Ebean v2.6.0

User Guide

May 2010

1 Contents

2 Introduction

2.1 Ebean

Ebean is an open source Object Relational Mapping tool.

It's goal is to provide a simpler alternative to JPA (Java Persistence API) implementations such as Hibernate and Eclipselink.

It does this by providing a “sessionless” API and a simpler query language.

That means:

- No Session Object (or UnitOfWork or EntityManager)
- No Attached or Detached Beans
- No merge(), persist(), flush(), or clear(). Instead Ebean has save() and delete()

I have found (so far) that this is a bit hard for many people familiar with JPA / Hibernate / Eclipselink etc to get their head around. The short answer is that Ebean, just like JPA has a Persistence Context but has some architectural differences to allow it to have a different approach to the entity bean lifecycle and removing the need to manage EntityManagers.

2.2 Why Ebean? ... why different from JPA?

Ebean uses the JPA Mapping annotations and will follow those very closely.

However, Ebean has been architected and built from a different perspective than JPA. The Architectural and Query language differences are reasonably fundamental to both Ebean and JPA so its hard to see these differences going away anytime soon.

It becomes a question of whether this different approach taken by Ebean has technical merit and has made Ebean an easier ORM to understand and use.

2.2.1 Architecture: Session / UnitOfWork / EntityManager

JPA is architected to use an “EntityManager” which closely matches a Hibernate “Session” and a Toplink “UnitOfWork”. This brings with it the concepts that an entity bean can be attached or detached from the EntityManager (with associated merge, flush clear operations etc). If the EntityManager is used across multiple Transactions the EntityManager needs to be managed typically by a EJB Session Bean, Seam, Spring or similar container/framework.

Ebean is architected to not require an EntityManager (or Session or UnitOfWork object) with the goal of making it easier to use and to remove the requirement to manage

EntityManager objects (Aka a different approach to Lifecycle management).

Although Ebean doesn't have a EntityManager it ***DOES*** have a “Persistence Context” (like JPA) and by default in Ebean the persistence context is transaction scoped and automatically managed.

In my opinion this makes Ebean easier to understand and use.

2.2.2 Query Language

JPA has defined a powerful query language. The issue with the query language is that I believe it will be difficult to evolve the language to support “Partial Objects” and generics (returning typed Lists/Sets etc) and this is due to specific issues with the JPQL select clause as it is currently defined.

Ebean has a simplier query language that I believe is more orientated to object graph construction. It has enabled Ebean to easily support “Partial Object” queries and return typed Lists Sets and Maps using generics.

It is my understanding the JPA expert group had discussed adding 'fetch groups' (which would be similar to Ebean's partial objects) to the JPA specification quite some time ago and decided not to. That is, they don't percieve the performance benefits to be worth the complexity. The Ebean expert group has a different opinion.

For Ebean “Partial Objects” are seen as an important feature from a performance stand point (fetch less from the database in a simple flexible manor) and from a design perspective (no need to change the design for “wide” objects and follow fixed designs based on “secondary” table properties and fixed annotations for eager or lazy loading) so there looks like a difference in opinion in this respect between JPA and Ebean.

In time it will be interesting to see how and if JPQL evolves and especially if there are moves for it to support “partial object” queries (possibly by introducing “fetch groups” into the JPA spec).

2.2.3 Why “Partial Objects” can greatly effect performance

So, given JPA may not have “Partial Object” support in the query language for some time I'll just outline the reasons it is considered important for Ebean. They are pretty obvious except for what I describe as ***“Index evaluation (database not having to read data blocks)”.***

Per Use Case

Having “partial object” support in the query language means you can write a query that is optimal for each use case. If you have fixed annotations to specify whether a property is lazy or eager loaded then you can not have optimal queries for a specific use cases (you will often end up with suboptimal queries).

Clobs / Blobs

Clobs and Blobs are especially expensive and the ability to include/exclude clob/blob properties per use case/query can be significant in terms of performance.

Wide tables/entities – network cost

The wider you entities get the more cost is involved in fetching Gnd □eading properties that a use case may never use. Some people have gone to various lengths to handle this issue such as having multiple representations of the same bean (to vertically partition a wide entity). With “partial objects” you don't have to change your design if you hit a performance 'pain point'.

This cost is not only in terms of the database reading columns that are not required but in the extra network traffic involved in retrieving this data that is never used by that part of the application.

Index evaluation (database not having to read data blocks)

Something special happens when your SQL query only includes columns that are included in indexes. When this occurs the database can completely evaluate the query from the data held in the indexes and ***NOT*** have to read data blocks (obviously a generalisation on RDBMS internals but it's a good generalisation).

```
// database doesn't have to read the data blocks
// if “id” and “name” are in index(s)
select c.id, c.name
from customer c
where c.name like 'Rob%'
```

If both the “id” and “name” columns are in indexes then the database could evaluate this query without having to read the data blocks. Compared to “Data blocks” “Index blocks” are more likely to be held in memory and smaller.

BUT once you include a column that is not in an index then this doesn't hold – so surely this is not that useful.

True, except something similar also occurs for joins where you only use columns on the joined table that are in an index(s).

For example, joining order_details to the products table and only using the product.name.

```
// database doesn't have to read 'product' table data blocks
select o.*, d.*, p.name
from order o
join order_details d on d.order_id = o.id
join product p on p.id = d.product_id
```

```
where o.id = 27
```

So the part of the query evaluation relating to joining products can be performed by only reading the index – the data blocks for the product table do ***NOT*** have to be read.

```
// the equivalent Ebean query
find order (*)
fetch details (*)
fetch details.product (name)
where id = 27
```

Now instead you may find more complex queries with many joins and you may be able to get this optimisation on a several of those joins. For example, add a join to customer to the example above just fetching the customer's name – now you get the optimisation on the join to product and on the join to customer.

You may find this optimisation occuring a lot more than you think – for some queries that need to be faster you may even look to add an index (probably a compound index) to target this optimisation specifically (to support a autocompletion UI control for example).

Overall, what this means is that “partial object” queries can result in either all or part of the query being evaluated by the database via data held in indexes without the database needing to read data blocks. For large databases this can be a very significant performance optimization. If you need this Ebean has made it very easy without requiring a change to your design (either ORM or Database design).

To be fair most of the major ORM's provide similar functionality via “Fetch Groups”. It is not JPA standard yet (and IMO not as elegant as “partial objects”) but is probably worth investigating if you are using one of those products.

Currently as I see it, the differences between Ebean's “partial objects” and “Fetch Groups” is that “Fetch Groups” seem to have limitations such as being read only (can't modify and save), and don't allow further lazy loading. Ebean's “partial objects” are really easy to use and can be treated just like other entity beans with lazy loading and support for being saved.

2.2.4 All the missing bits

JPA is a developing specification but in my opinion it leaves a lot of very useful features out. This includes partial objects, batching control, support for large queries (row level persistence context), background fetching, caching control, transaction isolation.

It also has “interesting” support for using raw SQL – in my opinion Ebean makes using raw SQL much easier. In my experience there are some tasks that are more naturally done in a Relational way.

Obviously JPA will evolve and may look to support some of these features in the future. In the meantime, if you are using JPA do not be surprised if you need to use vendor specific features.

2.3 Ibatis

ORM is neat but at the same time we should remember that there are times when it is easier/better/faster to have direct control over the SQL. Ibatis is such a tool and very popular with this in mind.

Part of Ebean's goal is to make using your own SQL easy (easier than Ibatis is the goal for Ebean). The @SqlSelect feature goes a long way to doing this along with SqlUpdate and CallableSql objects.

3 Ebean and EbeanServer

Ebean and EbeanServer are the objects that provide the main API to find and save entities etc. It is worth quickly explaining what they are and how they relate.

EbeanServer

- This provides the main API
- There is one EbeanServer per DataSource
- One of the EbeanServer's can be nominated as the “Default” EbeanServer
- A EbeanServer can be 'registered' with the Ebean singleton when it is created. This means it can later be accessed via:

```
EbeanServer s = Ebean.getServer(serverName);
```

Ebean (singleton)

- Is a Singleton
- Holds a map of EbeanServer's
- Provides methods that proxy through to the “Default” EbeanServer. This is convienient for applications that use a single DataSource.
- i.e. Ebean.find(Person.class, 7); actually calls the find method on the “Default” EbeanServer.... Ebean.getServer(null).find(Person.class, 7);

Most of the examples in this document will use Ebean (the singleton) rather than EbeanServer.

```
// Using Ebean (singleton)...
Customer customer = Ebean.find(Customer.class, 4);

// Is equivilent to...
EbeanServer defaultServer = Ebean.getServer(null);
Customer customer = defaultServer.find(Customer.class, 4);
```

4 Features Overview

4.1.1 Mapping

Ebean uses the same mapping as per the JPA specification. So you annotate your beans with @Entity, @Table, @Column, @OneToMany etc as per the JPA specification.

The Mapping is covered in detail in section

4.1.2 Query

Some query examples

```
// find a customer by their id
Customer customer = Ebean.find(Customer.class, 4);

// find a list...
List<Customer> customers =
    Ebean.find(Customer.class)
          .where().like("name", "Rob%")
          .orderBy("name desc")
          .findList();
```

You can optionally use select() and fetch() to specify only the properties you want to fetch. This returns "partially" populated beans. This is an important feature to help with the performance of your queries.

```
// with select() and fetch() you can specify
// only the properties you want to fetch.
// ... returning "Partially" populated objects

List<Order> orders =
    Ebean.find(Order.class)
          .select("status, shipDate, orderDate")
          .fetch("customer", "name")
          .where()
                .eq("status", Status.ACTIVE)
                .like("customer.name", "Rob%")
          .orderBy("customer.name desc");
          .findList();
```

A query using the query language

```
String q = "find order fetch customer fetch details where id=:id";

Order order = Ebean.createQuery(Order.class,q)
    .setParameter("id", 1)
    .findUnique();
```

The previous examples used a “fluid” API style where the methods are all chained together. You can equally choose a more traditional style, where you create a query *via Ebean.createQuery() or Ebean.find(Class beanType), and then* set various query properties and finally using *query.findUnique() query.findList() query.findSet()* or *query.findMap()* to return the result.

```
String q = "find order fetch customer fetch details where id=:id";

// using non-fluid style...
Query<Order> query = Ebean.createQuery(Order.class,q);
query.setParameter("id", 1);

Order order = query.findUnique();

// using non-fluid style...
Query<Order> query = Ebean.createQuery(Order.class);
query.select("status, shipDate, orderDate");
query.fetch("customer", "name");
query.where().eq("status", Status.ACTIVE)
                .like("customer.name", "Rob%");
query.orderBy("customer.name desc");

List<Order> orders = query.findList();
```

4.1.3 RawSql

You can explicitly specify the sql to use to build object graphs. This is useful for “Reporting” type requirements where you want to use aggregate functions such as sum() count() max() etc.

It is also useful if you need to use Database specific SQL for whatever reason.

```
/**
 * An example of an Aggregate object.
 *
 * Note the @Sql indicates to Ebean that this bean
 * is not based on a table but instead uses RawSql.
 *
 */
@Entity
@Sql
public class OrderAggregate {

    @OneToOne
    Order order;

    Double totalAmount;

    Double totalItems;

  //getters setters etc
```

RawSql Example 1:

```
String sql
    = " select order_id, o.status, c.id, c.name,
              sum(d.order_qty*d.unit_price) as totalAmount"
    + " from o_order o"
    + " join o_customer c on c.id = o.kcustomer_id "
    + " join o_order_detail d on d.order_id = o.id "
    + " group by order_id, o.status ";

RawSql rawSql =
    RawSqlBuilder
       .parse(sql)
      // map result columns to bean properties
       .columnMapping("order_id",  "order.id")
       .columnMapping("o.status",  "order.status")
       .columnMapping("c.id",      "order.customer.id")
       .columnMapping("c.name",    "order.customer.name")
       .create();


Query<OrderAggregate> query = Ebean.find(OrderAggregate.class);
    query.setRawSql(rawSql)
```

```
    // with "parsed" SQL we can add expressions to the
    // where and having clauses etc
    .where().gt("order.id", 0)
    .having().gt("totalAmount", 20);
```

Note that you can put RawSql into a ebean-orm.xml with a name and then use a RawSql query just as you would a named query.

Example 2:

```
// This example has uses fetch() to fetch related order
// and customer information after the initial RawSql
// query is executed

String sql
    = " select order_id, 'ignoreMe',
                sum(d.order_qty*d.unit_price) as totalAmount "
    + " from o_order_detail d"
    + " group by order_id ";

RawSql rawSql =
  RawSqlBuilder
    .parse(sql)
    .columnMapping("order_id",  "order.id")
    .columnMappingIgnore("'ignoreMe'")
    // don't need this when using column alias
    //.columnMapping("sum(d.order_qty*d.unit_price)","totalAmount")
    .create();


Query<OrderAggregate> query = Ebean.find(OrderAggregate.class);
query.setRawSql(rawSql)
    // after the RawSql query executes Ebean can execute
    // FetchConfig().query() joins ...
    .fetch("order", "status,orderDate",new FetchConfig().query())
    .fetch("order.customer", "name")
    .where().gt("order.id", 0)
    .having().gt("totalAmount", 20)
    .order().desc("totalAmount")
    .setMaxRows(10);
```

```
List<OrderAggregate> list = query.findList();
```

You can use RawSql with normal entity beans as well fetching only the properties to need (creating partially populated entity beans).

Note that all entity beans built with RawSql invoke lazy loading etc and act just the same as if they where populated via Ebean generated SQL.

4.1.4 Autofetch – Automatic Query Tuning

Ebean version 0.9.7 introduced support for Autofetch. This is a mechanism that can automatically tune your queries for optimal performance.

Autofetch automatically modifies your queries – essentially controlling the select() and fetch() clauses to fetch all the data your application uses but no more. This has the effect of reducing the amount of lazy loading and only fetches properties that are actually used.

Autofetch is explained in more section 5.1 Autofetch.

Note that Autofetch can now be used with a query that you have explicitly specified some fetch() paths. Autofetch can add additional fetch() paths and tune which properties to fetch per path.

4.1.5 Save & Delete

Saving and deleting is just a matter of calling the Ebean.save() or Ebean.delete() methods. Transaction demarcation is covered fully in 6. Transactions . Note if no transaction currently exists one will be created and commited for you (or rolled back if there was an error).

```
Order order = Ebean.find(Order.class, 12);

order.setStatus(OrderStatus.SHIPPED);
order.setShipDate(...);

// this will save the order
Ebean.save(order);
```

Cascading Save & Delete

The mapping annotations @ManyToOne, @OneToMany, @OneToOne and @ManyToMany provide a cascade attribute which is used to control whether saves and deletes are cascaded.

The default is to not cascade a save or delete (as per JPA spec).

The example below shows the Order entity bean with its mapping annotations. If you save an Order the details will be saved as well but the associated customer will not be saved as there is no cascade atttribute and the default is to not cascade.

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

    ...
    // no cascading
    @ManyToOne
    Customer customer;

    // save and delete cascaded
    @OneToMany(cascade=CascadeType.ALL)
    List<OrderDetail> details;
```

4.1.6 Update

Update provides a way on issuing a insert, update or delete statement.

This is useful for updating or deleting multiple rows (or a single row) with a single statement (often described as a “bulk” update).

This is also useful if you want to perform an update or delete without having to execute a query first. This is a typical approach for performing an update in a stateless web application.

The statement can be provided in as raw DML with the table names and column names or in a 'logical' form where entity name is used in place of the table name and property names are used in place of column names.

4.1.7 Transactions

If no Transactions are demarcated Ebean will automatically create and commit a transaction as required.

```
Order order = ...
...
```

```
// 'Implicit' transaction created and committed
Ebean.save(order);
```

Transactions can be demarcated with via @Transactional annotation (requires Enhancement to be used) and serveral programmatic approaches – TxRunnable/ TxCallable and beginTransaction() etc.

Transactions are covered fully in section 6. Transactions

Some examples are:

@Transactional example

```
...
public class MyService {

    @Transactional
    public void runFirst() throws IOException {

          // do multiple things in a single transaction
          User u1 = Ebean.find(User.class, 1);
          …
          Customer cust = Ebean.find(Customer.class, 27);
          …
          Ebean.save(cust);
    }
```

Programatic TxRunnable Example

```
public void myMethod() {
  ...
  System.out.println(" Some code in myMethod...");

  // run in Transactional scope...
  Ebean.execute(new TxRunnable() {
    public void run() {

          // code running in "REQUIRED" transactional scope
          User user = Ebean.find(User.class, 1);
          ...
```

```
            Ebean.save(user);
            ...
        }
    });

    System.out.println(" more code in myMethod...");
}
```

Programatic beginTransaction() example

```
Order order = ...
Customer customer = order.getCustomer();

// 'Explicit' transaction
Ebean.beginTransaction();
try {
    ...
    Ebean.save(customer);
    Ebean.save(order);
    Ebean.commitTransaction();

} finally {
    Ebean.endTransaction();
}
```

With a Transaction can also control JDBC batching (batch size, flushing), turn on/off transaction logging, turn on/off cascading of save & delete.

5 Relational Features

Object Relational Mapping is great but it is also possible some of your problems will be easier to tackle in a relational way. Ebean provides a set of relational features so that you choose a relational approach as you see fit.

5.1.1 SqlQuery

SqlQuery is where you specify the exact SQL SELECT statement and returns list, sets or maps of SqlRow objects. A SqlRow is a Map where the key is the column name.

This is a fairly lightweight API that you could use instead of going to raw JDBC (which is of course an option).

```
String sql = "select b.id, b.title, b.type_code, b.updtime"
          +" ,p.name as product_name "
          +"from b_bug b join b_product p on p.id = b.product_id "
          +"where b.id = :id";

SqlRow bug = Ebean.createSqlQuery(sql)
    .setParameter("id", 1)
    .findUnique();

String prodName = bug.getString("product_name");
String title = bug.getString("title");
```

Note that you can use “Named” queries and put the sql statements in orm.xml rather than having it in your code.

5.1.2 SqlUpdate

In similar fashion to SqlQuery you can specify a SQL INSERT, UPDATE or DELETE statement with named or positioned parameters.

```
String dml = "update b_bug set title=:title where id = :id";

SqlUpdate update = Ebean.createSqlUpdate(dml)
    .setParameter("title", "Updated Again")
    .setParameter("id", 1);

int rows = update.execute();
```

5.1.3 CallableSql

CallableSql provides a way to call a database stored procedure.

```
String sql = "{call sp_order_mod(?,?)}";

CallableSql cs = Ebean.createCallableSql(sql);
cs.setParameter(1, "turbo");
cs.registerOut(2, Types.INTEGER);

Ebean.execute(cs);

// read the out parameter
Integer returnValue = (Integer) cs.getObject(2);
```

You can extend CallableSql and you can also get the java.sql.Connection from a Transaction and use raw JDBC API to call a stored procedure.

5.1.4 Summary

These relational features provide an alternative “relational” approach to the ORM features without resorting to direct JDBC use.

5.2 Raw JDBC

You can't always predict when your application requirements can't be meet with the features in Ebean. It is nice you now you can easily use raw JDBC if and when you need to.

The java.sql.Connection object can be returned from a transaction, and with that you can perform any raw JDBC calls you like.

This may be useful for Savepoints, advanced Clob/Blob use or advanced stored procedure calls (if CallableSql doesn't do the business for you).

```
Transaction transaction = Ebean.beginTransaction();
try {

    Connection connection = transaction.getConnection();
    // use raw JDBC
    ...

 // assuming we updated the "o_shipping_details" table
 // inform Ebean so it can maintain it's 'L2' cache
 transaction.addModification("o_shipping_details",false,true,false);

    Ebean.commitTransaction();
} finally {
    Ebean.endTransaction();
}
```

The ***transaction.addModification()*** in the code above informs Ebean that your jdbc code updated the o_shipping_details table. Ebean uses this information to automatically manage its "L2" cache as well as maintain Lucene text indexes.

6 Queries

6.1 Background

Ebean has it's own query language. Prior to this decision JPQL (the JPA query language) was investigated to see if it would meet the desired goals of Ebean and it did not. Specifically I wanted to support “Partial Objects” via the query language and it is difficult to see how JPQL will evolve to support this (specifically difficulties around its select clause).

Apart from “Partial Object” support there was also a desire to simplify the join syntax, specifically Ebean will automatically determine the type of join (outer join etc) for you and also automatically add joins to support predicates and order by clauses.

JPQL is more powerful with the ability to mix entity beans with scalar values returning Object[]. However, this feature also could be a major stumbling block for it to evolve support for partial objects for any node in the object graph.

In summary you could say the Ebean query language is much simplier that JPQL with the benefit of proper support for “Partial Objects” for any node in the object graph (this is not possible with JPQL in it's current form).

“Partial Object” support in Ebean is important for design reasons and performance reasons. From a performance perspective your queries are more performant if they fetch less data back from the database. From a design perspective you do not need to model using secondary tables but instead use partial objects at any depth in the query.

For example, to build an object graph for an Order you may want some product information for each orderDetail.

6.2 Examples

```
// find all the orders fetching all the properties of order
find order


// find all the orders fetching all the properties of order
// ... this is the same as the first query
find order (*)


// find all the orders fetching the id, orderDate and shipDate
// ... This is described as a “partial object query”
```

```
// ... the ID property is *ALWAYS* fetched
find order (orderDate, shipDate)
```

```
// find all the orders (and orderDetails)
// ... fetching all the properties of order
// ... and all the properties of orderDetails
// ... the type of fetch(Outer etc) is determined automatically
find  order
fetch orderDetails


// find all the orders (with their orderDetails)
// ... fetching all the properties of order
// ... and all the properties of orderDetails
find  order (*)
fetch orderDetails (*)
```

```
// find all the orders (with orderDetails and products)
// ... fetching the order id, orderDate and shipDate
// ... fetching all the properties for orderDetail
// ... fetching the product id, sku and name
find  order (orderDate, shipDate)
fetch orderDetails (*)
fetch orderDetails.product (sku, name)
```

Every object in the object graph can be a partial object. This is what you can't do in JPQL yet and it's hard to see how this will be supported in JPQL due to its design – hopefully I'm wrong on this point.

These Partially populated objects are will lazy load as required and are fully updatable etc. You can treat them just like fully populated objects.

Autofetch can use partial objects to only fetch the properties that the application actually uses. In this way you can get the performance of partial objects without any work on your part (Autofetch determines the joins and properties to fetch for you).

```
// Ebean will automatically add joins to support
```

```
// where clauses and order by clauses as necessary
// ... in this case a join to customer is added
// ... and a join to the customers billing address is added
find order
where customer.name like :custname
order by customer.billingAddress.city
```

```
// you can use an order by and limit offset clause
find order
where customer.name like :custname
order by customer.name desc, id
limit 10 offset 20
```

```
// You can use +readonly hint on any part of the object graph
// ... which means those objects are not modifyable
find customer (+readonly)
fetch billingAddress (+readonly, line1, city)
```

6.3 API: createQuery(Class c) and find(Class c)

This needs a little clarification.

```
// these are the same
Query<Order> query = Ebean.createQuery(Order.class);

Query<Order> query = Ebean.find(Order.class);
```

This may be confusing but these two methods do exactly the same thing. The reason both exist is because the createQuery() style is consistent with JPA and could be argued is a better more accurate name. However, I feel that find() is more consistent with the fluid API style.

So, apologies in that there are 2 ways to do the same thing.

```
// fluid API style with find()
```

```
List<Order> list =
    Ebean.find(Order.class)
    .fetch("customer")
    .where().eq("status.code", "SHIPPED")
    .findList();
```

6.4 Named Queries

```
...
@NamedQueries(value={
  @NamedQuery(
    name="bugsSummary"
    ,query="find (name, email) fetch loggedBugs (title, status)
                where id=:id  "),
  @NamedQuery(
    name="bugStatus",
    query="fetch loggedBugs where loggedBugs.status = :bugStatus
           order by name")
})
@Entity
@Table(name="s_user")
public class User implements Serializable {
...
```

You can have named queries, where you define the query. Note that the names of the queries are per entity type (not global as they are in JPA).

Once you get a named query you set any named parameters and then execute it – in the case below we use findUnique() as we expect only one object graph returned.

```
User u = Ebean.createNamedQuery(User.class, "bugsSummary")
          .setParameter("id", 1)
          .findUnique();
```

6.4.1 Named Queries are Modifyable

Named queries are parsed early and returned as query objects to you that you can modify. This means that you can get a named query and then modify the query by adding to the where clause, setting the order by, limits etc.

This is an intentional feature and means that you can use Named Queries as a “starting point” to then modify via code and execute.

```
// you can treat namedQueries as starting points...
// ... in that you can modify them via code
// ... prior to executing the query

// you can modify a named query...
Set<User> users = Ebean.createQuery(User.class, "bugStatus")
    .setParameter("bugStatus", "NEW")
    // you can add to the where clause
    .where().ilike("name", "rob%")
    // you can set/override the order by
    .orderBy("id desc")
    // you can set/override limits (max rows, first row)
    .setMaxRows(20)
    .findSet();
```

6.5 FetchConfig - “Query Joins”

When you specify a Query with Ebean it can result in more than 1 SQL query. Sometimes you want explicit control over this (what the secondary queries are, batch size used, eager or lazily invoked)

FetchConfig gives you the ability to specify these “secondary queries” and let them executed lazily ("lazy loading join") or eagerly ("query join").

Note that Ebean will automatically convert some joins to “query joins” when it needs to (when it is building object graphs with multiple *ToMany relationships or when limit offset is used with a *ToMany relationship). So you don't need to explicitly use FetchConfig and leave it up to Ebean if you wish.

Example: Normal “Fetch Join”

```
// Orders and their customers fetch in a single SQL query
List<Order> l0 = Ebean.find(Order.class)
    .fetch("customer")
    .findList();
```

Example: “Query Join” … results in 2 SQL queries used to build the object graph

```
// 2 SQL statements are used to build this object graph
// The first SQL query fetches the Orders and the second
// SQL query fetches customers
List<Order> l0 = Ebean.find(Order.class)
    .fetch("customer", new FetchConfig().query())
    .findList();
```

The reason for using “Query Joins” as opposed to “Fetch joins” is that there are some cases where using multiple queries is more efficient that a single query.

Any time you want to load multiple OneToMany associations it will likely be more performant as multiple SQL queries. If a single SQL query was used that would result in a Cartesian product.

There can also be cases loading across a single OneToMany where 2 SQL queries (using Ebean "query join") can be more efficient than one SQL query (using Ebean "fetch join"). When the "One" side is wide (lots of columns) and the cardinality difference is high (a lot of "Many" beans per "One" bean) then this can be more efficient loaded as 2 SQL queries.

Example: Two “Query Joins” results in 3 SQL queries used to build this object graph

```
// A more advanced example with multiple query joins
List<Order> l0 = Ebean.find(Order.class)
 .select("status, shipDate")

 .fetch("details", "orderQty, unitPrice", new FetchConfig().query())
 .fetch("details.product", "sku, name")

 .fetch("customer", "name", new FetchConfig().query(10))
 .fetch("customer.contacts","firstName, lastName, mobile")
 .fetch("customer.shippingAddress","line1, city")
 .findList();
```

The resulting 3 sql queries are:

```
// query 1 … the main query
<sql summary='Order' >
select o.id c0, o.status c1, o.ship_date c2, o.customer_id c3
from o_order o
</sql>
```

Query 1 - Note: customer_id was automatically added to support query join.

```
// query 2 … query join on customer
<sql mode='+query' summary='Customer, shippingAddress
+many:contacts' load='path:customer batch:10 actual:2' >
select c.id c0, c.name c1
        , cs.id c2, cs.line_1 c3, cs.city c4
        , cc.id c5, cc.first_name c6, cc.last_name c7, cc.mobile c8
from o_customer c
left outer join o_address cs on cs.id = c.shipping_address_id
left outer join contact cc on cc.customer_id = c.id
where c.id in (?,?,?,?,?,?,?,?,?,?)
order by c.id
</sql>
```

Query 2 - fetching the first 10 customers referenced (batch:10) but there where actually only 2 to fetch (actual:2).

```
// query 3 … query join on details
<sql mode='+query' summary='Order +many:details, details.product'
load='path:details batch:100 actual:3' >
select o.id c0
        , od.id c1, od.order_qty c2, od.unit_price c3
        , odp.id c4, odp.sku c5, odp.name c6
from o_order o
left outer join o_order_detail od on od.order_id = o.id
left outer join o_product odp on odp.id = od.product_id
where o.id in (?,?,?,?,?,?,?,?,?,?,?,?,?,?,?,?,?,?,?,?,?,?)
order by o.id
</sql>
```

Query 3 – fetching the order details for the first 100 orders (batch:100).

6.5.1 FetchConfig.lazy() - “Lazy Joins”

If a join is not defined at all (neither a fetch join or a query join) – then lazy loading will by default just fetch all the properties for that entity.

FetchConfig.lazy() allows you to control that lazy loading query – define the batch size, properties to select and also fetch paths to include on the lazy load query.

This is very similar to a “query join” except that the loading occurs on demand (when the property is requested and not already loaded).

The reason you would want to control the lazy loading query is to optimise performance for further lazy loading (avoid N+1 queries, define joins that should be included for lazy loading queries, load only the properties required and no more).

Example: Control the query used to lazy load

```
// control the lazy loading of customers ...
List<Order> list = Ebean.find(Order.class)
    .fetch("customer","name", new FetchConfig().lazy(5))
    .fetch("customer.contacts","contactName, phone, email")
    .fetch("customer.shippingAddress")
    .where().eq("status",Order.Status.NEW)
    .findList();
```

In the example above the orders are loaded. Only when the application requests a

customer property (that is not the customer's id) then the lazy loading of the customer is invoked. At that point the customer name is loaded, with the contacts and shippingAddress – this is done in batch of 5 customers.

Note that if the customer status is requested (rather than the customer name) and that invokes the lazy loading then all the customer's properties are loaded (rather than just the customers name).

```
Order order = list.get(0);
Customer customer = order.getCustomer();

// this invokes the lazy loading of 5 customers
String name = customer.getName();
```

The resulting lazy loading query is …

```
<sql mode='+lazy' summary='Customer, shippingAddress +many:contacts'
load='path:customer batch:5 actual:2' >
select c.id c0, c.name c1
        , cs.id c2, cs.line_1 c3, cs.line_2 c4, cs.city c5,
cs.cretime c6, cs.updtime c7, cs.country_code c8
        , cc.id c9, cc.phone c10, cc.email c11
from o_customer c
left outer join o_address cs on cs.id = c.shipping_address_id
left outer join contact cc on cc.customer_id = c.id
where c.id in (?,?,?,?,?)
order by c.id
</sql>
```

6.5.2 Using both - new FetchQuery.queryFirst(100).lazy(10);

You can use both queryFirst() and lazy() on a single join. The queryFirst() part defines the number of beans that will be loaded eagerly via an additional query and then lazy defines the batch size of the lazy loading that occurs after than (if there is any).

6.5.3 +query and +lazy – query language syntax

To define “query joins” and “lazy joins” in the query language you can use +query and +lazy. Optionally you can specify the batch size for both.

```
find order
join customers (+query )
where status = :status
```

```
find order (status, shipDate)
join customers (+lazy(10)  name, status)
where status = :orderStatus
```

6.6 Asynchronous Query Execution – findFutureList() etc

Ebean has built in support for executing queries asynchronously. These queries are executed in a background thread and “Future” objects are returned.

The “Future” objects returned extend ***java.util.concurrent.Future***. This provides support for cancelling the query, checking if it is cancelled or done and getting the result with waiting and timeout support.

```
// Methods on Query for ansychronous execution

public FutureList<T> findFutureList();

public FutureIds<T> findFutureIds();

public FutureRowCount<T> findFutureRowCount();
```

Example: Using FutureList

```
Query<Order> query = Ebean.find(Order.class);

// find list using a background thread
FutureList<Order> futureList = query.findFutureList();
```

```
// do something else ...

if (!futureList.isDone()){
    // you can cancel the query. If supported by the JDBC
    // driver and database this will actually cancel the
    // sql query execution on the database
    futureList.cancel(true);
}
// wait for the query to finish … no timeout
List<Order> list = futureList.get();

// wait for the query to finish … with a 30sec timeout
List<Order> list2 = futureList.get(30, TimeUnit.SECONDS);
```

6.7 PagingList Query

PagingList is used to make it easy to page through a query result. Paging through the results means that instead of all the results are not fetched in a single query Ebean will use SQL to limit the results (limit/offset, rownum, row_number() etc).

Instead of using PagingList you could just use setFirstRow() setMaxRows() on the query yourself. If you are building a stateless application (not holding the PagingList over multiple requests) then this approach is a good option.

For Stateful applications PagingList provides some benefits.

- Fetch ahead (background fetching of the next page via a FutureList query)
- Automatic propagation of the persistence context
- Automatically getting the total row count (via a FutureRowCount query)

So with PagingList when you use Page 2 it will automatically fetch Page 3 data in the background (using a FutureList query). The persistence context is automatically propagated meaning that all the paging queries use the same persistence context.

```
int pageSize = 10;
```

```
PagingList<TOne> pagingList =
    Ebean.find(TOne.class)
          .where().gt("name", "2")
          .findPagingList(pageSize);


// get the row count in the background...
// ... otherwise it is fetched on demand
// ... when getTotalRowCount() or getTotalPageCount()
// ... is called
pagingList.getFutureRowCount();

// get the first page
Page<TOne> page = pagingList.getPage(0);

// get the beans from the page as a list
List<TOne> list = page.getList();


int totalRows = page.getTotalRowCount();

if (page.hasNext()) {
    Page<TOne> nextPage = page.next();
    ...
}
```

In a stateless application you should set fetch ahead to false as you are not going to benefit from it.

```
PagingList<TOne> pagingList =
    Ebean.find(TOne.class)
          .where().gt("name", "2")
          .findPagingList(10);

// fetchAhead not useful in a stateless application
pagingList.setFetchAhead(false);

Page<TOne> firstPage = pagingList.getPage(0);
```

7 Autofetch

7.1.1 What is Autofetch

Autofetch is an idea that has come from Ali Ibrahim's work at the University of Texas. Ali and colleagues developed the idea and are working on a Hibernate implementation of Autofetch.

Link: http://www.cs.utexas.edu/~aibrahim/autofetch/

In short, Autofetch is a feature of an ORM where the ORM will automatically tune your queries for optimal performance by using profiling information. The ORM will gather object graph usage (profiling) by the application for queries and then use this information to automatically tune the future queries (specify joins and properties to fetch etc).

In my opinion “Autofetch” will have a profound effect on how ORM's will define and execute queries (build object graphs). It provides a means for highly performant object graph traversal in a transparent manor – this is a ***very big deal*** !!!

7.1.2 Autofetch in Ebean

When I first came across “Autofetch” I was immediately sold on the idea. “Autofetch” is not a bolt on feature for Ebean but instead has been built into it's core internals and I see it as a hugely important feature for Ebean.

With the “Partial Object” support in Ebean, Autofetch is even sweeter as it automatically takes care of selecting just the properties that the application uses as well as handling the joins. This means you get the performance benefit of partial objects without any work on the part of the developer – which is important as your applications get bigger and more complex.

Explicit Control

On the query object you can explicitly specify if you want to use autofetch or not.

```
// explicitly turn on Autofetch for this query
query.setAutofetch(true);
```

Implicit Control

There are a number of properties in ebean.properties which control how autofetch works.

```
# enable autofetch
```

```
ebean.autofetch.querytuning=true

# enable collection of profiling information
ebean.autofetch.profiling=true

# implicit autofetch mode
# default_off, default_on, default_on_if_empty
ebean.autofetch.implicitmode=default_on

# minimum amount of profiling to collect before
# autofetch will start tuning the query
ebean.autofetch.profiling.min=1

# profile every query up to base
ebean.autofetch.profiling.base=10

# after base collect profiling on 5% of queries
ebean.autofetch.profiling.rate=0.05
```

property	*type/values*	*description*
ebean.autofetch.querytuning	boolean	If true enables Autofetch to tune queries
ebean.autofetch.profiling	boolean	If true enables profiling information to be collected
ebean.autofetch.implicitmode	default_off default_on default_on_if_empty	default_on_if_empty means autofetch will only tune the query if neither select() nor fetch() has been explicitly set on the query.
ebean.autofetch.profiling.min	integer	The minimum amount of profiled queries to be collected before the automatic query tuning will start to occur
ebean.autofetch.profiling.base	integer	Will profile every query up to this number and after than will profile based on the profiling.rate (5% of queries etc)
ebean.autofetch.profiling.rate	float	The percentage of queries that are profiled after the base number has been collected

JMX and Programatic Control of Autofetch

You can manage Autofetch at runtime either via JMX or programmatically.

```
// get the 'default' EbeanServer
EbeanServer server = Ebean.getServer(null);
AdminAutofetch adminAutofetch = server.getAdminAutofetch();

// change autofetch properties at runtime
adminAutofetch.setQueryTuning(false);
adminAutofetch.setProfiling(false);
adminAutofetch.setProfilingRate(0.50f);

adminAutofetch.clearProfilingInfo();
adminAutofetch.clearTunedQueryInfo();
```

8 Persistence Context

Although Ebean doesn't have an “Entity Manager” it does have a “persistence context”. In fact, you could go as far to say that any ORM worth using needs a “persistence context”.

8.1 Definition

JPA v1.0 specification - section 5.1

“A persistence context is a set of managed entity instances in which *for any persistent entity* ***identity*** *there is a* ***unique entity instance***. Within the persistence context, the entity instances and their lifecycle are managed by the entity manager.”

Ebean has a “Persistence Context” to ensure ... unique entity instances ***BUT*** Ebean has a different approach to lifecycle management.

That is, Ebean has a persistence context to ensure “... unique entity instance.” (the blue section of JPA's definition) but has a different approach to the lifecycle management compared with JPA. Ebean has no entity manager and no persist/merge/flush lifecycle methods.

8.2 Unique Entity Instances

Ebean uses the “persistence context” for queries and lazy loading (when it is building object graphs). The purpose of this is to ensure that 'consistent' object graphs are constructed (1 unique instance per identity).

For example, in fetching a list of orders and their customers … the persistence context ensures that you only get 1 customer instance for it's given id (e.g. you are ***NOT*** allowed to have 2 or more instances of “customer id=7”.

You could even say that any ORM worth using needs a persistence context when it builds object graphs from relational result sets due to the nature of relational result sets.

For example, if you didn't have a persistence context and did allow 2 or more instances of “customer 7” … and modified one instance but not the other … things get very ugly. The “persistence context” ensures the user/application code works with unique entity instances.

8.3 Lifecycle Management

Ebean has a different approach to lifecycle management. The core difference is that with Ebean each bean itself has it's own dirty checking (detects when it has been modified and holds it's old/original values for optimistic concurrency checking).

With JPA implementations generally the dirty checking is performed by the entity manager. The entity manager generally holds the old/original values for optimistic concurrency checking and the beans need to be 'attached' to an entity manager to be 'flushed' (as an

insert/update/delete). [Note: JDO based JPA implementations do this a bit differently].

PRO's for Ebean's approach

- No need to manage Entity Manager's
- save/delete simpler that attached/detached beans with persist/merge/flush etc

CON's for Ebean's approach

- Ebean makes an assumption that scalar types are immutable. Most scalar types (String, Integer, Double, Float, BigDecimal etc) are immutable, however, some such as java.util.Date are not. What this means is that with Ebean if you mutate a java.util.Date Ebean will ***NOT*** detect the change – instead you have to set a different java.util.Date instance.

8.4 Transaction scoped

With Ebean the persistence context is transaction scoped. This means that when you begin a new transaction (implicitly or explicitly) Ebean will start a new persistence context.

The persistence context uses weak references and lives beyond the end of a transaction. This enables any lazy loading occuring after the transaction ends to use the same persistence context that the instance was created with.

This means, a persistence context

- Starts when a transaction starts
- Is used during the transactions scope to build all object graphs (queries)
- Lives beyond the end of a transaction so that all lazy loading occuring on that object graph also uses the same persistence context

8.5 Multi-threaded object graph construction

The persistence context is designed/implemented to be thread safe. Ebean internally can use multiple threads for object graph construction (query execution) and I expect some advanced users to take an interest in this (and may want to do this in their own application code).

When Ebean uses background threads for fetching (findFutureList, PagingList, useBackgroundFetchAfter etc) Ebean will automatically propagate the persistence context. In these cases multiple threads can be using the same persistence context concurrently.

This also enables Ebean to provide more advanced multi-threaded query strategies in the future.

8.6 Persistence Context as a “first level cache”

The persistence context is sometimes described as the “first level cache”. I have also seen

it described as the “transactional cache” in that it is scoped to a transaction or longer.

```
// a new persistence context started with the transaction
Ebean.beginTransaction();
try {
    // find "order 72" results in that instance being put
    // into the persistence context
    Order order = Ebean.find(Order.class, 72);

    // finds an existing "order 72" in the persistence context
    // ... so just returns that instance
    Order o2 = Ebean.find(Order.class, 72);
    Order o3 = Ebean.getReference(Order.class, 72);

    // all the same instance
    Assert.assertTrue(order == o2);
    Assert.assertTrue(order == o3);

} finally {
    Ebean.endTransaction();
}
```

The code above shows that there is only 1 instance of “Order 72”. As we try to fetch it again (during the scope of a single persistence context) we end up getting back the same instance.

However, typically you don't write code that fetches the same Order multiple times in a single transaction. The code above is not something you would typically write.

A more realistic example would be when the persistence context is used:

```
// a new persistence context started with the transaction
Ebean.beginTransaction();
try {
    // find "customer 1" results in this instance being
    // put into the persistence context
    Customer customer = Ebean.find(Customer.class, 1);

    // for this example … "customer 1" placed "order 72"

    // when "order 72" is fetched/built it has a foreign
    // key value customer_id = 1...

    // As customer 1 is already in the persistence context
    // this same instance of customer 1 is used
```

```
    Order order = Ebean.find(Order.class, 72);
    Customer customerB = order.getCustomer();

    // they are the same instance
    Assert.assertTrue(customer == customerB);

} finally {
    Ebean.endTransaction();
}
```

From these examples you should hopefully see that the persistence context acts as a cache to some degree. It can sometimes reduce the number of database queries required when you get object graphs and navigate them.

However, the primary function of the persistence context is to ensure ... unique instances for a given identity (so that the object graphs are constructed in a consistent manor). The fact that it sometimes looks/acts like a cache is more of a side effect.

9 Caching (L2 Server cache)

When we want to talk about caching for performance we are talking about the “Level 2” cache or the “server cache”. It is called the “Level 2 cache” because the persistence context is often referred to as the “Level 1 cache”.

The goal of the L2 server cache is to gain very significant performance improvement by not having to hit the database.

9.1 Bean and Query caches

Ebean has 2 types of caches – Bean caches and Query caches.

Bean Caches

Bean caches hold entity beans and are keyed by their Id values.

Query Caches

Query caches hold the results of queries (Lists, Sets, Maps of entity beans) and are keyed by the query hash value (effectively a hash of the query and its bind values).

The entries in a query cache are invalidated by ***ANY*** change to the underlying table – insert, update or delete. This means that the query cache is only useful on entities that are infrequently modified (typically “lookup tables” such as countries, currencies, status codes etc).

9.2 Read Only and Shared Instances

For a performance optimisation when using the cache you can inform Ebean that you want “read only” entities. If you ask for “read only” entities Ebean can give you the instance that is in the cache rather than creating a new copy (creating a new instance and copying the data from the cached instance).

To be safe in allowing many threads to share the same instances (from the cache) Ebean ensures that these instances can not be mutated. It sets flags (sharedInstance=true, readOnly=true) and any attempt to modify the entity (via setters or putfields) results in an IllegalStateException being thrown.

```
// Cache countries. Use readOnly=true so unless explicitly
// stated in the query we will return read only/shared instances
@CacheStrategy(readOnly=true,warmingQuery="order by name")
@Entity
@Table(name="o_country")
public class Country {
```

Note that Countries is a good candidate for a default setting of readOnly=true. This is because (for my application) country information is very rarely changed. The application code mostly treats the countries as read only.

Now, whenever we get a country (via direct query or indirectly via relationships/joins) unless we explictly say query.setReadOnly(false) we are going to get back readOnly instances that we will not be able to mutate.

```
// we will use the cache .. and the instance
// in the cache is returned to us (not a copy)
Country country = Ebean.find(Country.class, "NZ");

// this country instance is readOnly
Assert.assertTrue(Ebean.getBeanState(country).isReadOnly());

try {
    // we can't modify a readOnly bean
    // …  a IllegalStateException is thrown
    country.setName("Nu Zilund");
    Assert.assertFalse("Never get here",true);
```

```
} catch (IllegalStateException e){
    Assert.assertTrue("This is readOnly",true);
}

// explicitly state we want a MUTABLE COPY
// … not the same instance as the one in cache
// … a copy is made and returned instead
Country countryCopy = Ebean.find(Country.class)
    .setReadOnly(false)
    .setId("NZ")
    .findUnique();

// we can mutate this one
countryCopy.setName("Nu Zilund");

// save it, automatically maintaining the cache ...
// evicting NZ from the Country bean cache and
// clearing the Country query cache
Ebean.save(countryCopy);
```

9.3 Shared Instances

Ebean sets a sharedInstance flag on a bean whenever it is put into the cache. This is used to ensure that the bean is always treated in a read only fashion (and can be safely shared by multiple threads concurrently).

You can invoke lazy loading on a sharedInstance. When that occurs the sharedInstance flag is propagated to the lazily loaded beans. If you lazy load a collection (list, set or map) then the collection is also marked with the sharedInstance flag and that means you can't add or remove elements from the collection (list, set or map).

A sharedInstance and all its associated beans and collections are are all ensured to be read only and can be safely shared by multiple threads concurrently.

9.4 Automatic Cache Maintenance

When you save entity beans or use an Update or SqlUpdate, Ebean will automatically invalidate the appropriate parts of the cache.

If you save a entity bean that results in an update and there is a matching bean in the cache it will be evicted automatically from the cache at commit time.

If you save an entity bean that results in an insert then the bean cache is not effected.

Whenever ***ANY*** change is made (insert/update or delete) the entire query cache for that bean type is invalidated.

9.5 Handling External Modification (via stored procedures etc)

When you save/delete beans via Ebean.save() and Ebean.delete() etc Ebean will automatically maintain its cache (removing cached beans and cached queries as appropriate). However, you may often find yourself modifying the database outside of Ebean.

For example, you could be using other frameworks, your own JDBC code, stored procedures, batch systems etc. When you do so (and you are using Ebean caching) then you can inform Ebean so that it invalidates appropriate parts of its cache.

```
// inform Ebean that some rows have been inserted and updated
// on the o_country table.
// … Ebean will maintain the appropriate caches.
boolean inserts = true;
boolean updates = true;
boolean deletes = false;
Ebean.externalModification("o_country", inserts, updates, deletes);

// clearAll() caches via the ServerCacheManager ...
ServerCacheManager serverCacheManager =
    Ebean.getServerCacheManager();

// Clear all the caches on the default/primary EbeanServer
serverCacheManager.clearAll();

// clear both the bean and query cache
// for Country beans ...
serverCacheManager.clear(Country.class);

// Warm the cache of Country beans
Ebean.runCacheWarming(Country.class);
```

9.6 @CacheStrategy - automatically using the bean cache

The easiest way to use caching is to specify the @CacheStrategy annotation on the entity class. This means that Ebean will try to use the bean cache as much as possible when it fetches beans of that type.

```
// Cache countries. Use readOnly=true so unless explicitly
// stated in the query we will return read only/shared instances
@CacheStrategy(readOnly=true,warmingQuery="order by name")
@Entity
@Table(name="o_country")
public class Country {
```

```
// automatically use the cache
Country country = Ebean.find(Country.class,"NZ");

// references automatically use the cache too
Country countryRef = Ebean.getReference(Country.class,"NZ");

// hit the country cache automatically via join
Customer customer = Ebean.find(Customer.class, 1);
Address billingAddress = customer.getBillingAddress();
Country c2 = billingAddress.getCountry();
```

ReadOnly

The readOnly attribute of @CacheStrategy is used to determine if by default Ebean should return the same instance from the cache (instances in the cache are readOnly and effectively immutable) or whether Ebean should create a new instance and copy the data from the cached bean onto the new instance.

The readOnly attribute of @CacheStrategy is the “default” Ebean will use unless you explicitly specify the readOnly attribute of the query.

```
// explicitly state we want a MUTABLE COPY
// … not the same instance as the one in cache
// … a copy is made and returned instead
Country countryCopy = Ebean.find(Country.class)
    .setReadOnly(false)
    .setId("NZ")
    .findUnique();

// we can mutate this one
countryCopy.setName("Nu Zilund");

// save it, automatically maintaining the cache ...
// evicting NZ from the Country bean cache and
// clearing the Country query cache
Ebean.save(countryCopy);
```

9.7 Manually specifing to use the bean cache

If you don't use @CacheStrategy you can programmatically specify to use the bean cache via query.setUseCache(true);

```
// explicitly state we want to use the bean cache
Customer customer = Ebean.find(Customer.class)
    .setUseCache(true)
    .setId(7)
    .findUnique();

// use readOnly=true to return the 'sharedInstance'
// from the cache (which is effectively immutable)
Customer customer = Ebean.find(Customer.class)
    .setUseCache(true)
    .setReadOnly(true)
    .setId(7)
    .findUnique();
```

9.8 Using the Query Cache

To use the query cache you have to explicitly specify its use on a query.

```
// use the query cache
List<Country> list = Ebean.find(Country.class)
    .setUseQueryCache(true)
    .where().ilike("name", "New%")
    .findList();
```

The query cache is generally useful for returning lists that are very infrequently changed. These lists would often be used to populate drop down lists / combo boxes in user interfaces.

If you are familiar with the term “Lookup Tables” or “Reference Tables” these are typical candidates for using cached queries. Some examples of lookup/reference tables could be, countries, currencies and order status.

Query cache lists are readOnly by default

```
// by default the lists returned from the query
// cache are readOnly.  Use setReadOnly(false) to
// return mutable lists
List<Country> list = Ebean.find(Country.class)
    .setUseQueryCache(true)
    .setReadOnly(false)
    .where().ilike("name", "New%")
    .findList();
```

10 Id Generation

- DB Identity / Autoincrement
- DB Sequences
- UUID
- Custom ID Generation

There are 4 ways that ID's can be automatically generated for new Entities. This occurs when a entity is going to be inserted and it does not already have an Id value.

The first 3 options are highly recommended for 2 reasons.

1) They are standard approaches that can also be used by other programs, stored procedures, batch loading jobs etc that could be written in other languages etc. That is, if you choose a custom ID Generation then this can make it more difficult to use other programs / tools to insert into the DB.
2) They support good concurrency – can you really do better?

Most Databases support Sequences or Identity/Autoincrement. DB2 and H2 support both.

10.1 UUID Id Generation

To use UUID's with Ebean all you need to do is use the UUID type for your id property. Ebean will automatically assign an appropriate UUID Id generator.

```
@Entity
public class MyEntity {

    @Id
    UUID id;
    ...
```

10.2 DB Sequences / DB Autoincrement

Refer: com.avaje.ebean.config.dbplatform.DatabasePlatform & DbIdentity

For each database type (Oracle, MySql, H2, Postgres etc) there is a specific DatabasePlatform which defines whether the database supports sequences or autoincrement. This then defines whether DB sequences or DB Identity / Autoincrement will be used. This also provides a sequence generator specific to that database.

For DB sequences the NamingConvention is used to define the default name of the sequences. This name will be used unless the sequence name is explicitly defined via annotations.

What this means is that, typically you only need to the the @Id annotation unless you need to override a sequence name (when it doesn't match the naming convention).

```
@Entity
public class MyEntity {

    @Id
    Integer id;
    ...
```

10.3 Batched fetch of Db Sequences

For performance reasons we don't want to fetch a sequence value each time we want an Id. Instead we fetch a 'batch' of sequences (refer to ServerConfig setDatabaseSequenceBatchSize()) - the default batch size is 20.

Also note that when the number of available Id's for a given sequence drops to half the batch size then another batch of sequences is fetched via a background thread.

For Oracle, Postgres and H2 we use Db sequences. It is worth noting that this allows the use of JDBC batch statements (PreparedStatement.addBatch() etc) which is a significant performance optimization. You can globally turn on the use of JDBC batching via ServerConfig.setUsePersistBatching() … or you can turn it on for a specific Transaction.

11 Mapping

11.1 Goal of Mapping

> The main goal of "Mapping" is to isolate the application code from the Database Schema.

This means that *some* changes can occur to the schema without breaking the application.

The application code can be written without reference to the specific table names, view names and column names. This means that your application can more easily withstand some unforseen changes.

11.2 JPA Mapping

Ebean uses the same mapping as per the JPA specification. You can learn and use the same mapping annotations. This is generally a very good part of the specification and I'd expect this part of the specification to mostly stand the test of time.

11.3 DDL Generation

Ebean v2.0 introduces support for DDL generation.

The DDL the is generated is useful for agile development and testing. It is also useful to help get an understanding of the mapping.

For simple Databases the DDL generated will be sufficient but for large databases it is not really 'production quality'. For large Databases you will likely use it as a starting point. DBA's will want to add more control over physical aspects of Tables and Indexes (specify tablespaces etc to spread IO across disks, partition large tables, control freespace depending on the etc).

11.4 Naming Convention

Ebean has a Naming Convention API to map column names to property names. It also maps entity to table names and can take into account database schema and catalog if required.

Refer to: com.avaje.ebean.config.NamingConvention

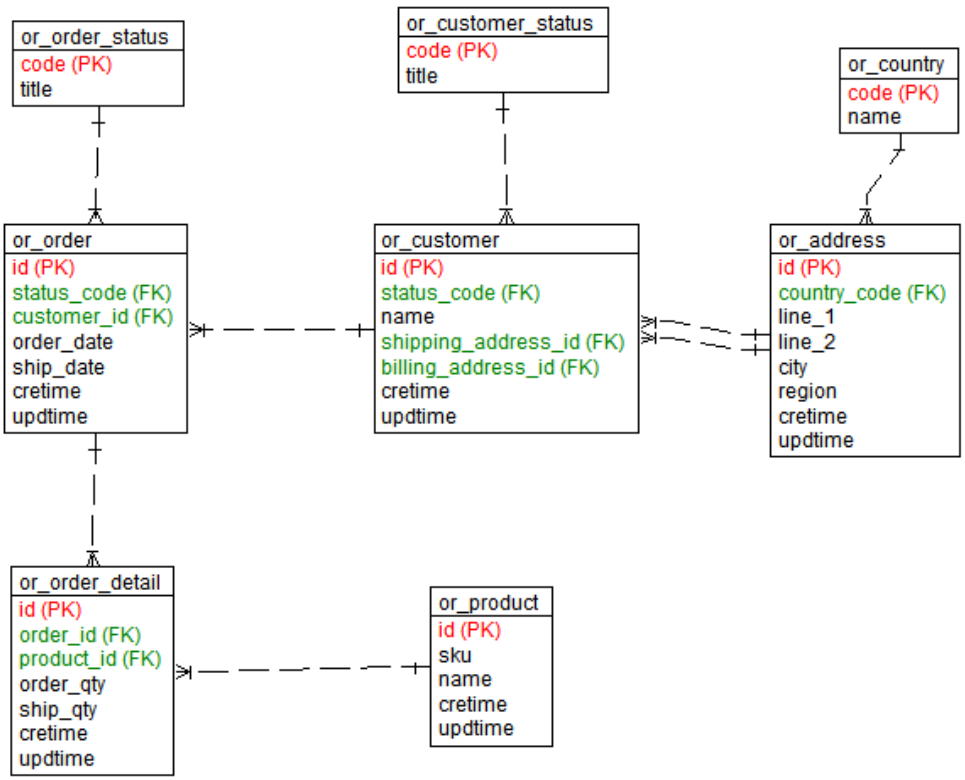
The default UnderscoreNamingConvention converts column names with underscores into normal java camel case property names (e.g. “first_name” maps to “firstName”).

You can also use the MatchingNamingConvention or implement your own.

11.5 Example Database Design

The following is a database design with some tables. It is a fairly typical Orders, Customers, Products design that I will use to illustrate the Mapping.

[1,1]



11.6 Mapping Annotations

11.6.1 Basics

@Entity

This simply marks an Entity Bean. Ebean has the same restrictions as per the JPA spec with the entity beans requiring a default constructor and with properties following the java beans conventions (with getter and setter names).

@Table

Here you can specify the table name that the entity bean will use. More specifically this is the “base table” as an entity bean could have a number of “secondary” tables as well.

@Id and @EmbeddedId

Use one of these to mark the property that is the id property. You should use @Id if the id property is a simple scalar type (like Integer, String etc) and you should use @EmbeddedId if the id type is complex (an embedded bean).

@Column

Use this if the naming convention does not match the bean property name to the database column or if you need to use quoted identifiers. Otherwise it is not required.

@Lob

This marks a property that is mapped to a Clob/Blob/Longvarchar or Longvarbinary.

@Transient

This marks a property that is not persistent.

11.6.2 Relationships

Database Design and Normalisation

If you are familiar with Database design and normalisation then I believe the relationships will become clear fairly quickly. If you are not familiar then I'd recommend you take a look at these topics (a quick trip to wikipedia) as I believe they will help you a lot in this area of ORM mapping.

DB Foreign Keys and ORM Relationships

Assuming your DB has foreign keys and has been well designed then the ORM mapping should follow quite naturally. If you DB has a more “interesting” design then the ORM

mapping can be a lot more painful with more compromises.

One-to-Many relationship

This is probably the most common relationship so we will start with a One to Many relationship. The @OneToMany and the @ManyToOne represent the two ends of a relationship. This relationship typically maps directly to a Database Foreign Key constraint...

Database Foreign Key constraint

A typical database design is full of “One to Many/Many to One” relationships implemented using a Foreign Key constraint. A foreign key constraint has an “imported” side and an “exported” side.

“A Customer has many Orders”

“An Order belongs to a Customer”

The customer table “exports” its primary key to the order table

The order table “imports/references” the customer table's primary key.

A Foreign Key constraint can be viewed from the exported side (customer table) or the imported side (order table).

... @OneToMany and @ManyToOne map directly to this.

The Customer entity bean...

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

    ...
    @OneToMany
    List<Order> orders;
```

The Order entity bean...

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

    ...
    @ManyToOne
    Customer customer;
```

Because the @OneToMany and @ManyToOne are both mapped this is a "Bidirectional" relationship. You can navigate the object graph in either direction.

Unidirectional Relationships

To turn a Bidirectional relationship into a Unidirectional relationship you need to either remove the @OneToMany (Customer.orders property) or the @ManyToOne (Order.customer property).

Removing a OneToMany - no problem

eg. Remove List<Order> from Customer

You can generally remove the OneToMany side of a relationship without any major issues. The issue being that you can not navigate the object graph in that direction.

Why remove a OneToMany? Sometimes the OneToMany side is not useful to the application or even dangerous if used. For example on Product there could be a List<OrderDetail> but that could be considered useless or even dangerous if it was navigated (and all order details for a product where lazy loaded).

Removing a ManyToOne - watch those inserts...

eg. Remove Customer from Order

If you remove a ManyToOne this effects how a bean is saved (specifically the insert). The reason is because it is the ManyToOne (importing) side of the relationship that holds the foreign key column (e.g. or_order table holds the customer_id foreign key column).

Q: If the customer property is removed from the Order object how would you specify which customer placed an order when you create a new order?

In Database speak this translates to ... when inserting an order how is the customer_id column populated?

A: You have to use cascading save on the customer.orders and save the customer. Sounds like a pain... and it would be in this case... lets look at a more realistic case where you want to remove a ManyToOne...

eg. Remove Order from OrderDetail

Lets say you remove the Order property from the OrderDetail bean. Now lets say you want to write some code to add a OrderDetail to an Order (insert). How do you specify which Order it should go to?

"Turn on" cascade save on the @OneToMany side

```
@Entity
@Table(name="or_order")
public class Order {
    ...
    // must cascade the save
    @OneToMany(cascade=CascadeType.ALL)
    List<OrderDetail> details;
```

And save the order... which cascade saves the details

```
// create or fetch the order
Order order = ...
List<OrderDetail> details = new ArrayList<OrderDetail>();
OrderDetail orderDetail = ...
details.add(orderDetail);

// set the new details...
order.setDetails(details);

// save the order... which cascade saves
// the order details...
Ebean.save(order);
```

So when the order is saved, because the @OneToMany relationship has cascade.ALL the save is cascaded to all the order details.

Note that you can update OrderDetails individually (without relying on cascade save) but to insert a new OrderDetail we are relying on the cascading save.

Removing a ManyToOne typically reflects a strong "ownership" relationship. The Order "owns" the OrderDetails, they are persisted as

one via cascade save.

Managed Relationships = @OneToMany + cascade save

If cascade save is on a @OneToMany when the save is cascaded down from the 'master' to the 'details' Ebean will 'manage' the relationship.

For example, with the Order - OrderDetails relationship when you save the order Ebean will get the order id and make sure it is 'set' on all the order details. Ebean does this for both Bidirectional relationships and Unidirectional relationships.

What this means is that if your OrderDetails has an @ManyToOne Order property (its bidirectional) you do not need to set the order against each orderDetail when you use cascade save. Ebean will automatically set the 'master' order to each of the details when you save the Order and that cascades down to the details.

@OneToMany Notes

When you assign a @OneToMany you typically specify a mappedBy attribute. This is for Bi-directional relationships and in this case the “join” information is read from the other side of the relationship (meaning you don't specify any @JoinColumn etc on this side).

If you don't have a mappedBy attribute (there is no matching property on the other related bean) then this is a Unidirectional relationship. In this case you can specify a @JoinColumn if you wish to override the join column information from the default).

```
@Entity
@Table(name="s_user")
public class User implements Serializable {

    // unidirectional …
    // … can explicitly specify the join column if needed
    @OneToMany
    @JoinColumn(name=”pref_id”)
    List<Preference> preferences;

    // bi-directional
    // … join information always read from the other side
    @OneToMany(mappedBy="userLogged")
    List<Bug> loggedBugs;
```

One-to-One relationship

A One-to-One relationship is exactly the same as a One-to-Many relationship except that the many side is limited to a maximum of one.

That means that one of the @OneToOne sides operates just like a @ManyToOne (the imported side with the foreign key column) and the other @OneToOne operates just like a @OneToMany (exported side).

So you put the mappedBy on the 'exported side' – as if it was a @OneToMany.

From a Database perspective a One to One relationship is implemented with a foreign key constraint (like one to many) and adding a unique constraint on the foreign key column. This has the effect of limiting the “many” side to a maximum of one (has to be unique).

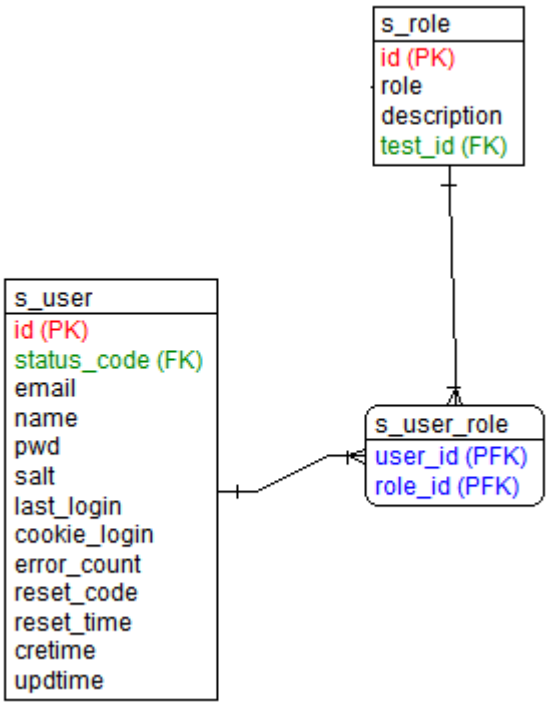
Many-to-Many relationship

You are probably aware that there are no Many to Many relationships in a physical database design. These are implemented with an intersection table and two One to Many relationships.

Lets look at an example...

A User can have many Roles

A Role can be assigned to many Users



A Many to Many between user and role

In the database diagram above there is an intersection table called s_user_role. This represents a logical many to many relationship between user and role.

Q: When is a Many to Many better represented as two One to Many relationships?

A: If there is ever an additional column in the intersection table then you should consider changing this from a Many to Many to two One to Many's and including the intersection table in the model.

One way to think of this is that each @ManyToMany operates just like it was a @OneToMany. The relationship must be “managed” meaning Ebean must take care of inserting into and deleting from the intersection table.

The way this works is that any additions or removables from the many list/set/map are noted. These become inserts into and deletes from the intersection table.

```
@Entity
@Table(name="s_user")
public class User implements Serializable {
    ...
    @ManyToMany(cascade=CascadeType.ALL)
    List<Role> roles;
```

```
@Entity
@Table(name="s_role")
public class Role {
    ...
    @ManyToMany(cascade=CascadeType.ALL)
    List<User> users;
```

The intersection table name and foreign key columns can default or be specified by @JoinTable etc.

The following code shows a new role added to a user.

```
User user = Ebean.find(User.class, 1);
List<Role> roles = user.getRoles();

Role role = Ebean.find(Role.class, 27);

// adding a role to the list...this is remembered and will
// result in an insert into the intersection table
// when save cascades...
roles.add(role);

// save cascades to roles... and in this case
// results in an insert into the intersection table
Ebean.save(user);
```

Note that if a role was removed from the list this would result in an appropriate delete from the intersection table.

11.6.3 @Formula

Formula can be used to get read only values using SQL Literals, SQL Expressions and SQL Functions.

With a Formula the ***${ta}*** is a special token to represent the table alias. The table alias is dynamically determined by Ebean and you can put the ${ta} in the select or join attributes.

A SQL Expression

Example: The caseForm field using a SQL case expression

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


    @Id
    Integer id;

    @Formula(select="(case when ${ta}.id > 4 then 'T' else 'F' end)")
    boolean caseForm;

...
```

Note the **${ta}** in place of the table alias

Note in this deployment 'T' and 'F' are mapped to boolean values.

A SQL Function

```
@Formula(select="(select count(*) from f_topic _b where _b.user_id =
${ta}.id)")
int countAssiged;
```

The formula properties can be used as normal properties. This includes in query select and where expressions.

```
// include the countAssigned property
```

```
Query<User> query = Ebean.createQuery(User.class);
query.select("id, name, countAssiged");
query.fetch("topics");

List<User> list = query.findList();
```

The SQL generated from the query above is:

```
<sql summary='[app.data.User]'>
select u.id, u.name,
    (select count(*) from f_topic _b where _b.user_id = u.id)
from s_user u
</sql>
```

Note the ***"u"*** in the sql has replaced the ${ta} [table alias placeholder] specified in the select attribute of the formula.

It is also worth noting that this is potentially not great SQL!!! You should check SQL in this form (get the explain plan for the query – get your DBA to review the sql etc) but there is a good chance the sub query (select count(*) ... _b.user_id = u.id) will effectively execute for each row returned. If that is the case the query above can quickly become expensive and you may find you have an unhappy DBA .

The above can be re-written to use a view (if one exists). The benefit is that we can use a join rather than a subquery which can perform much better from a database perspective.

```
// VIEW: vw_topic_aggr
// Lets say there is a view base on this SQL.
// It is typically more performant to JOIN
// to a view rather than use a subquery

create view vw_topic_aggr as
select user_id, max(id) as topic_max, count(*) as topic_count
from f_topic
group by user_id
```

And use the **join** attribute of @Formula

```
@Formula(
  select="_b${ta}.topic_count",
  join="join vw_topic_aggr as _b${ta} on _b${ta}.user_id = id")
  int countWithJoin;
```

Now, if the view does not exist we can do something similar ...

In this next @Formula the join attribute contains the select effectively replacing the vw_topic_aggr view with (select user_id, max(id) as topic_max, count(*) as topic_count from f_topic group by user_id).

```
@Formula(
  select="_b${ta}.topic_count",
  join="join (select user_id, max(id) as topic_max, count(*) as
topic_count from f_topic group by user_id) as _b${ta} on _b$
{ta}.user_id = id")
int countWithJoin;
```

```
Query<User> query = Ebean.createQuery(User.class);
query.select("id, name, countWithJoin");
List<User> list = query.findList();
```

Results in the following SQL: - *will generally perform better than the subquery*

```
<sql summary='[app.data.User]'>
select u.id, u.name, _bu.topic_count
from s_user u
join (select user_id, max(id) as topic_max, count(*) as topic_count
from f_topic group by user_id) as _bu on _bu.user_id = id
</sql>
```

It is also worth noting that the formula fields can be used in where expressions.

Example: where countWithJoin > 1

```
Query<User> query = Ebean.createQuery(User.class);
query.select("id, name, countWithJoin");
// using the formula field in the where
query.where().gt("countWithJoin", 1);

List<User> list = query.findList();
```

Resulting SQL:

```
<sql summary='[app.data.User]'>
select u.id, u.name, _bu.topic_count
from s_user u
join (select user_id, max(id) as topic_max, count(*) as topic_count
from f_topic group by user_id) as _bu on _bu.user_id = id
where _bu.topic_count > ?
</sql>
```

11.6.4 @EnumMapping

This is an Ebean specific annotation (not part of JPA) for mapping Enum's to database values. The reason it exists is that IMO the JPA approach for mapping of Enums is highly dangerous (in the case of Ordinal mapping) or not very practical (in the case of String mapping).

Lets take the example of this Enumeration:

```
public enum UserStatus {
    ACTIVE, INACTIVE, NEW
}
```

Enum Ordinal Mapping is Dangerous

In my opinion JPA Ordinal Mapping for Enum's is very dangerous (So dangerous I highly recommend avoiding it). The reason is because the ordinal values for Enum depends on the order in which they appear.

```
public class TestStatus {

    public static void main(String[] args) {

          int ord0 = UserStatus.ACTIVE.ordinal();
          int ord1 = UserStatus.INACTIVE.ordinal();
          int ord2 = UserStatus.NEW.ordinal();

          // 0, 1, 2
          System.out.println("ord 0:"+ord0+" 1:"+ord1+" 2:"+ord2);

          String str0 = UserStatus.ACTIVE.name();
          String str1 = UserStatus.INACTIVE.name();
          String str2 = UserStatus.NEW.name();

          // "ACTIVE", "INACTIVE", "NEW"
          System.out.println("str 0:"+str0+" 1:"+str1+" 2:"+str2);
    }
}

OUTPUT:

ord 0:0 1:1 2:2
str 0:ACTIVE 1:INACTIVE 2:NEW
```

The problem is that if you change the order of the Enum elements such as in this example (DELETED is now first with Ordinal value of 0) ...

```
public enum UserStatus {
    DELETED, ACTIVE, INACTIVE, NEW
}
```

With the above code the Ordinal values for ACTIVE, INACTIVE and NEW have all changed. This is a very subtle change and now every status existing in the database will be incorrectly represented in the application. Hopefully this issue would be picked up quickly but there could be situations where this subtle data issue is not picked up before a real disaster has occured.

Enum String mapping is limited

It is more likely that your DBA would prefer to save space by mapping this to a VARCHAR(1) column and use “A”, “I”, “N” and “D” as codes to represent ACTIVE, INACTIVE, NEW and DELETED.

The issue with the String mapping is that more frequently than not the names of the Enumeration elements will have to be comprimised to short less-meaningful names to map into DB values or your DBA will be unhappy with long wasteful values.

@EnumValue

```
public enum UserStatus {

    @EnumValue(“D”)
    DELETED,

    @EnumValue(“A”)
    ACTIVE,

    @EnumValue(“I”)
    INACTIVE,

    @EnumValue(“N”)
    NEW
}
```

With EnumValue (Ebean specific annotation) you explicitly specify the value map the entry to. This Annotation has been logged with the Eclipselink project in the hope it makes it's way into the JPA spec.

12 Transactions

12.1 Transaction Logging

Ebean also has transaction logging built in (SQL, bind variables etc).

The transaction logs are useful to follow the behaviour of JDBC batching, cascading behaviour and identify when lots of lazy loading is being invoked.

Also, if you are developing relatively complex batch processing you should note that you can ***add your own comments*** *so that they* ***appear in the transaction log***. This makes it easier to relate the statements in the transaction log back to your application code.

```
...
try {
  Ebean.execute(new TxRunnable() {
    public void run() {
      // this comment appears in the transaction log
      Ebean.currentTransaction().log("-- saving holder first time");
      Ebean.save(holder);
      ...
```

You can control what is logged and the level of detail via ebean.properties.

```
## Use java util logging to log transaction details
#ebean.loggingToJavaLogger=true

## General logging level: (none, explicit, all)
ebean.logging=all

## location of transaction logs
ebean.logging.directory=logs
#ebean.logging.directory=${catalina.base}/logs/trans

## Specific Log levels (none, summary, binding, sql)
ebean.logging.iud=sql
ebean.logging.query=sql
ebean.logging.sqlquery=sql
```

In your early stages of using Ebean you should find where the transaction logs are going so that you can see exactly what Ebean is doing.

When Ebean starts it will output to the log the directory where the transaction logs will get written to.

```
...
INFO: Entities enhanced[0] subclassed[38]
INFO: Transaction logs in: C:/apps/tomcat6/logs/trans
...
```

12.2 Implicit Transactions

If you do execute a query or use save() or delete() without a transaction (without a @Transaction annotation, TxRunnable , beginTransaction() etc) then Ebean will create an “implicit” transaction.

Ebean first checks to see if there is a current transaction, and if there is not one it creates one, performs the action (query, save() delete() etc) and then commits the transaction automatically at the end (or performs a rollback if there was an error).

```
// execute a query will create an implicit
// transaction if there is no current Transaction
List<User> users =
                Ebean.find(User.class)
                      .fetch("customer")
                      .where().eq("state", UserState.ACTIVE)
                      .findList();

// execute a save will create an implicit
// transaction if there is no current Transaction
Ebean.save(user);
```

Following are ways to explicitly demarcate your transactions via @Transactional annotation and various programmatic approaches (TxRunnable etc).

12.3 @Transactional annotation

The Transaction annotation is only usable if you use “Enhancement” (via the IDE Plugin, ANT task or javaagent) .

The transformer (enhancement code) that enhances Entity beans will also by default look for any class that has a @Transactional annotation and enhances those classes/methods

adding in the appropriate transactional scope management (create a transaction if required, commit or rollback as necessary, suspend and resume an existing transaction if required etc).

The annotation follows the Spring approach of supporting specific transaction isolation levels, explicit rollback/no rollback of exceptions, and read only indicator. These are currently missing from the standard EJB annotation.

EJB: No Rollback of Checked Exceptions

It seems counter-intuitive but in EJB (and Spring decided follow the EJB approach) a checked exception does not actually cause a rollback to occur. So if you had a method that is Transactional and throws a checked exception (not a RuntimeException)... then it ***would NOT rollback*** if that checked exception was thrown.

Ebean: configurable ... default is to rollback on any Exception

With Ebean this is not the default behaviour and instead Ebean by default will rollback on any Exception (Checked or Runtime). You can make Ebean follow the EJB behaviour by setting ebean.transaction.rollbackOnChecked=false.

```
## Ebean default is rollbackOnChecked=true
## set this to false to get EJB behaviour
ebean.transaction.rollbackOnChecked=false
```

Put the @Transactional annotation on a method. Via Enhancement (IDE Plugin, javaagent or ANT task) Ebean enhances the method adding the transaction management around the method invocation.

```
...
public class MyService {

    @Transactional
    public void runFirst() throws IOException {

          System.out.println("runFirst");
          User u1 = Ebean.find(User.class, 1);

          runInTrans();
    }

    @Transactional(type = TxType.REQUIRES_NEW)
```

```
public void runInTrans() throws IOException {

      System.out.println("runInTrans ...");
      User u1 = Ebean.find(User.class, 1);
      ...
```

This supports nested transactions (as does TxRunnable and TxCallable) and makes it easy to demarcate transactions.

You can also put @Transactional on interface methods, and classes implementing the interface will inherit the transactional definitions from the interface.

Note that the precedence of reading specific @Transactional attributes is that it first uses the annotation attributes from the local method (if there is one), if not then it tries the local class level (if there is one), then the interface method (if there is one), then the interface class level (if there is one).

BUG: If you are using the eclipse IDE enhancer plugin, then you should note there is a bug.

If you change the @Transactional annotation on an interface, this change is not reflected in the implementation classes until they are saved/compiled. Changing the interface will not necessarily force a save/comile of all the classes that implement that interface.

The workaround for this is that after you change a @Transactional annotation on an interface to make sure all implementation classes of that interface are saved/compiled perform a build all.

12.4 Programatic: TxRunnable and TxCallable

You can get the same type of functionality as @Transactional programmatically via the TxRunnable and TxCallable.

```
public void myMethod() {
  ...
  System.out.println(" Some code in myMethod...");

  // run in Transactional scope...
  Ebean.execute(new TxRunnable() {
    public void run() {

          // code running in "REQUIRED" transactional scope
          // ... as "REQUIRED" is the default TxType
          System.out.println(Ebean.currentTransaction());

          // find stuff...
          User user = Ebean.find(User.class, 1);
          ...

          // save and delete stuff...
          Ebean.save(user);
          Ebean.delete(order);
          ...
    }
  });

  System.out.println(" more code in myMethod...");
}
```

You can specify a TxScope with options such as isolation level and rollback / noRollback for specific Exception types.

```
TxScope txScope = TxScope
                .requiresNew()
                .setIsolation(TxIsolation.SERIALIZABLE)
                .setNoRollbackFor(IOException.class);

Ebean.execute(txScope, new TxRunnable() {
    public void run() {
          ...
```

You can use a mixture of @Transaction and TxRunnable / TxCallable with nesting (transactional methods calling other transactional methods etc). They handle the suspending and resuming of nested transactions for you.

TxType has values REQUIRED (the default), REQUIRES_NEW, MANDATORY, SUPPORTS, NOT_SUPPORTS, NEVER. These are an exact match of the EJB TransactionAttributeTypes.

12.5 Programatic: beginTransaction()

You can also write transaction demarcation code in a more “traditional” way using a try finally block.

```
Ebean.beginTransaction();
try {

    User u = Ebean.find(User.class, 1);
    ...
    Ebean.commitTransaction();

} finally {
    Ebean.endTransaction();
}
```

The code above will generally use a ThreadLocal to hold the Transaction to begin, commit and end (end will perform a rollback if required). This makes it easy to use but the limitation is that you can only have one active transaction per EbeanServer (The above code is using Ebean (rather than EbeanServer) so it is a transaction against the *“default”* EbeanServer).

12.6 Programmatic: createTransaction()

With all the previous transaction approaches Ebean helps manage the transaction. The limitation this imposes is that you can only have one active transaction per EbeanServer per Thread.

You can have “nested” transactions when you use REQUIRES_NEW. The “outer” transaction is suspended – the method is run with its NEW transaction and then the “outer” transaction is resumed. With “nested” transactions you can only use 1 transaction

at a time – there is only 1 active transaction at any given moment.

There is extra API on the EbeanServer for using Transactions more explicitly. These transactions are created and not managed by Ebean – so you can have any number of them but you need to manage them yourself (make sure you commit or rollback).

In practice the easiest way to do this is to use a try finally block and have transaction.end() in the finally block. The transaction.end() will perform a rollback but only if the transaction has not already been committed.

```
// explicit transaction API is on EbeanServer
// get the default server...
EbeanServer server = Ebean.getServer(null);


// create a transaction not "managed" by Ebean
Transaction transaction = server.createTransaction();
try {

    Query<User> query = ...;
    // query using explicit transaction
    server.findList(query, transaction);

    User user = ...;
    // save using explicit transaction
    server.save(user, transaction);

    // delete using explicit transaction
    server.delete(order, transaction);


    transaction.commit();

} finally {
    // rollback if required
    transaction.end();
}
```

Using your own explicit transactions like the code above means that you are not restricted to one transaction per EbeanServer per Thread, but it does mean you need to manage the transaction making sure you commit or rollback – otherwise the transaction may be lost resulting in a connection pool leak.

12.7 Spring Transactions

The ebean-spring module includes integration into Spring transaction managers.

This means that you can use a Spring Transaction Manager to control the transactions and Ebean will use those transactions. Ebean registers with Spring's TransactionSynchronizationManager and is notified off commits and rollbacks – this enables Ebean to automatically manage its server caches and invoke listeners etc so there are no limitations in using the Spring TransactionManager with Ebean.

13 Data Types

Compound Types

Scalar Types

Enum

Boolean

Date / Timestamp / util Date / util Calender

Defining / Registering a new Scalar Type

14 Field Access and Property Access

Field and Property access are the two ways in which a field/property is managed/intercepted. What this means is that the ORM needs to intercept the reading or writing of either Fields or Properties.

An ORM does this interception to provide lazy loading and dirty checking functionality and an ORM achieves this using either Enhancement or Subclass generation (Subclass generation is also referred to as “Proxy classes”).

To explain Field Access and Property Access I'll use an example focusing on a single field (in this case “firstName”) and how Property Access and Field Acccess works in Ebean.

```
...
@Entity
public class Person {

    @Id
    int id;
    String firstName;
    String lastName;

    ...
    // An extra method... NOT a getter...
    // here to highlight the difference between
    // Field and Property Access
    public String getFullName() {
          return firstName + " " + lastName;
    }

    public String getFirstName() {
          return firstName;
    }

    public void setFirstName(String firstName) {
          this.firstName = firstName;
    }

    // getters and setters for id and lastName not shown...
    ...
```

14.1 Property Access

By “Property” we really mean a Java Bean Property. In the case of the “firstName” property this means the related getter and setter methods of getFirstName() and setFirstName(...). Refer to the Java Bean naming convention for clarification.

With Property Access you ***MUST*** have a setter and a getter.

With Property Access ***ONLY*** the getter and setter are intercepted. If you use the field in other methods then access to that field is not intercepted/managed (See the getFullName() method as an example).

14.1.1 Property Access with Ebean ...

```
...
public String getFullName() {
    // not getter or setter so NOT intercepted...
    return firstName+" "+ lastName;
}

public String getFirstName() {
    _ebean_intercept.preGetter("firstName");
    return firstName;
}

public void setFirstName(String newValue) {
  _ebean_intercept.preSetter("firstName", newValue, getFirstName());
  this.firstName = newValue;
}
```

This is an example of how Ebean modifies/generates code to support “property access” (NB: other ORMs could generate quite different code).

The modified/generated class now has a ***_ebean_intercept*** field.

In calling getName() the preGetter() method is called on the intercept field and this will invoke lazy loading if required.

In calling setName() getName() is called which in turn can invoke lazy loading. The getName() returns the “Old Value” / “Previous Value” of the name. The preSetter() method is called on the intercept and will maintain the “Dirty Checking” information.

If your entity beans just have getter and setter methods they are well suited to property access. From a design perspective they may also be called “Anemic” in the sense that

they don't have any logic and can almost be viewed as pure data.

14.2 Field Access

Field access in general is implemented via bytecode Enhancement (Also refered to as Weaving and Transformation).

The class bytes are modified replacing the GETFIELD and PUTFIELD bytecode calls to “persistent fields” with method calls.

With Ebean Field access looks something like this... (note: other ORMs could generate quite different code)

```
public String getFullName() {
    // GETFIELD firstName replaced...
    return _ebean_getFirstName()+" "+_ebean_getLastName();
}
public String getFirstName() {
    // GETFIELD firstName replaced
    return _ebean_getFirstName();
}

public void setFirstName(String newValue) {
    // PUTFIELD firstName replaced
    _ebean_setFirstName(newValue);
}

// new method generated by Ebean
// used to replace GETFIELD firstName
String _ebean_getFirstName() {
    _ebean_intercept.preGetter("firstName");
    return this.firstName;
}

// new method generated by Ebean
// used to replace PUTFIELD firstName
void _ebean_setFirstName(String newValue) {
    String currentValue = _ebean_getFirstName();
    _ebean_intercept.preSetter("firstName", newValue, currentValue);
    this.firstName = newValue;
}
```

With Field access it actually doesn't matter where the field is used the GETFIELD and PUTFIELD byte code instructions are replaced. It is important to note that in the example

above the getFullName() method is also modified to support the interception/mangement by the ORM.

Unlike property access field access ***does not require*** the getter or setter to even exist – they can be removed if you want.

Essentially with Field access you can write whatever code you like (within reason) and it will behave as expected (ORM interception occuring as and how you would expect).

14.2.1 JPA – Choose access approach on a per Field / Property basis

With JPA you choose between field or property access on a per field / property basis. If you put the annotation on the field this implies that you want field access and if you put the annotation on the associated getter method of the property then you should get property access.

Ebean does not work this way!!!

14.2.2 Ebean – Field vs Property access

Ebean has not followed the JPA approach. With Ebean you get Field Access when you use Enhancement and you get Property Access when you use Subclassing.

The reason for this comes down to...

- Keeping things simple
- Field Access is the preferred approach
- Subclassing approach introduces security / visibility issues for Field Access

In fact when building Ebean v0.9.8 I was personally pretty keen to do away with Property access altogether. This would have meant also removing support for the Subclassing approach. This would have meant making Ebean internally simplier...

... but I got talked out of it...

There are a few downsides to Field Access.

- Serialization: Your entity beans now include and depend on Ebean objects. Where you deserialize them you will need some Ebean classes in you class path.
- Using JAVAAGENT or ANT or ... to support the enhancement can be a pain. Ebean has tried to make this as simple and robust as possible.
- Java Web Start does not support JAVAAGENT. However, you could enhance the classes at build time using an ANT task.

14.2.3 Ebean interception notes

Id Interception

Id fields/properties are not intercepted ever (either for reading or writing). That is, getting or setting an ID will never invoke lazy loading or cause the entity to be marked as dirty.

OneToMany and ManyToMany Interception

The List Set and Map persistent properties are not intercepted when they are being set. That is, setting a associated many property does not invoke lazy loading or mark the entity as dirty.

When you get a entity back from a query every associated many property (List Set or Map) will have a 'proxy' in place. It is not until you invoke a method on the List Set or Map proxy that lazy loading of that proxy will occur.

A reference object (obtained by Ebean.getReference()) will lazy load the entity bean if a List Set or Map associated many property is read (via getter etc).

toString() interception

No interception is invoked by a toString() method. The reason for this decision is that toString() is often invoked by an IDE while running a debugger – in the past this has lead to confusion to have lazy loading being invoked accidentally in this fashion.

equals() and hashCode() generation

If your entity bean does not have a equals() or hashCode() method then one will be generated for you based on the identity property using the “” technique.

The generated equals() or hashCode() methods do not invoke lazy loading.

@Transient on methods

If you are using Ebean enhancement (Field Access) then you can put the @Transient annotation on a method and it is ***NOT*** intercepted. Specifically the method is left exactly as it is (rather than having the GETFIELD PUTFIELD byte codes replaced with intercepted method calls).

This is an Ebean extension to the JPA spec. This enables you to write a method on an entity bean that you know will not have its field access intercepted.

14.2.4 Rob Opinion: The future is Field Access... lets make it work

I'm a fan of Field Access in that a developer can write their entity bean however they like (except Ebean still requires a default constructor). You can write any type of business logic into your Entity bean and it will work correctly (in terms of ORM lazy loading and dirty

checking).

Enhancement is the natural way to support field access so I believe it's Ebean's job to make Enhancement easy work for the developers (ANT tasks, IDE Integration as well as JAVAAGENT support).

Q: JPA supports the mixing of Field and Property access on the same entity bean. Will Ebean look to support this?

A: No. If you can do field access then you should. On the web there is mention of benefits to using Property access such as performance and type conversion benefits. None of these apply to Ebean. I see no justification or benefits to mixing the access type but I do see problems and confusion, so there is no plan to mix the access types on a single entity bean.

Q: So why does Ebean support Property Access again?

A: Because Ebean supports the Subclassing approach (also known as “Dynamic Proxy approach”). The Subclassing approach lends itself to Property access due to the fact that the generated subclass is defined in a different ClassLoader to the original Class. This means that in a normal java environment that generated subclass can not access the non-public fields of the original class.

Q: How does the enhancement work for with Groovy or Scala entity beans?

A: You can use Ebean enhancement on Groovy or Scala beans. Note this includes support for Scala's properties (which don't follow the Java Bean Spec getter setter method naming conventions). Ebean can tell if it is a Groovy bean or Scala bean and take that into account as needed.

15 Enhancement and Subclass Generation

There are two main techniques used by Java ORM vendors to support their features.

1. Enhancement (aka Weaving) – using a javaagent or an ANT task etc to enhance the classes prior to class loading (at load time or build time).
2. Subclass Generation (aka Dynamic Proxy) - Using a ClassLoader internal to the ORM and generating Subclasses of your entity beans dynamically (also known as the “Dynamic Proxy” approach).

Ebean supports both approaches.

Both of these approaches have their pros and cons. This section is aimed at explaining the two approaches and how they work in Ebean so that you can make an informed choice.

Ebean supports Enhancement / Weaving via javaagent, ant task and an Eclipse IDE plugin. When Ebean starts up Ebean will check if the entity classes are enhanced and if not will automatically generate a Subclass and use that instead.

Rob Opinion: I believe the enhancement approach will be the better approach over the long term with less restrictions on the code. You don't need getters or setters and enhanced classes allowing any logic to be put into the entity beans (you need to be more careful with the Subclass generation approach).

15.1 Why are these techniques needed?

All the Java ORM's that I am aware of need this to support “Lazy Loading” and that includes Ebean.

In addition to that Ebean (and some other ORMs) use these techniques to support “Dirty Checking” / “Optimistic Concurrency Checking”. That is, detecting when an entity bean has been modified (made dirty) and maintaining original values to support Optimistic Concurrency Checking.

In short, ORMs use these techniques to intercept calls to getters, setters and other methods to magically invoke lazy loading as required and maintain “Dirty Checking” information. These techniques (Enhancement and Subclass generation) enable this magic to occur.

15.2 Why are these techniques popular?

The reason this is a popular approach is that your entity beans do not need to implement any special interface or extend any special class.

In this example Person does not extend a special class or implement a special interface.

```
package model;

import java.sql.Date;
import javax.persistence.Entity;
import javax.persistence.Id;

@Entity
public class Person {

    @Id
    int id;
    String firstName;
    String lastName;
    Date dob;

    public int getId() {
          return id;
    }
    public void setId(int id) {
          this.id = id;
    }
    public String getFirstName() {
          return name;
    }
    public void setFirstName(String name) {
          this.name = name;
    }
    ...
}
```

15.3 ... and in Eclipselink, OpenJPA and Hibernate

Rob Opinion: It is my understanding/opinion as at May 10 that ...

Eclipselink doesn't support the Subclassing approach. This is interesting because I'd guess this project has the most resources behind it and it is the JPA 2.0 reference implementation.

OpenJPA supports both approaches but strongly suggests to users to use Enhancement approach (javaagent or ant) for production deployment.

Hibernate supports both approaches but IMO the majority of hibernate users will be using the Subclassing approach. This is possibly because it is simplier where you don't have to

muck around with javaagent or ant etc.

In my opinion, the reason many people dislike the Enhancement approach (via javaagent or ant task etc) is because many have had problems getting this to work (trouble configuring and using javaagent, finding configuration files etc)... or because it slows down the... code – compile – run cycles (you have to run the ant task or make sure a javaagent is configured etc). In my opinion the Eclipse Ebean Enhancer plugin provides a good solution in that the classes are enhanced transparently whenever you save in the IDE. This is the approach I recommend for Eclipse users of Ebean.

Note that there is also an IDEA enhancer for Ebean so please ask for that if required.

15.4 Subclass Generation

Subclass generation is a technique made popular by Hibernate and also supported in OpenJPA and Ebean.

How this works in Ebean is that for each entity bean a new class is generated that is a subclass of that entity bean.

For the Person entity a subclass Person$$EntityBean would be generated...

```
package model;

...
import com.avaje.ebean.bean.EntityBean;

public class Person$$EntityBean extends Person implements EntityBean
{
...
```

The bytes for the Person$$EntityBean class are generated by Ebean and then defined using a ClassLoader.

Note: When you query the ORM returning a Person you actually get a Person$$EntityBean class. This can confuse people.

```
Person p = Ebean.find(Person.class, 1);

System.out.println("vanilla: "+Person.class.getName());
System.out.println("subclass:"+p.getClass().getName());
```

output:

```
vanilla:  app.data.Person
subclass: app.data.Person$$EntityBean$testdb
```

Note: The ClassLoader that defines the Person$$EntityBean class is not the same ClassLoader than loaded the Person class. This can lead to visibility and security issues such as Person$$EntityBean not being able to see fields or methods in Person.

This is essentially why with Ebean the subclassing approach uses Property access and not field acccess.

```
Bug bug = Ebean.find(Bug.class, 1);

System.out.println("ClassLoaders...  ");
System.out.println("vanilla:  "+Bug.class.getClassLoader());
System.out.println("subclass: "+bug.getClass().getClassLoader());
```

output:

```
ClassLoaders...
vanilla:  sun.misc.Launcher$AppClassLoader@11b86e7
subclass: com.avaje.ebean.enhance.subclass.SubClassFactory@1db7df8
```

15.4.1 Ebean's Subclass Generation – Property access

As part of the initialisation of an EbeanServer it gathers all the meta data about all the entity beans etc. As part of this process for each entity bean Ebean checks to see if the class is already enhanced. If the class implements the EntityBean interface then it has already been enhanced via javaagent, ant, Eclipse Ebean Enhancer plugin etc.

If the class has not been enhanced then Ebean will generate a subclass of that entity bean type. In the case of the Person entity bean this would look something like...

```
package model;

...
import com.avaje.ebean.bean.EntityBean;
```

```
public class Person$$EntityBean extends Person implements EntityBean
{

// intercept field is added
EntityBeanIntercept _ebean_intercept;
...

public Person$$EntityBean() {
    super();
    _ebean_intercept = new EntityBeanIntercept(this);
}

// property access is used
public String getFirstName() {
    _ebean_intercept.preGetter("firstName");
    return super.getFirstName();
}

// property access is used
public void setFirstName(String newValue) {
  _ebean_intercept.preSetter("firstName", newValue, getFirstName());
  super.setFirstName(newValue);
}

// Other property getters and setters ...

// Other methods added to implement EntityBean ...

public EntityBeanIntercept _ebean_getIntercept() { ... }
public Object _ebean_createCopy() { ... }
public void _ebean_setField(...) { ... }
public Object _ebean_getField(...) { ... }


...
```

Note that when the subclass is generated Property access is used (Field access is not available for subclass generation in Ebean). The reason for this is that the Person$ $EntityBean.class is defined in a different ClassLoader to that of the Person.class – and by default Java security restricts access by Person$$EntityBean to the public and protected fields and methods of Person.class.

That is, Person$$EntityBean is in a different “Runtime Package” as per JVM spec.

“At run time, a class or interface is determined not by its name alone, but by a pair: its fully qualified name and its defining class loader. Each such class or interface belongs to a single runtime package. The runtime package of a class or interface is determined by the package name and defining class loader of the class or interface.”

http://java.sun.com/docs/books/jvms/second_edition/html/ConstantPool.doc.html#72007

http://java.sun.com/docs/books/jvms/second_edition/html/ConstantPool.doc.html#75929

The ***big upside*** of the subclass generation approach is that there is no “configuration” required – no javaagent or ant task to execute. You can just write you entity beans and test them.

The ***big downside*** of this approach is that Ebean's implementation uses Property access. That is, field access is not available with subclass generation due to JVM security.

Rob Opinion: Subclass Generation is easy to get working (no javaagent setup etc)... but I personally feel that Field access is the better approach long term – specifically allowing more freedom in how the entity beans are coded. In my opinion it's a matter of making the enhancement process easy to use. The Eclipse IDE plugin to automatically enhance the beans as they are saved is one way to make enhancement easy and I am keen to make that work well.

15.5 Enhancement

I am using the term “Enhancement” to cover all the ways (javaagent, ant, IDE plugin etc) that are used to modify the class in question – as opposed to generating a new class (subclass generation).

Other similar terms to Enhancement that are used around the place include Weaving, Transformation and byte code manipulation.

LTW – Load time weaving is used to refer to when the class manipulation occurs at “load time” typically via javaagent compared with class manipulation occuring at “build time” typically via an ANT task or IDE plugin.

In raw terms a class is just a byte[] and we are simply manipulating those bytes prior to the class being defined by its ClassLoader.

15.5.1 Ebean's Enhancement – Field access

The ANT Task, javaagent and the Eclipse IDE Enhancer plugin all use the same enhancement code.

Already Enhanced

As part of the enhancement Ebean detects if the bean has already been enhanced and if so will skip the enhancement. In this way it does not matter if you try to enhance a set of classes more than once or use a number of techniques (ANT Task, Enhancer Plugin and javaagent).

Mixing Enhanced and Subclassed entity beans

You can generally use a mixture of enhanced and subclassed entity beans but it is not a recommended approach. This could happen if you use ANT to enhance some but not all of the entity beans. The ones not enhanced will end up being "subclassed" (Any entity bean class that is not enhanced when an EbeanServer initialises will have a subclass generated for it).

However, you can ***NOT*** use a mixture of enhanced and subclassed beans for a given inheritance hierarchy. For a given inheritance heirarchy all the beans involved in that heirarchy need to be either enhanced or subclassed – not a mixture.

To avoid confusion it is recommended to use either Enhancement or Subclassing and not a mixture of both. Ebean will detect if there is a mixture of both and will log a warning.

Example: warning when some entity beans are enhanced and some are subclassed.

```
...
INFO: Entities enhanced[3] subclassed[5]
WARNING: Mixing enhanced and subclassed entities. Subclassed
classes:[User, OrderStatus, BugDetail, TestEntity, Order]
...
```

15.5.2 javaagent

Examples where d:/jarlib/ebean-agent-2.0.0.jar is the location of the ebean agent jar file...

-javaagent:d:/jarlib/ebean-agent-2.0.0.jar

-javaagent:d:/jarlib/ebean-agent-2.0.0.jar=debug=3

-javaagent:d:/jarlib/ebean-agent-2.0.0.jar=debug=3;packages=app.data.*,org.test.model.*

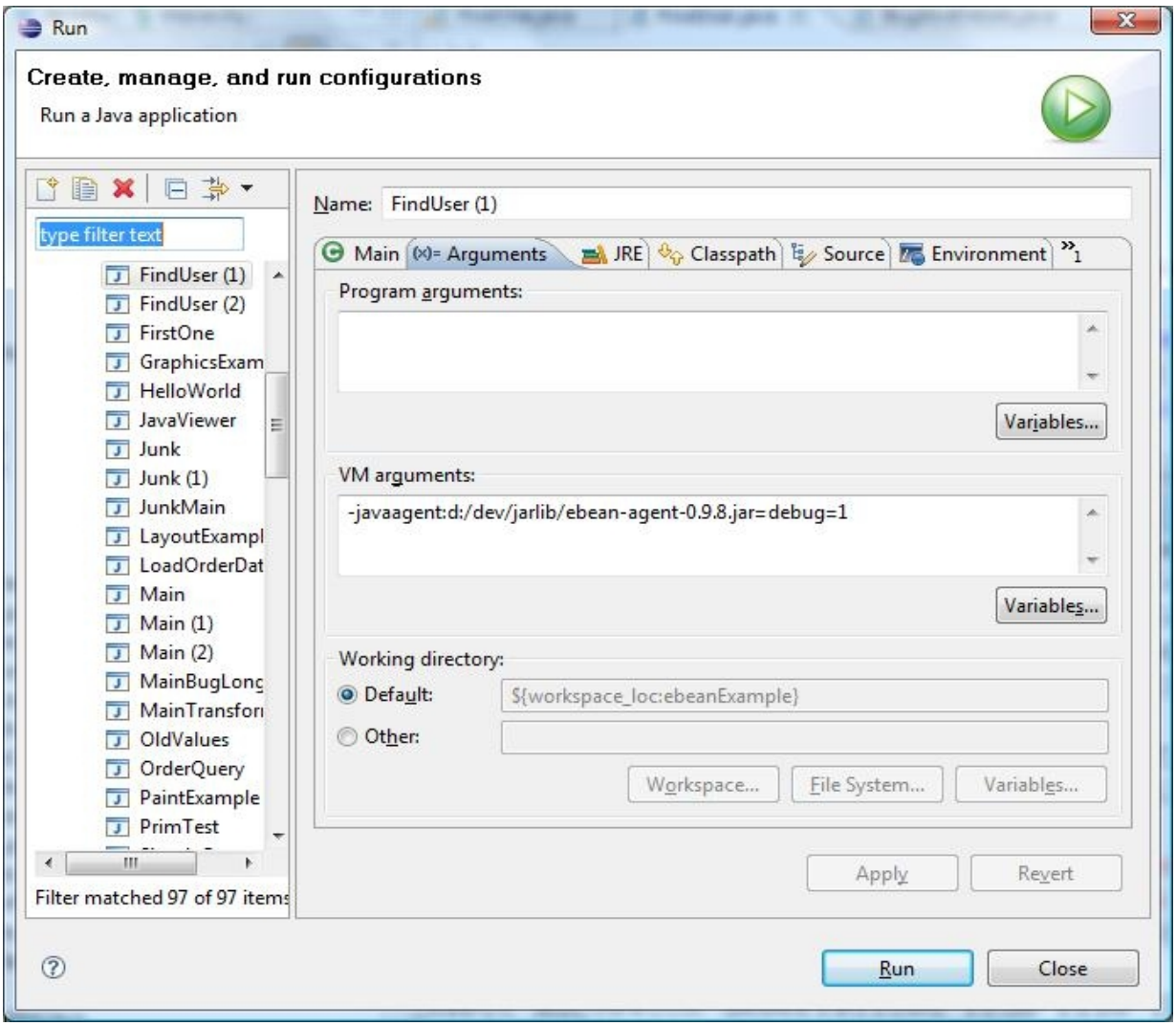
To use the javaagent approach you have to change the parameters passed to the JVM when it is started.

The Ebean javaagent can take 2 optional parameters:

debug: An int between 0 and 10, with 0 producing no output and 10 producing lots of debug output.

packages: a comma delimited list of packages. This is used to speed up the processing by skipping transformation on any class not in one of the listed packages.

The following images shows the Eclipse run dialog with the Ebean java agent specified in the VM arguments.



Rob Opinion: The javaagent approach has 3 known critisims:

Criticism 1: Slows the JVM Bootup – as every class is passed to transformers prior to being defined by the ClassLoaders. Ebean skips classes that are in well known packages such as known JDBC drivers, Sun classes, junit classes and common apache libraries. You can also use the packages parameter to only transform classes in those packages.

Criticism 2: Not supported by Java Web Start. You have to use ANT or IDE Plugin for this scenario.

Criticism 3: Not great in a Server setup like Tomcat. You really want to specify the javaagent on a per webapp basis rather than globally to the whole server. In the future I'm going to look at a specific Tomcat Loader to see if that is useful.

15.5.3 ANT Task

Modify your ant build.xml file to:

1. Define the AntEnhanceTask.

2. Create a target that uses the AntEnhanceTask to enhance the entity classes.

```
<taskdef name="ebeanEnhance"
   classname="com.avaje.ebean.enhance.ant.AntEnhanceTask"
   classpath="your_path_to/ebean-x.x.x.jar" />

<target name="ormEnhance" depends="clean,compile">
    <!-- eg. enhance entities in packages below app.entity -->
    <ebeanEnhance classSource="your_classes_directory"
     packages="app.entity*"
     transformArgs="debug=5" />

</target>
```

classSource: This is the directory that contains your class files. That is, the directory where your IDE will compile your java class files to, or the directory where a previous ant task will compile your java class files to.

classDest: The directory where the enhanced classes are written to. If not specified this defaults to the classSource effectively replacing the original class file with the enhanced class file.

packages: a comma delimited list of packages that contain entity classes. All the classes in these packages are searched for entity classes to be enhanced.

transformArgs: This contains a debug level (0 - 10) .

Rob Opinion:

The ANT Task is great. However, I personally found that using the ANT Task by itself was not satisfactory in that it negatively effected the ... CODE – COMPILE – RUN cycle. That is, having to run the ANT task slows down that cycle and is generally a pain in this scenario.

I think it is more likely that the ANT Task will be used as part of the build process (building a jar or war for deployment) and other approaches such as the IDE Plugin are better suited to use during the actual development CODE – COMPILE – RUN process.

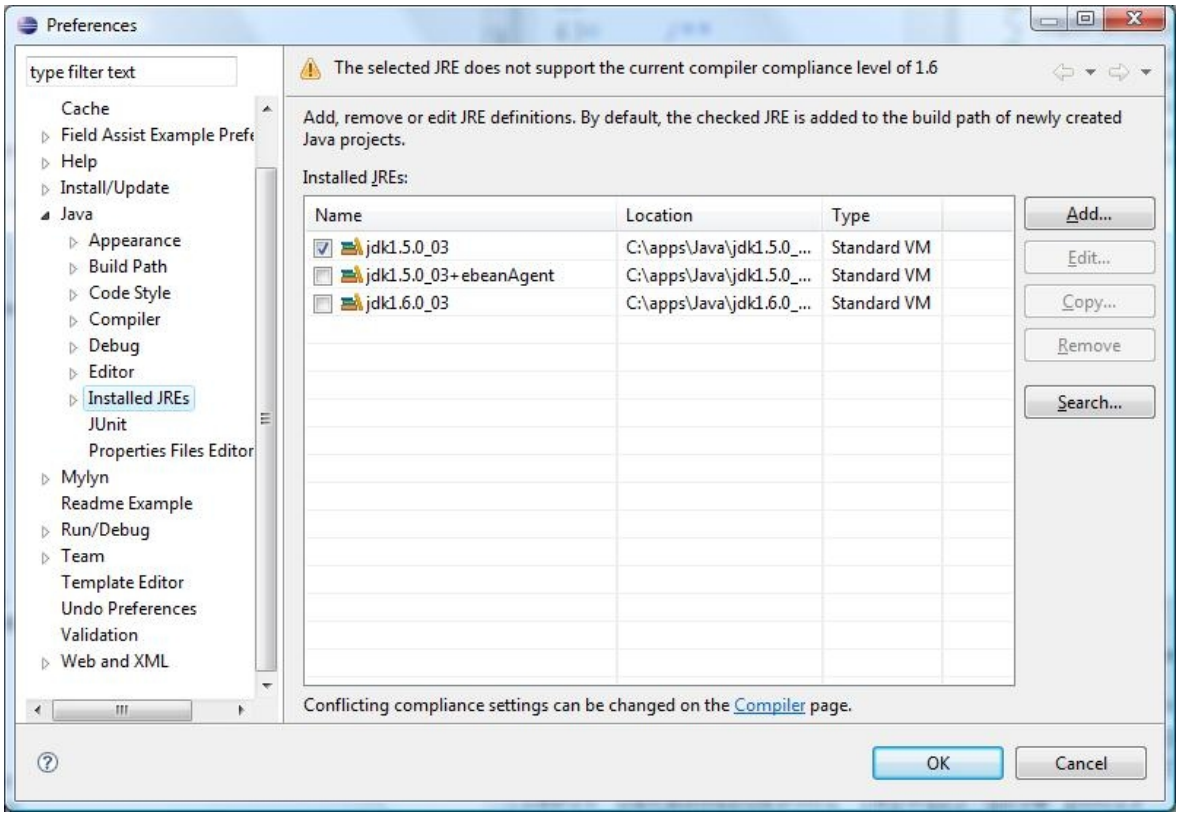
15.5.4 Eclipse IDE – Configure Project JRE

In Eclipse you can specify the javaagent parameter as the VM arguments in the run configuration. Eclipse – Run – Open Run Dialog – Arguments – VM Arguments.

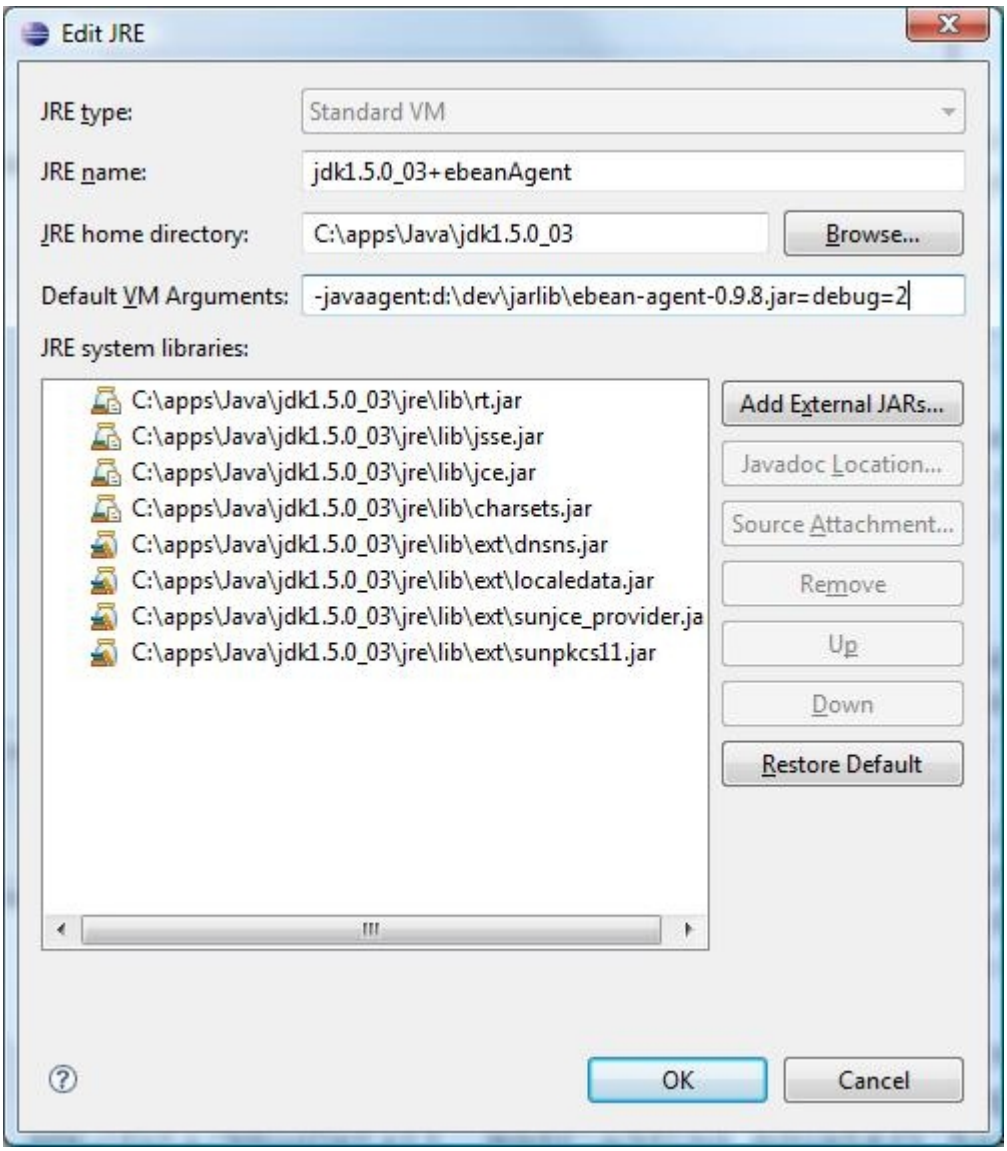
However, you would need to do this for each class with a main() method that you want to run. This could be tedious if you have a lot of different classes with main() methods.

Another approach is to register a new “Installed JRE” with the javaagent parameter set as the default VM argument and then use this JRE for your project rather than a normal JRE.

Eclipse – Preferences – Java – Installed JREs – Add



- Using the Browse button find a JRE (I'd find a JDK actually).
- Enter the Default VM Arguments (-javaagent ...)



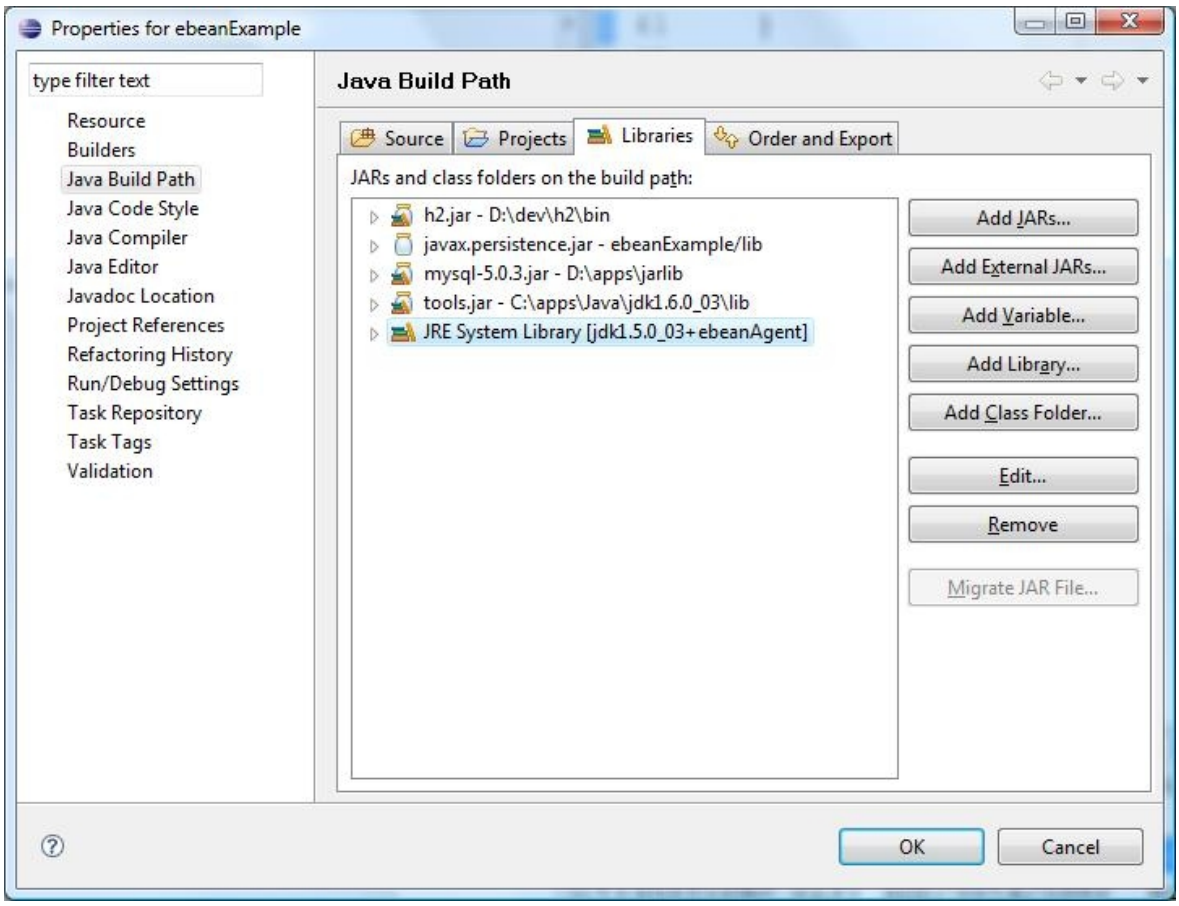
Eclipse – project – Build Path – Configure Build Path ...

Libraries – Add Library – JRE System Library – Alternate JRE...

(Choose your JRE that has the javaagent configured)

You now have two JRE Libraries in your projects build path. Remove the original one leaving you with the one that has the javaagent configured.

Your project Build Path should now look something like this (below) where my JRE configured with the javaagent is called “jdk1.5.0_03+ebeanAgent”



Now, whenever you run a class with a main() method your javaagent is configured for you.

In the example above there is the “debug=1” parameter and with this you will see in the console (which shows the agent is being used) followed by enhancement debug output.

```
premain loading Transformer args:debug=1
```

15.5.5 Eclipse IDE Enhancer Plugin

The Eclipse Enhancer plugin enhances classes as they are saved by the Eclipse IDE.

The benefit of using this approach is that you do not need to configure a javaagent nor do you need to remember to run an ANT task before running your application. That is, you can just code your entity beans and then run your code or junit tests.

Install the plugin

Eclipse 3.4:

Help – Software Updates – Add Site - http://www.avaje.org/eclipseupdate/

Manage Sites – Check the newly added site – OK

Check the http://www.avaje.org/eclipseupdate/ site and follow the instructions.

Note: the trailing slash after eclipseupdate/ is required.

Using the Enhancer Plugin

You need to enable the Enhancer on a per project basis. To do this select a project – right mouse click menu – Toggle Enable/Disable Ebean Enhancement.

When you enable the Enhancer it adds a “Builder” to the list of “builders” ...

Select a project – right mouse click menu – properties – builders

When the enhancement is enabled you see 2 builders both enabled. The Java Builder and the Ebean Builder.

Preferences

There is a Ebean preference page added where you can set debug levels on the Enhancement process and the plugin itself.

How to tell if the Enhancement is occuring?

If the Plugin has initialised then project – right mouse click menu ... will show the menu item to be “Disable Ebean Enhancer” indicating that the enhancer is enabled on this project. If the Plugin has not yet been initialised then the menu item just shows “Toggle Ebean Enhancer” which will turn it on or off.

Turn on debugging and check the log files (Window – Preferences – Ebean)

The other option is to look at the logs that Ebean writes during its startup phase. Specifically it reports the number of entity beans that have been “enhanced” and the number that have been “subclassed” and will log a warning if you are using a mix of both.

In the following example log you can see that ***34*** classes where enhanced and ***3*** were

subclassed. It is recommended not to mix enhancement and subclassing though so when this mixing is detected a warning is displayed.

```
...
INFO: Validation: [on] autocreate.notnull=[true] ...
INFO: Deployment xml [orm.xml]  loaded.
INFO: Entities enhanced[34] subclassed[3]
WARNING: Mixing enhanced and subclassed entities. Subclassed ...
INFO: Transaction logs in: logs
INFO: Autofetch deserialized from file [...\ebean.mysql.autofetch]
...
```

Specialised Tomcat Loader for per-webapp Enhancement

This is on the future development TODO list...

16 Groovy

You can use Ebean with Groovy classes. There is nothing special you need to do. Just annotate your groovy beans with the JPA annotations.

```
//GROOVY CODE (generates the getters and setters etc)
package test

import javax.persistence.*;

@Entity
@Table(name="f_forum")
public class PersonG{

    @Id
    Integer id

    @Column(name="title")
    String name

    @OneToMany(cascade=CascadeType.ALL)
    List<Topic> topics;
}
```

You can use Ebean just as you would in Java.

```
// GROOVY CODE
package test

import com.avaje.ebean.*

public class MainEbean{

    public static void main(String[] args) {

          PersonG g = Ebean.getReference(PersonG.class, 1);
          String name = g.getName();

          List<PersonG> list = Ebean
                .find(PersonG.class)
                .fetch("topics")
                .findList()
```

```
            println "Got list "+list

            list.each() {
                  print " ${it.id} ${it.name} \n"
                  print " GOT DETAILS: "+it.topics
            };
            println "done";
      }
}
```

Note that if you want more groovy integration please make some suggestions of what you would like to see.

17 Scala

You can use Ebean with Scala as well. Again, annotate your scala “bean” with the JPA annotations as you would normally.

Ebean 2.6 has added support for Scala 2.8 mutable Buffer, Set and Map and Option types (you no longer have to use the Java collection types).

Please contact the Ebean google group for the latest Scala examples.

18 Appendix

18.1 BeanFinder BeanPersistController and BeanPersistListener

TODO

18.2 Object Identity – equals() & hashCode()

TODO

18.3 Optimistic Concurrency Checking

TODO

18.4 Cascading Save & Delete

TODO

18.5 JDBC Batching

TODO

18.6 JMX

18.7 Getting Metadata at Runtime

18.7.1 MetaQueryStatistic

TODO

18.7.2 MetaAutofetchStatistic

TODO

... coming MetaBean, MetaTable

18.8 Validation

TODO