Running the Persistence Examples

This chapter explains how to use the Java Persistence API.

The material here focuses on the source code and settings of three examples.

The first example, `order`, is an application that uses a stateful session bean to manage entities related to an ordering system.

The second example, `roster`, is an application that manages a community sports system.

The third example, `address-book`, is a web application that stores contact data.

This chapter assumes that you are familiar with the concepts detailed in Chapter 32, Introduction to the Java Persistence API.

The following topics are addressed here:

- The `order` Application
- The `roster` Application
- The `address-book` Application

The `order` Application

The `order` application is a simple inventory and ordering application for maintaining a catalog of parts and placing an itemized order of those parts.

The application has entities that represent parts, vendors, orders, and line items.

These entities are accessed using a stateful session bean that holds the business logic of the application.

A simple singleton session bean creates the initial entities on application deployment.

A Facelets web application manipulates the data and displays data from the catalog.

The information contained in an order can be divided into elements.

What is the order number?

What parts are included in the order?

What parts make up that part?

Who makes the part?

What are the specifications for the part?

Are there any schematics for the part?

The `order` application is a simplified version of an ordering system that has all these elements.

The `order` application consists of a single WAR module that includes the enterprise bean classes, the entities, the support classes, and the Facelets XHTML and class files.

Entity Relationships in the `order` Application

The `order` application demonstrates several types of entity relationships: self-referential, one-to-one, one-to-many, many-to-one, and unidirectional relationships.

Self-Referential Relationships

A self-referential relationship occurs between relationship fields in the same entity.

`Part` has a field, `bomPart`, which has a one-to-many relationship with the field `parts`, which is also in `Part`.

That is, a part can be made up of many parts, and each of those parts has exactly one bill-of-material part.

The primary key for `Part` is a compound primary key, a combination of the `partNumber` and `revision` fields.

This key is mapped to the PARTNUMBER and REVISION columns in the EJB_ORDER_PART table:

```
...
@ManyToOne
@JoinColumns({
@JoinColumn(name="BOMPARTNUMBER",
referencedColumnName="PARTNUMBER"),
@JoinColumn(name="BOMREVISION",
referencedColumnName="REVISION") })
public Part getBomPart()
```

```
{ return bomPart; } ...
@OneToMany(mappedBy="bomPart")
public Collection<Part> getParts()
{ return parts; }
...
```

One-to-One Relationships

`Part` has a field, `vendorPart`, that has a one-to-one relationship with `VendorPart`'s `part` field.

That is, each part has exactly one vendor part, and vice versa.

Here is the relationship mapping in `Part`:

```
@OneToOne(mappedBy="part")
public VendorPart getVendorPart()
{ return vendorPart; }
```

Here is the relationship mapping in `VendorPart`:

```
@OneToOne
```

```
@JoinColumns({
@JoinColumn(name="PARTNUMBER",
referencedColumnName="PARTNUMBER"),
@JoinColumn(name="PARTREVISION",
referencedColumnName="REVISION")
})
public Part getPart()
{ return part; }
```

Note that, because `Part` uses a compound primary key, the `@JoinColumns` annotation is used to map the columns in the `PERSISTENCE_ORDER_VENDOR_PART` table to the columns in `PERSISTENCE_ORDER_PART`.

The `PERSISTENCE_ORDER_VENDOR_PART` table's `PARTREVISION` column refers to `PERSISTENCE_ORDER_PART`'s `REVISION` column.

One-to-Many Relationship Mapped to Overlapping Primary and Foreign Keys

`Order` has a field, `lineItems`, that has a one-to-many relationship with `LineItem`'s field `order`.

That is, each order has one or more line item.

`LineItem` uses a compound primary key that is made up of the `orderId` and `itemId` fields.

This compound primary key maps to the `ORDERID` and `ITEMID` columns in the `PERSISTENCE_ORDER_LINEITEM` table.

`ORDERID` is a foreign key to the `ORDERID` column in the `PERSISTENCE_ORDER_ORDER` table.

This means that the ORDERID column is mapped twice: once as a primary key field, orderId; and again as a relationship field, order.

Here's the relationship mapping in Order:

```
@OneToMany
(cascade=ALL, mappedBy="order")
public Collection<LineItem>
getLineItems() {return lineItems;}
```

Here is the relationship mapping in `LineItem`:

```
@ManyToOne
public Order getOrder()
{ return order; }
```

Unidirectional Relationships

`LineItem` has a field, `vendorPart`, that has a unidirectional many-to-one relationship with `VendorPart`.

That is, there is no field in the target entity in this relationship:

```
@ManyToOne
public VendorPart getVendorPart()
{ return vendorPart; }
```

Primary Keys in the `order` Application

The `order` application uses several types of primary keys: single-valued primary keys, compound primary keys, and generated primary keys.

Generated Primary Keys

`VendorPart` uses a generated primary key value.

That is, the application does not assign primary key values for the entities but instead relies on the persistence provider to generate the primary key values.

The `@GeneratedValue` annotation is used to specify that an entity will use a generated primary key.

In `VendorPart`, the following code specifies the settings for generating primary key values:

```
@TableGenerator(
name="vendorPartGen",
```

```
table=
"PERSISTENCE_ORDER_SEQUENCE_GENERATOR",
pkColumnName="GEN_KEY",
valueColumnName="GEN_VALUE",
pkColumnValue="VENDOR_PART_ID",
allocationSize=10)
@Id
@GeneratedValue(
strategy=GenerationType.TABLE,
generator="vendorPartGen")
public Long getVendorPartNumber()
```

```
{  return vendorPartNumber;  }
```

The `@TableGenerator` annotation is used in conjunction with `@GeneratedValue`'s `strategy=TABLE` element.

That is, the strategy used to generate the primary keys is to use a table in the database.

The `@TableGenerator` annotation is used to configure the settings for the generator table.

The name element sets the name of the generator, which is `vendorPartGen` in `VendorPart`.

The `EJB_ORDER_SEQUENCE_GENERATOR` table, whose two columns are `GEN_KEY` and `GEN_VALUE`, will store the generated primary key values.

This table could be used to generate other entity's primary keys, so the `pkColumnValue` element is set to `VENDOR_PART_ID` to distinguish this entity's generated primary keys from other entity's generated primary keys.

The `allocationSize` element specifies the amount to increment when allocating primary key values.

In this case, each `VendorPart`'s primary key will increment by 10.

The primary key field `vendorPartNumber` is of type `Long`, as the generated primary key's field must be an integral type.

Compound Primary Keys

A compound primary key is made up of multiple fields and follows the requirements described in Primary Keys in Entities.

To use a compound primary key, you must create a wrapper class.

In `order`, two entities use compound primary keys: `Part` and `LineItem`.

- `Part` uses the `PartKey` wrapper class.

  `Part`'s primary key is a combination of the part number and the revision number.

  `PartKey` encapsulates this primary key.

- `LineItem` uses the `LineItemKey` class.

  `LineItem`'s primary key is a combination of the order number and the item number.

  `LineItemKey` encapsulates this primary key.

This is the `LineItemKey` compound primary key wrapper class:

```
package order.entity;
public final class LineItemKey
implements java.io.Serializable {
private Integer orderId;
private int itemId;
public int hashCode() {
return ((this.getOrderId()==null
?0:this.getOrderId().hashCode())
^ ((int) this.getItemId()));
}
```

```
public boolean equals
(Object otherOb) {
if (this == otherOb){return true;}
if(!(otherOb instanceof LineItemKey))
{ return false; }
LineItemKey other =
(LineItemKey) otherOb;
return ((this.getOrderId()==null
?other.orderId==null:this.getOrderI
d().equals(other.orderId)) &&
(this.getItemId == other.itemId));}
```

```
public String toString()
{return "" + orderId+ "-"+itemId;}
}
```

The `@IdClass` annotation is used to specify the primary key class in the entity class.

In `LineItem`, `@IdClass` is used as follows:

```
@IdClass
(order.entity.LineItemKey.class)
@Entity...
public class LineItem {...}
```

The two fields in `LineItem` are tagged with the `@Id` annotation to mark those fields as part of the compound primary key:

```
@Id
public int getItemId()
{ return itemId; } ...
@Id
@Column(name="ORDERID",
nullable=false,
insertable=false, updatable=false)
public Integer getOrderId()
{ return orderId; }
```

For `orderId`, you also use the `@Column` annotation to specify the column name in the table and that this column should not be inserted or updated, as it is an overlapping foreign key pointing at the `PERSISTENCE_ORDER_ORDER` table's `ORDERID` column (see One-to-Many Relationship Mapped to Overlapping Primary and Foreign Keys).

That is, `orderId` will be set by the `Order` entity.

In `LineItem`'s constructor, the line item number (`LineItem.itemId`) is set using the `Order.getNextId` method:

```
public LineItem(Order order,
int quantity, VendorPart
vendorPart){
```

```
this.order = order;
this.itemId = order.getNextId();
this.orderId = order.getOrderId();
this.quantity = quantity;
this.vendorPart = vendorPart;
}
```

`Order.getNextId` counts the number of current line items, adds 1, and returns that number:

```
public int getNextId()
{return this.lineItems.size() + 1;}
```

`Part` doesn't require the `@Column` annotation on the two fields that comprise `Part`'s compound primary key, because `Part`'s compound primary key is not an overlapping primary key/foreign key:

```
@IdClass
(order.entity.PartKey.class)
@Entity...
public class Part{...
@Id
public String getPartNumber()
{ return partNumber; }...
@Id
public int getRevision()
{ return revision; } ...
}
```

Entity Mapped to More Than One Database Table

`Part`'s fields map to more than one database table: `PERSISTENCE_ORDER_PART` and `PERSISTENCE_ORDER_PART_DETAIL`.

The PERSISTENCE_ORDER_PART_DETAIL table holds the specification and schematics for the part.

The @SecondaryTable annotation is used to specify the secondary table.

```
...
@Entity
@Table
(name="PERSISTENCE_ORDER_PART")
@SecondaryTable(
```

```
name="PERSISTENCE_ORDER_PART_DETAIL",
pkJoinColumns={
@PrimaryKeyJoinColumn
(name="PARTNUMBER",
referencedColumnName="PARTNUMBER"),
@PrimaryKeyJoinColumn
(name="REVISION",
referencedColumnName="REVISION")
})
public class Part {...}
```

PERSISTENCE_ORDER_PART_DETAIL and PERSISTENCE_ORDER_PART share the same primary key values.

The pkJoinColumns element of @SecondaryTable is used to specify that PERSISTENCE_ORDER_PART_DETAIL's primary key columns are foreign keys to PERSISTENCE_ORDER_PART.

The `@PrimaryKeyJoinColumn` annotation sets the primary key column names and specifies which column in the primary table the column refers to.

In this case, the primary key column names for both `PERSISTENCE_ORDER_PART_DETAIL` and `PERSISTENCE_ORDER_PART` are the same: `PARTNUMBER` and `REVISION`, respectively.

Cascade Operations in the `order` Application

Entities that have relationships to other entities often have dependencies on the existence of the other entity in the relationship.

For example, a line item is part of an order; if the order is deleted, then the line item also should be deleted.

This is called a cascade delete relationship.

In `order`, there are two cascade delete dependencies in the entity relationships.

If the `Order` to which a `LineItem` is related is deleted, the `LineItem` also should be deleted.

If the `Vendor` to which a `VendorPart` is related is deleted, the `VendorPart` also should be deleted.

You specify the cascade operations for entity relationships by setting the `cascade` element in the inverse (nonowning) side of the relationship.

The cascade element is set to `ALL` in the case of `Order.lineItems`.

This means that all persistence operations (deletes, updates, and so on) are cascaded from orders to line items.

Here is the relationship mapping in `Order`:

```
@OneToMany
(cascade=ALL, mappedBy="order")
public
Collection<LineItem> getLineItems()
{ return lineItems; }
```

Here is the relationship mapping in `LineItem`:

```
@ManyToOne
public Order getOrder()
{ return order; }
```

BLOB and CLOB Database Types in the `order` Application

The `PARTDETAIL` table in the database has a column, `DRAWING`, of type `BLOB`.

`BLOB` stands for binary large objects, which are used for storing binary data, such as an image.

The `DRAWING` column is mapped to the field `Part`.

`drawing` of type `java.io.Serializable`.

The `@Lob` annotation is used to denote that the field is large object.

```
@Column
(table="PERSISTENCE_ORDER_PART_DETAIL")
```

```
@Lob
public Serializable getDrawing()
{  return drawing;  }
```

PERSISTENCE_ORDER_PART_DETAIL also has a column, SPECIFICATION, of type CLOB.

CLOB stands for character large objects, which are used to store string data too large to be stored in a VARCHAR column.

SPECIFICATION is mapped to the field Part.specification of type java.lang.String.

The @Lob annotation is also used here to denote that the field is a large object.

```
@Column
(table="PERSISTENCE_ORDER_PART_DETAIL")
```

```
@Lob
public String getSpecification()
{  return specification;  }
```

Both of these fields use the `@Column` annotation and set the `table` element to the secondary table.

Temporal Types in the `order` Application

The `Order.lastUpdate` persistent property, which is of type `java.util.Date`, is mapped to the `PERSISTENCE_ORDER_ORDER.LASTUPDATE` database field, which is of the SQL type `TIMESTAMP`.

To ensure the proper mapping between these types, you must use the `@Temporal` annotation with the proper temporal type specified in `@Temporal`'s element.

`@Temporal`'s elements are of type `javax.persistence.TemporalType`.

The possible values are

- `DATE`, which maps to `java.sql.Date`
- `TIME`, which maps to `java.sql.Time`
- `TIMESTAMP`, which maps to `java.sql.Timestamp`

Here is the relevant section of `Order`:

```
@Temporal(TIMESTAMP)
public Date getLastUpdate()
{ return lastUpdate; }
```

Managing the `order` Application's Entities

The `RequestBean` stateful session bean contains the business logic and manages the entities of `order`.

RequestBean uses the @PersistenceContext annotation to retrieve an entity manager instance, which is used to manage order's entities in RequestBean's business methods:

```
@PersistenceContext
private EntityManager em;
```

This `EntityManager` instance is

a container-managed entity manager, so the container takes care of all the transactions involved in the managing `order`'s entities.

Creating Entities

The `RequestBean.createPart` business method creates a new `Part` entity.

The `EntityManager.persist` method is used to persist the newly created entity to the database.

```
Part part = new Part(
partNumber, revision,
description, revisionDate,
specification, drawing);
em.persist(part);
```

The `ConfigBean` singleton session bean is used to initialize the data in `order`.

`ConfigBean` is annotated with `@Startup`, which indicates that the EJB container should create `ConfigBean` when `order` is deployed.

The `createData` method is annotated with `@PostConstruct` and creates the initial entities used by `order` by calling `RequestsBean`'s business methods.

Finding Entities

The `RequestBean.getOrderPrice` business method returns the price of a given order, based on the `orderId`.

The `EntityManager.find` method is used to retrieve the entity from the database.

```
Order order =
em.find(Order.class, orderId);
```

The first argument of `EntityManager.find` is the entity class, and the second is the primary key.

Setting Entity Relationships

The `RequestBean.createVendorPart` business method creates a `VendorPart` associated with a particular `Vendor`.

The `EntityManager.persist` method is used to persist the newly created `VendorPart` entity to the database, and the `VendorPart.setVendor` and `Vendor.setVendorPart` methods are used to associate the `VendorPart` with the `Vendor`.

```
PartKey pkey = new PartKey();
pkey.partNumber = partNumber;
pkey.revision = revision;
```

```
Part part =
em.find(Part.class, pkey);
VendorPart vendorPart =
new VendorPart
(description, price, part);
em.persist(vendorPart);
Vendor vendor =
em.find(Vendor.class, vendorId);
vendor.addVendorPart(vendorPart);
vendorPart.setVendor(vendor);
```

Using Queries

The `RequestBean.adjustOrderDiscount` business method updates the discount applied to all orders.

This method uses the `findAllOrders` named query, defined in `Order`:

```
@NamedQuery(
name="findAllOrders",
query="SELECT o FROM Order o"
)
```

The `EntityManager.createNamedQuery` method is used to run the query.

Because the query returns a `List` of all the orders, the `Query.getResultList` method is used.

```
List orders = em.createNamedQuery
("findAllOrders").getResultList();
```

The `RequestBean.getTotalPricePerVendor` business method returns the total price of all the parts for a particular vendor.

This method uses a named parameter, `id`, defined in the named query `findTotalVendorPartPricePerVendor` defined in `VendorPart`.

```
@NamedQuery(name=
"findTotalVendorPartPricePerVendor"
, query="SELECT SUM(vp.price) " +
"FROM VendorPart vp " +
"WHERE vp.vendor.vendorId = :id"
)
```

When running the query, the `Query.setParameter` method is used to set the named parameter `id` to the value of `vendorId`, the parameter to `RequestBean.getTotalPricePerVendor`:

```
return (Double) em.createNamedQuery
("findTotalVendorPartPricePerVendor")
.setParameter("id", vendorId)
.getSingleResult();
```

The `Query.getSingleResult` method is used for this query because the query returns a single value.

Removing Entities

The `RequestBean.removeOrder` business method deletes a given order from the database.

This method uses the `EntityManager.remove` method to delete the entity from the database.

```
Order order =
em.find(Order.class, orderId);
em.remove(order);
```

Building, Packaging, Deploying, and Running the `order` Application

This section explains how to build, package, deploy, and run the `order` application.

To do this, you will create the database tables in the Java DB server, then build, deploy, and run the example.

To Build, Package, Deploy, and Run `order` Using NetBeans IDE

1. From the File menu, choose Open Project.

2. In the Open Project dialog, navigate to:
   `tut-install/examples/persistence/`

3. Select the `order` folder.

4. Select the Open as Main Project check box.

5. Click Open Project.

6. In the Projects tab, right-click the `order` project and select Run.

NetBeans IDE opens a web browser to `http://localhost:8080/order/`.

To Build, Package, Deploy, and Run `order` *Using Ant*

1. In a terminal window, go to:

   `tut-install/examples/persistence/order/`

2. Type the following command:

   `ant`

This runs the `default` task, which compiles the source files and packages the application into a WAR file located at *tut-install*`/examples/persistence/order/dist/order.war`.

3. To deploy the WAR, make sure that the GlassFish Server is started, then type the following command:

```
ant deploy
```

4. Open a web browser to `http://localhost:8080/order/` to create and update the order data.

The `all` Task

As a convenience, the `all` task will build, package, deploy, and run the application.

To do this, type the following command:

```
ant all
```

The `roster` Application

The `roster` application maintains the team rosters for players in recreational sports leagues.

The application has four components: Java Persistence API entities (`Player`, `Team`, and `League`), a stateful session bean (`RequestBean`),

an application client (`RosterClient`), and three helper classes (`PlayerDetails`, `TeamDetails`, and `LeagueDetails`).

Functionally, `roster` is similar to the `order` application, with three new features that `order` does not have: many-to-many relationships, entity inheritance, and automatic table creation at deployment time.

Relationships in the `roster` Application

A recreational sports system has the following relationships:

- A player can be on many teams.
- A team can have many players.
- A team is in exactly one league.
- A league has many teams.

In `roster` this system is reflected by the following relationships between the `Player`, `Team`, and `League` entities.

- There is a many-to-many relationship between `Player` and `Team`.

- There is a many-to-one relationship between `Team` and `League`.

The Many-To-Many Relationship in `roster`

The many-to-many relationship between `Player` and `Team` is specified by using the `@ManyToMany` annotation.

In `Team.java`, the `@ManyToMany` annotation decorates the `getPlayers` method:

```
@ManyToMany
@JoinTable(
name="EJB_ROSTER_TEAM_PLAYER",
joinColumns=
@JoinColumn(name="TEAM_ID",
referencedColumnName="ID"),
inverseJoinColumns=
@JoinColumn(name="PLAYER_ID",
referencedColumnName="ID")
)
```

```
public Collection<Player>
getPlayers() { return players; }
```

The @JoinTable annotation is used to specify a database table that will associate player IDs with team IDs.

The entity that specifies the `@JoinTable` is the owner of the relationship, so the `Team` entity is the owner of the relationship with the `Player` entity.

Because `roster` uses automatic table creation at deployment time, the container will create a join table named `EJB_ROSTER_TEAM_PLAYER`.

`Player` is the inverse, or nonowning, side of the relationship with `Team`.

As one-to-one and many-to-one relationships, the nonowning side is marked by the `mappedBy` element in the relationship annotation.

Because the relationship between `Player` and `Team` is bidirectional, the choice of which entity is the owner of the relationship is arbitrary.

In `Player.java`, the `@ManyToMany` annotation decorates the `getTeams` method:

```
@ManyToMany(mappedBy="players")
public Collection<Team> getTeams()
{ return teams; }
```

Entity Inheritance in the `roster` Application

The `roster` application shows how to use entity inheritance, as described in Entity Inheritance.

The `League` entity in `roster` is an abstract entity with two concrete subclasses: `SummerLeague` and `WinterLeague`.

Because League is an abstract class, it cannot be instantiated:

```
...
@Entity
@Table(name = "EJB_ROSTER_LEAGUE")
public abstract class League
implements java.io.Serializable
{ ... }
```

Instead, when creating a league, clients use `SummerLeague` or `WinterLeague`.

`SummerLeague` and `WinterLeague` inherit the persistent properties defined in `League` and add only a constructor that verifies that the sport parameter matches the type of sport allowed in that seasonal league.

For example, here is the `SummerLeague` entity:

```
...
@Entity
public class SummerLeague extends
League
implements java.io.Serializable {
/** Creates a new instance
 * of SummerLeague
 */
public SummerLeague() { }
```

```
public SummerLeague(String id,
String name, String sport)
throws IncorrectSportException{
this.id = id;
this.name = name;
if
(sport.equalsIgnoreCase("swimming")
|| sport.equalsIgnoreCase("soccer")
|| sport.equalsIgnoreCase
("basketball")
```

```
|| sport.equalsIgnoreCase
("baseball"))
{ this.sport = sport; }
else {
throw new IncorrectSportException
("Sport is not a summer sport.");
} }
}
```

The `roster` application uses the default mapping strategy of `InheritanceType.SINGLE_TABLE`, so the `@Inheritance` annotation is not required.

If you want to use a different mapping strategy, decorate `League` with `@Inheritance` and specify the mapping strategy in the `strategy` element:

```
@Entity
@Inheritance(strategy=JOINED)
@Table(name="EJB_ROSTER_LEAGUE")
public abstract class League
implements java.io.Serializable
{ ... }
```

The `roster` application uses the default discriminator column name, so the `@DiscriminatorColumn` annotation is not required.

Because you are using automatic table generation in `roster`, the Persistence provider will create a discriminator column called `DTYPE` in the `EJB_ROSTER_LEAGUE` table, which will store the name of the inherited entity used to create the league.

If you want to use a different name for the discriminator column, decorate `League` with `@DiscriminatorColumn` and set the `name` element:

```
@Entity
@DiscriminatorColumn
(name="DISCRIMINATOR")
@Table(name="EJB_ROSTER_LEAGUE")
public abstract class League
implements java.io.Serializable
{ ... }
```

Criteria Queries in the `roster` Application

The `roster` application uses Criteria API queries, as opposed to the JPQL queries used in `order`.

Criteria queries are Java programming language, typesafe queries defined in the business tier of `roster`, in the `RequestBean` stateless session bean.

Metamodel Classes in the `roster` Application

Metamodel classes model an entity's attributes and are used by Criteria queries to navigate to an entity's attributes.

Each entity class in `roster` has a corresponding metamodel class, generated at compile time, with the same package name as the entity and appended with an underscore character (_).

For example, the `roster.entity.Person` entity has a corresponding metamodel class, `roster.entity.Person_`.

Each persistent field or property in the entity class has a corresponding attribute in the entity's metamodel class.

For the `Person` entity, the corresponding metamodel class is:

```
@StaticMetamodel(Person.class)
public class Person_ {
public static volatile
SingularAttribute<Player, String>
id;
public static volatile
SingularAttribute<Player, String>
name;
public static volatile
SingularAttribute<Player, String>
position;
```

```
public static volatile
SingularAttribute<Player, Double>
salary;
public static volatile
CollectionAttribute<Player, Team>
teams;
}
```

Obtaining a `CriteriaBuilder` Instance in `RequestBean`

The `CrtiteriaBuilder` interface defines methods to create criteria query objects and create expressions for modifying those query objects.

RequestBean creates an instance of CriteriaBuilder by using a @PostConstruct method, init:

```
@PersistenceContext
private EntityManager em;
private CriteriaBuilder cb;
@PostConstruct
private void init()
{ cb = em.getCriteriaBuilder(); }
```

The `EntityManager` instance is injected at runtime, and then that `EntityManager` object is used to create the `CriteriaBuilder` instance by calling `getCriteriaBuilder`.

The `CriteriaBuilder` instance is created in a `@PostConstruct` method to ensure that the `EntityManager` instance has been injected by the enterprise bean container.

Creating Criteria Queries in `RequestBean`'s Business Methods

Many of the business methods in `RequestBean` define Criteria queries.

One business method, `getPlayersByPosition`, returns a list of players who play a particular position on a team:

```
public List<PlayerDetails>
getPlayersByPosition
(String position){
logger.info("getPlayersByPosition");
List<Player> players = null;
try {
CriteriaQuery<Player> cq =
cb.createQuery(Player.class);
if (cq != null) {
Root<Player> player =
cq.from(Player.class);
```

```
// set the where clause
cq.where(cb.equal
(player.get(Player_.position),position));
cq.select(player);
TypedQuery<Player> q =
em.createQuery(cq);
players = q.getResultList();
}
return
copyPlayersToDetails(players);
} catch (Exception ex)
```

```
{ throw new EJBException(ex); }
}
```

A query object is created by calling the `CriteriaBuilder` object's `createQuery` method, with the type set to `Player` because the query will return a list of players.

The query root, the base entity from which the query will navigate to find the entity's attributes and related entities, is created by calling the `from` method of the query object.

This sets the FROM clause of the query.

The WHERE clause, set by calling the `where` method on the query object, restricts the results of the query according to the conditions of an expression.

The `CriteriaBuilder.equal` method compares the two expressions.

In `getPlayersByPosition`, the `position` attribute of the `Player_` metamodel class, accessed by calling the `get` method of the query root, is compared to the `position` parameter passed to `getPlayersByPosition`.

The SELECT clause of the query is set by calling the `select` method of the query object.

The query will return `Player` entities, so the query root object is passed as a parameter to `select`.

The query object is prepared for execution by calling `EntityManager.createQuery`, which returns a `TypedQuery<T>` object with the type of the query, in this case `Player`.

This typed query object is used to execute the query, which occurs when the `getResultList` method is called, and a `List<Player>` collection is returned.

Automatic Table Generation in the `roster` Application

At deployment time, the GlassFish Server will automatically drop and create the database tables used by `roster`.

This is done by setting

the `eclipselink.ddl-generation` property to `drop-and-create-tables` in `persistence.xml`:

```
<?xml version="1.0"
encoding="UTF-8"?>
<persistence version="2.0"
xmlns=
"http://java.sun.com/xml/ns/persistence"
```

```
xmlns:xsi="http://www.w3.org/2001/
XMLSchema-instance"
xsi:schemaLocation="http://
java.sun.com/xml/ns/persistence
http://java.sun.com/xml/ns/
persistence/persistence_2_0.xsd">
<persistence-unit name="em"
transaction-type="JTA">
<jta-data-source>
jdbc/__default
</jta-data-source>
```

```
<properties>
<property
name="eclipselink.ddl-generation"
value="drop-and-create-tables"/>
</properties>
</persistence-unit>
</persistence>
```

This feature is specific to the Java Persistence API provider used by the GlassFish Server and is nonportable across Java EE servers.

Automatic table creation is useful for development purposes, however, and the `eclipselink.ddl-generation` property may be removed from `persistence.xml` when preparing the application for production use or when deploying to other Java EE servers.

Building, Packaging, Deploying, and Running the `roster` Application

This section explains how to build, package, deploy, and run the `roster` application.

You can do this using either NetBeans IDE or Ant.

To Build, Package, Deploy, and Run `roster` Using NetBeans IDE

1. From the File menu, choose Open Project.

2. In the Open Project dialog, navigate to:
   `tut-install/examples/persistence/`

3. Select the `roster` folder.

4. Select the Open as Main Project and Open Required Projects check boxes.

5. Click Open Project.

6. In the Projects tab, right-click the `roster` project and select Run.

You will see the following partial output from the application client in the Output tab:

```
List all players in team T2:
P6 Ian Carlyle goalkeeper 555.0
P7 Rebecca Struthers midfielder
777.0
P8 Anne Anderson forward 65.0
P9 Jan Wesley defender 100.0
P10 Terry Smithson midfielder
100.0
```

```
List all teams in league L1:
T1 Honey Bees Visalia
T2 Gophers Manteca
T5 Crows Orland
List all defenders:
P2 Alice Smith defender 505.0
P5 Barney Bold defender 100.0
P9 Jan Wesley defender 100.0
P22 Janice Walker defender 857.0
P25 Frank Fletcher defender 399.0
...
```

To Build, Package, Deploy, and Run `roster` *Using Ant*

1. In a terminal window, go to:

   `tut-install/examples/persistence/roster/`

2. Type the following command:

   `ant`

This runs the `default` task, which compiles the source files and packages the application into an EAR file located at

tut-install`/examples/persistence/roster/dist/roster.ear`.

3. To deploy the EAR, make sure that the GlassFish Server is started; then type the following command:

`ant deploy`

The build system will check whether the Java DB database server is running and start it if it is not running, then deploy `roster.ear`.

The GlassFish Server will then drop and create the database tables during deployment, as specified in `persistence.xml`.

After `roster.ear` is deployed, a client JAR, `rosterClient.jar`, is retrieved.

This contains the application client.

4. To run the application client, type the following command:

```
ant run
```

You will see the output, which begins:

```
[echo] running application client container.
[exec] List all players in team T2:
[exec] P6 Ian Carlyle goalkeeper 555.0
[exec] P7 Rebecca Struthers midfielder 777.0
[exec] P8 Anne Anderson forward 65.0
[exec] P9 Jan Wesley defender 100.0
[exec] P10 Terry Smithson midfielder 100.0
[exec] List all teams in league L1:
[exec] T1 Honey Bees Visalia
[exec] T2 Gophers Manteca
[exec] T5 Crows Orland
```

```
[exec] List all defenders:
[exec] P2 Alice Smith defender 505.0
[exec] P5 Barney Bold defender 100.0
[exec] P9 Jan Wesley defender 100.0
[exec] P22 Janice Walker defender 857.0
[exec] P25 Frank Fletcher defender 399.0
...
```

The `all` Task

As a convenience, the `all` task will build, package, deploy, and run the application.

To do this, type the following command:

```
ant all
```

The `address-book` Application

The `address-book` example application is a simple web application that stores contact data.

It uses a single entity class, `Contact`, that uses the Java API for JavaBeans Validation (Bean Validation) to validate the data stored in the persistent attributes of the entity, as described in Validating Persistent Fields and Properties.

Bean Validation Constraints in `address-book`

The `Contact` entity uses the `@NotNull`, `@Pattern`, and `@Past` constraints on the persistent attributes.

The `@NotNull` constraint marks the attribute as a required field.

The attribute must be set to a non-null value before the entity can be persisted or modified.

Bean Validation will throw a validation error if the attribute is null when the entity is persisted or modified.

The `@Pattern` constraint defines a regular expression that the value of the attribute must match before the entity can be persisted or modified.

This constraint has two different uses in `address-book`.

- The regular expression declared in the `@Pattern` annotation on the `email` field matches email addresses of the form *`name@domain name.top level domain`*, allowing only valid characters for email addresses.

For example, `username@example.com` will pass validation, as will `firstname.lastname@mail.example.com`.

However, `firstname,lastname@example.com`, which contains an illegal comma character in the local name, will fail validation.

- The `mobilePhone` and `homePhone` fields are annotated with a `@Pattern` constraint that defines a regular expression to match phone numbers of the form

`(xxx) xxx-xxxx`.

The `@Past` constraint is applied to the birthday field, which must be a `java.util.Date` in the past.

Here are the relevant parts of the `Contact` entity class:

```
@Entity
public class Contact implements
Serializable {
private static final
long serialVersionUID = 1L;
@Id
@GeneratedValue
(strategy = GenerationType.AUTO)
```

```
private Long id;
@NotNull
protected String firstName;
@NotNull
protected String lastName;
@Pattern(regexp="
[a-z0-9!#$%&'*+/=?^_`{|}~-]+(?:\\."
+"[a-z0-9!#$%&'*+/=?^_`{|}~-]+)*"
+"@(?:[a-z0-9](?:[a-z0-9-]*
[a-z0-9])?\\.)+[a-z0-9]
(?:[a-z0-9-]*[a-z0-9])?",
```

```
message="{invalid.email}")
protected String email;
@Pattern(regexp="^\\(?(\\d{3})\\)?
[- ]?(\\d{3})[- ]?(\\d{4})$",
message="{invalid.phonenumber}")
protected String mobilePhone;
@Pattern(regexp="^\\(?(\\d{3})\\)?
[- ]?(\\d{3})[- ]?(\\d{4})$",
message="{invalid.phonenumber}")
protected String homePhone;
```

```
@Temporal
(javax.persistence.TemporalType.DATE)
@Past
protected Date birthday;
...
}
```

Specifying Error Messages for Constraints in `address-book`

Some of the constraints in the `Contact` entity specify an optional message:

```
@Pattern(regexp="^\\(?(\\d{3})\\)?
[- ]?(\\d{3})[- ]?(\\d{4})$",
message="{invalid.phonenumber}")
protected String homePhone;
```

The optional message element in the @Pattern constraint overrides the default validation message.

The message can be specified directly:

```
@Pattern(regexp="^\\(?(\\d{3})\\)?
[- ]?(\\d{3})[- ]?(\\d{4})$",
message="Invalid phone number!")
protected String homePhone;
```

The constraints in `Contact`, however, are strings in the resource bundle

tut-install`/examples/persistence/address-book/src/java/ValidationMessages.properties`.

This allows the validation messages to be located in one single properties file and the messages to be easily localized.

Overridden Bean Validation messages must be placed in a resource bundle properties file named `ValidationMessages.properties` in the default package, with localized resource bundles taking the form `ValidationMessages_`*locale-prefix*`.properties`.

For example, `ValidationMessages_es.properties` is the resource bundle used in Spanish speaking locales.

Validating `Contact` Input from a JavaServer Faces Application

The `address-book` application uses a JavaServer Faces web front end to allow users to enter contacts.

While JavaServer Faces has a form input validation mechanism using tags in Facelets XHTML files, `address-book` doesn't use these validation tags.

Bean Validation constraints in JavaServer Faces backing beans, in this case in the `Contact` entity, automatically trigger validation when the forms are submitted.

The following code snippet from the Create.xhtml Facelets file shows some of the input form for creating new Contact instances:

```
<h:form>
<h:panelGrid columns="3">
<h:outputLabel value=
"#{bundle.CreateContactLabel_firstName}"
for="firstName" />
```

```
<h:inputText id="firstName"
value=
"#{contactController.selected.firstName}"
title=
"#{bundle.CreateContactTitle_firstName}"
/>
<h:message for="firstName"
errorStyle="color: red"
infoStyle="color: green" />
```

```
<h:outputLabel value=
"#{bundle.CreateContactLabel_lastName}"
for="lastName" />
<h:inputText id="lastName"
value=
"#{contactController.selected.lastName}"
title=
"#{bundle.CreateContactTitle_lastName}"
/>
```

```
<h:message for="lastName"
errorStyle="color: red"
infoStyle="color: green" />
...
</h:panelGrid>
</h:form>
```

The `<h:inputText>` tags `firstName` and `lastName` are bound to the attributes in the `Contact` entity instance `selected` in the `ContactController` stateless session bean.

Each `<h:inputText>` tag has an associated `<h:message>` tag that will display validation error messages.

The form doesn't require any JavaServer Faces validation tags, however.

Building, Packaging, Deploying, and Running the `address-book` Application

This section describes how to build, package, deploy, and run the `address-book` application.

You can do this using either NetBeans IDE or Ant.

Building, Packaging, Deploying, and Running the `address-book` Application in NetBeans IDE

1. From the File menu, choose Open Project.

2. In the Open Project dialog, navigate to:
   `tut-install/examples/persistence/`

3. Select the `address-book` folder.

4. Select the Open as Main Project and Open Required Projects check boxes.

5. Click Open Project.

6. In the Projects tab, right-click the `address-book` project and select Run.

After the application has been deployed, a web browser window appears at the following URL:

```
http://localhost:8080/
address-book/
```

7. Click Show All Contact Items, then Create New Contact.

Type values in the form fields; then click Save.

If any of the values entered violate the constraints in `Contact`, an error message will appear in red beside the form field with the incorrect values.

Building, Packaging, Deploying, and Running the `address-book` Application Using Ant

1. In a terminal window, go to:

   `tut-install``/examples/persistence/address-book`

2. Type the following command:

   `ant`

This will compile and assemble the `address-book` application.

3. Type the following command:

```
ant deploy
```

This will deploy the application to GlassFish Server.

4. Open a web browser window and type the following URL:

```
http://localhost:8080/
address-book/
```

Tip - As a convenience, the `all` task will build, package, deploy, and run the application.

To do this, type the following command:

```
ant all
```

5. Click Show All Contact Items, then Create New Contact.

Type values in the form fields; then click Save.

If any of the values entered violate the constraints in `Contact`, an error message will appear in red beside the form field with the incorrect values.