | upper | lower**. These parallel commonly available functions in SQL.

While it is not required, to keep things simple I recommend querying only aliased fields that are either group-by aliases or function aliases. Also, remember that not all specified aliased fields may be included in the list of fields to retreive in the query because this will over-constrain the query.

Grouping and summarizing data can be very useful for certain types of custom coded reports or for data migration and cleansing.

> ### *Tips*
>
> *Note that view-entities can get pretty complex in a hurry. Those familiar with SQL can look at the generated SQL code to make sure it is doing what they intended. For everyone there are a few tips that might be helpful.*
>
> *First pay attention to ordering of view-links and in some cases field aliases. In view-links, except for the first one, make sure the member-entity referred to in the entity-alias attribute has been referred to in a previous view-link (this is how the link tree is kept clean).*

View Entity Element Reference

view-entity

Attribute Name	Required?	Description
entity-name	Y	The name of the entity as it is referred to when using the Entity Engine Java API and various other places in the Entity Engine.
package-name	Y	The name of the package that this entity is contained in. With hundreds of entities in a large data model this is used to organize and structure the entities definitions.
dependent-on	N	This can be used to specify a parent entity or an entity that this entity is dependent on. This is currently not used for anything automated in the Entity Engine, but can be used to specify an heirarchical entity structure.
never-cache	N	If this is set to true caching of this entity will not be allowed. Automatic cache clearing will not be done to improve efficiency and any attempt to use the cache methods on the entity will results in an exception so that it is easier to find and eliminate where this is being done. Must be true or false. Defaults to false.
title	N	A title for the entity. If not specified defaults to the global setting for the file the entity is in.
copyright	N	The copyright of the entity. If not specified defaults to the global setting for the file the entity is in.
author	N	The author of the entity. If not specified defaults to the global setting for the file the entity is in.
version	N	The version of the entity. If not specified defaults to the global setting for the file the entity is in.

Sub-Element Name	How Many	Description
description	0 or 1	A description of the entity. If not specified defaults to the global setting for the file the entity is in. This element has no attributes and should contain only a simple string of characters.
member-entity	1 to many	Used to declare which entities will be part of this view-entity and what alias will be used to refer to them.
alias	1 to many	Used to declare the aliased fields from the member-entities that will be part of the view entity.
view-link	1 to many	Used to declare how the member-entities in the view will be linked together.
relation	0 to many	Used to declare relationships between entities.

member-entity

Attribute Name	Required?	Description
entity-alias	Y	The alias to use for this member-entity. Must be unique among all member-entities. This will be used to refer to the member-entity in other places in the view-entity definition.
entity-name	Y	The name of the entity definition that this member entity corresponds to. The same entity can be used multiple times in the same view-entity with different entity-aliases.

alias

Attribute Name	Required?	Description
entity-alias	Y	The entity-alias of the member-entity that this aliased field corresponds to.
name	Y	The name of the field alias. This is used when interacting with the view-entity the same way a field name is used when interacting with an entity.
field	N	The name of the field from the member-entity with the given entity-alias that this field alias corresponds to. If not specified defaults to the alias name as specified in the name attribute.
prim-key	N	Used to specify if this alias should be part of the view-entities primary key. If specified must be either true or false. If not specified the Entity Engine will look at the definition of the field it came from to see if it is part of the original entity's primary key.
group-by	N	Used to specify that the aliased field should be used for grouping results and should be used in conjunction with the function attribute on other aliased fields. For a more complete discussion see the main text. Must be either true or false. Defaults to false.
function	N	Used to specify a function to be used on this field to calculate summary information. Should be used in conjunction with the group-by attribute to specify how the summary results should be grouped. See the main text for a more complete discussion of how to use this.

view-link

Attribute Name	Required?	Description
entity-alias	Y	The alias of the entity the link is coming from.
rel-entity-alias	Y	The alias of the entity the link is going to.
rel-optional	N	Used to specify whether or not the related entity is optional. If this is true it effects an outer join to the related entity. Must be either true or false. Defaults to false.

Sub-Element Name	How Many	Description
key-map	1 to many	The key-map is used to specify a field in this entity that corresponds to a field in the related entity. This element has two attributes: **field-name** and **rel-field-name**. These are used to specify the name of the field on this entity and the name of the field on the related entity.

relation

Attribute Name	Required?	Description
type	Y	Specifies the type of the relationship including the cardinality of the relationship (in one direction) and if a foreign key should be created for cardinality one relationships. Must be "one", "one-nofk", or "many".
title	N	Because you may want to have more than one relationship to a single entity this attribute allows you to specify a title that will be prepended to the rel-entity-name to make up the name of the relationship. If not specified the rel-entity-name alone will be used as the relationship name.
rel-entity-name	Y	The name of the related entity. The relationship goes from this entity to the related entity.
fk-name	N	The foreign key name can be created automatically from the relationship name, but this is not recommended for two reasons: many databases have a very small maximum size (like 18 characters) for foreign key and index names, and many databases require that the FK name be unique for the entire database and not just for the table the FK is coming from.

Sub-Element Name	How Many	Description
key-map	1 to many	The key-map is used to specify a field in this entity that corresponds to a field in the related entity. This element has two attributes: **field-name** and **rel-field-name**. These are used to specify the name of the field on this entity and the name of the field on the related entity.

The Entity Engine API

The Entity Engine classes in the package **org.ofbiz.entity** define the API used to interact with Entity Engine entity data. From a users point of view only three classes really need to be understood. They are **GenericDelegator**, **GenericValue** and **GenericPK**. The **GenericDelegator** class, usually used with the instance name '**delegator**', is used to do create, find, store and other operations on a **GenericValue** object. Once a **GenericValue** object is created it will contain a reference to the delegator that created it and through this reference it knows how to store, remove and do other operations without requiring a program to invoke methods on the delegator itself.

I've been trying to think of how best to present information about this API but short of writing a number of documents about the specific usage of each piece there is not much that is useful that I could write here. I recommend reading through the JavaDocs for the Entity Engine and other framework components of OFBiz, and browsing through the various applications in OFBiz for examples. These heavily use the Entity Engine API.

A few quick notes to help you get started might be in order.

Factory Methods

Rather than trying to construct a GenericValue or a GenericPK yourself, you should use the **makeValue** and **makePK** methods on the **GenericDelegator**. These create an object without persisting it and allow you to add to it and create or store it later using their own **create** or **store** method, or calling the **create** or **store** method on the **delegator** object.

Creating, Storing and Removing

To create (or insert) values into the database, use the **create** method on the **GenericValue** or **GenericDelegator** objects. To store (or update) existing values, use the **store** method on the **GenericValue** or **GenericDelegator** objects.

For storing multiple entities the **GenericDelegator** class has a method called **storeAll**. This method takes many **GenericValue** instances and stores them in the same transaction. Actually, to say that it stores them is incorrect. It checks to see if they exist and if so does an update, if not it does an insert. This may be optimized in the future for speed by allowing you to specify whether it should be inserted or updated based on prior knowledge of the existence of that entity. Note that this is a DIFFERENT behavior than the store method, which just does an update.

Removal of entities is done through the **remove** method on either the **delegator**, or the **GenericValue**.

Finding

Value instances can be retrieved from the database with the **findByPrimaryKey** methods, or a collection can be retrieved using the **findAll** or **findByAnd** methods.

There are two main types of findByAnd methods. Each type has a number of variations that may include the use of a cache, and may accept an orderBy field list. The two main types accept different field lists. One accepts a Map of fields and finds entities by anding together expressions where each named field must equal the corresponding value in the map. The other type of findByAnd accepts a list of **EntityExpr** objects that are used to specify small expressions that will be anded together. Each EntityExpr specifies a field name, an operation, and a value for the field.

The EntityCondition Object

Originally the EntityExpr object was meant to be nestable to allow for more flexible queries, but was never completed. Also, even if you could nest it the types of queries you could run would be limited because you couldn't have two ANDs (for instance) inside a set of parentheses. To address these issues, and complete the EntityExpr implementation, the EntityCondition abstract object has been introduced along with the EntityConditionList and EntityFieldMap objects which both extend EntityCondition. The EntityExpr object has also been changed to extend EntityCondition.

The EntityConditionList and EntityFieldMap objects are pretty simple. They are created with a List or Map, respectively, and an EntityOperator to specify operator used to join the members of these containers, generally AND or OR.

The EntityExpr class now has two primary constructors: one for comparing a field to a value (the String, EntityOperator, Object constructor), and one for comparing two EntityCondition objects (the EntityCondition, EntityOperator, EntityCondition constructor).

A findByCondition method is now available that accepts an EntityCondition argument (as well as some other useful arguments) and this EntityCondition can be an EntityExpr, EntityConditionList or EntityFieldMap.

The code inside the Entity Engine has changed somewhat because these EntityCondition objects now create their own WHERE clauses. This is a nice architectural point because other custom EntityConditions could also be created and used as desired by a savvy developer.

On that note a "short cut" implementation of the EntityCondition abstract class has been created to incorporate SQL WHERE clause snippets or full clauses into the EntityCondition querying framework. The class is named EntityWhereString and is constructed with a simple String argument that represents the SQL that will be inserted into the final generated SQL. This is not recommended when other options are available, but it necessary for certain functionality that the Entity Engine does not otherwise support.

The EntityListIterator Object

The EntityListIterator class implements the ListIterator interface for convenience, but also has other methods that are necessary for its operation, like a close() method for when you are finished.

This object allows you to iterate through query results efficiently in both directions by keeping a reference to the ResultSet that comes back from the query. This makes it possible to use the cursor feature in the database and especially for large queries uses memory much more efficiently. This object constructs GenericValue objects on the fly rather than creating a bunch all at once, so if you need to export the results of a huge query to a file or something, it can be done without a massive amount of memory.

The EntityListIterator also has helpful methods to get a subset of the result set with a start index value and a number of results desired and to get all of the results at once rather than having to create a custom loop to do so.

The Entity Engine Cache

Because the performance cost of retrieving data from a database can be very high and can have a serious impact on the overall performance of an application or component of an application, the ability to cache data from the database is often very important. Many of the find methods on the GenericDelegator have corresponding cache methods with the "Cache" suffix on the function name. This makes it very easy to cache individual values found by primary key as well as value lists found by and, or by all.

The Entity Engine cache uses the OFBiz **`UtilCache`** class to implement the actual cache. UtilCache has many features such as limited cache size, expiring cache entries after a configurable amount of time, and soft references so that the garbage collector can reclaim entries from large caches when more memory is needed.

There are two ways to configure UtilCache based caches. The first is to modify the cache.properties file (as documented in the Core Configuration Guide) for permanent changes. The second is to make temporary changes through the Cache Management pages in the WebTools webapp. Those pages can also be used to view statistics, clear cache values, and perform other cache related maintenance.

A big issue with this type of cache, where it is not tied directly to the database, is clearing "dirty" cache entries. When create, store or remove operations are done through the Entity Engine, it will normally be able to automatically clear any cache entries that might contain the updated value. Distributed cache clearing is also implemented and can be configured in a number of ways using the flexibility of the Service Engine.

JTA Support

The Entity Engine JTA Support is simple to use, but has a few complications in configuration. The support runs through an API and a Factory class so that no direct contact with the particular JTA implementation is necessary. The TransactionFactory class can be used to get the two main objects needed for JTA use: UserTransaction and TransactionManager. The current implementation supports the Tyrex JTA/JTS implementation. To use a different implementation simply change the TransactionFactory class; everything else uses that. That's the tricky configuration part, if you aren't using Tyrex. For Tyrex make sure that a domain configuration XML file called tyrexdomain.xml is on the running classpath.

To demarcate transactions you can use the TransactionUtil class. This class wraps the UserTransaction class and only throws GenericEntityExceptions and runtime exceptions. The basic methods needed are begin(), commit(), and rollback(), but the rest are included and can be very useful. A transaction is attached to the current thread so it is not necessary to pass it around all over the place. After beginning a transaction make sure it is always either committed or rolled back. This is normally done by committing at the end of a try block and rolling back in each catch block. You can also use the standard UserTransaction object by getting one from the TransactionFactory.

Core Web Tools

The WebTools web application contains a number of useful tools for working with the entity engine. These include a cross linked reference to the entity definitions, a tool for editing entity and relation definitions, and a JSP which acts as an XML template and also saves the XML entity definitions to their corresponding files.

There is also a JSP that acts as a front end to the routines which checks the current state of the database table definitions and reports any differences. Where possible tables or columns that are missing can be added to the database. This is the same routine that optionally runs when the server loads and optionally creates missing tables and columns.

The entity code generator has been removed from the project, or deprecated if you will, because of the next generation entity tool which is the entity engine described herein. There are still some occasions where the use of templates to create entity specific code or other text is useful, and in fact, necessary. One example of this is the JSP which creates the XML for the entity definitions from those definitions, and is used for writing out the XML after entity definitions have changed. Other uses for templates include manually generating database-specific table creation SQL and quick start JSPs and event handlers that allow for finding, viewing and editing entity specific data. These can be used as starting points for task specific applications.

Where entity data editing is not task specific, but instead is entity specific, the Entity Data Maintenance pages in WebTools can be used. They are dynamic pages that rely on the in-memory entity definitions to create forms for the entering of data, and events for handing the entered data (including validators specified in the entity definition), and finding specific entities by any of the fields on the entity, or finding any relation entity instances for a given entity instance. For instance, when viewing the OrderHeader entity all of the relations to that entity can be viewed as well, including links to edit and view them. These related entities would include OrderType (one relation), OrderLine (many relation), and many others.

Data in the database can be imported from and exported to Entity Engine XML files in the import and export pages. Importing data causes corresponding entity instances to either be created or updated, depending on whether or not they already exist. The Export page allows you to specify which entities you want to export the data for by using a big list of check boxes, one for each entity. For more granular control over exported data, the Entity Data Maintenance pages mentioned above would be the place to look (not yet finished though...).