The order Application

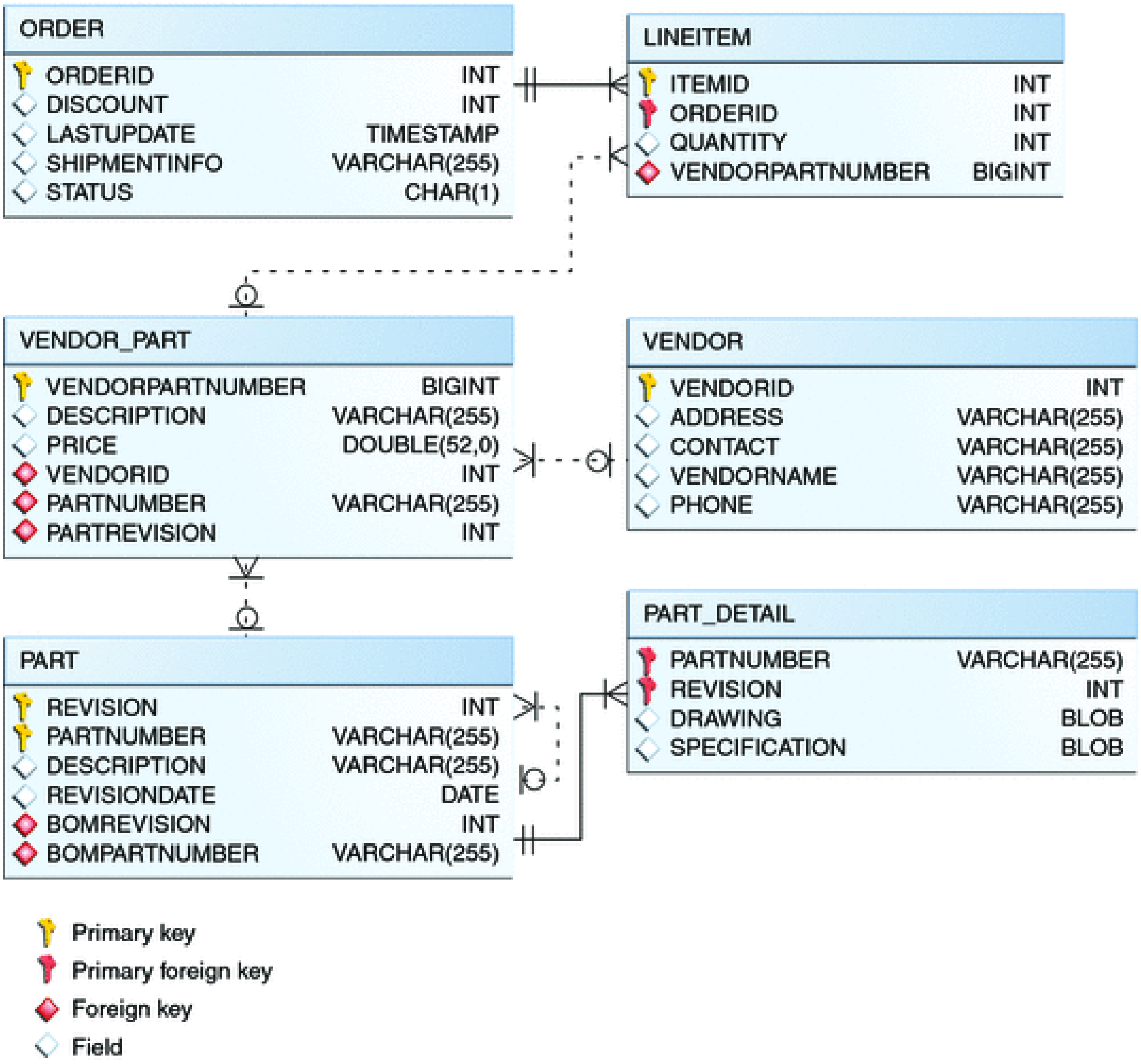
The order application is a simple inventory and ordering application for maintaining a catalog of parts and placing an itemized order of those parts. The application has entities that represent parts, vendors, orders, and line items. These entities are accessed using a stateful session bean that holds 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.

The database schema in the Java DB database for order is shown in Figure 33-1.

Figure 33-1 Database Schema for the order Application



Note - In this diagram, for simplicity, the PERSISTENCE_ORDER_ prefix is omitted from the table names.

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 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;
}
```

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 inPrimary 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.getOrderId().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 Orderentity.

In LineItem's constructor, the line item number (LineItem.itemId) is set using theOrder.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 andPERSISTENCE_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 thatPERSISTENCE_ORDER_PART_DETAIL's primary key columns are foreign keys toPERSISTENCE_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 andPERSISTENCE_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 ofOrder.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 DRAWINGcolumn is mapped to the field Part.drawing of type java.io.Serializable. The @Lobannotation 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. CLOBstands for character large objects, which are used to store string data too large to be stored in aVARCHAR column. SPECIFICATION is mapped to the field Part.specification of typejava.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 thePERSISTENCE_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 `RequestBean`'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);
```

Running the `order` Example

You can use either NetBeans IDE or Ant to build, package, deploy, and run the `order` application. First, you will create the database tables in the Java DB server.

To Run the `order` Example 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 Run the `order` Example 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
```