The NetBeans E-commerce Tutorial - Integrating Transactional Business Logic

Tutorial Contents

The purpose of this tutorial unit is to demonstrate how you can use the object-relational mapping (ORM) capabilities provided by EJB and JPA technologies to gather data from a web request and write to a back-end database. Of particular interest is EJB's support for *container-managed* transactions (refer to the GlassFish v3 Java EE Container diagram). By applying several non-intrusive annotations, you can transform your EJB class into a transaction manager, thereby ensuring the integrity of the data contained in the database. In other words, the transaction manager handles multiple write actions to the database as a single unit of work. It ensures that the work-unit is performed either in its entirety or, if failure occurs at some point during the process, any changes made are rolled back to the database's pre-transaction state.

Within the context of the `AffableBean` application, this tutorial unit focuses on processing a customer order when data from the checkout form is received. You create an `OrderManager` EJB to process the checkout form data along with the session `cart` object. The `OrderManager` performs a transaction that involves multiple write actions to the `affablebean` database. If any of the actions fails, the transaction is rolled back.

You can view a live demo of the application that you build in this tutorial: NetBeans E-commerce Tutorial Demo Application.

Software or Resource	Version Required
NetBeans IDE	Java bundle, 6.8 or 6.9
Java Development Kit (JDK)	version 6
GlassFish server	v3 or Open Source Edition 3.0.1
MySQL database server	version 5.1
AffableBean project	snapshot 7

Notes:

- The NetBeans IDE requires the Java Development Kit (JDK) to run properly. If you do not have any of the resources listed above, the JDK should be the first item that you download and install.
- The NetBeans IDE Java Bundle includes Java Web and EE technologies, which are required for the application you

build in this tutorial.

- The NetBeans IDE Java Bundle also includes the GlassFish server, which you require for this tutorial. You could download the GlassFish server independently, but the version provided with the NetBeans download has the added benefit of being automatically registered with the IDE.
- You can follow this tutorial unit without having completed previous units. To do so, see the setup instructions, which describe how to prepare the database and establish connectivity between the IDE, GlassFish, and MySQL.

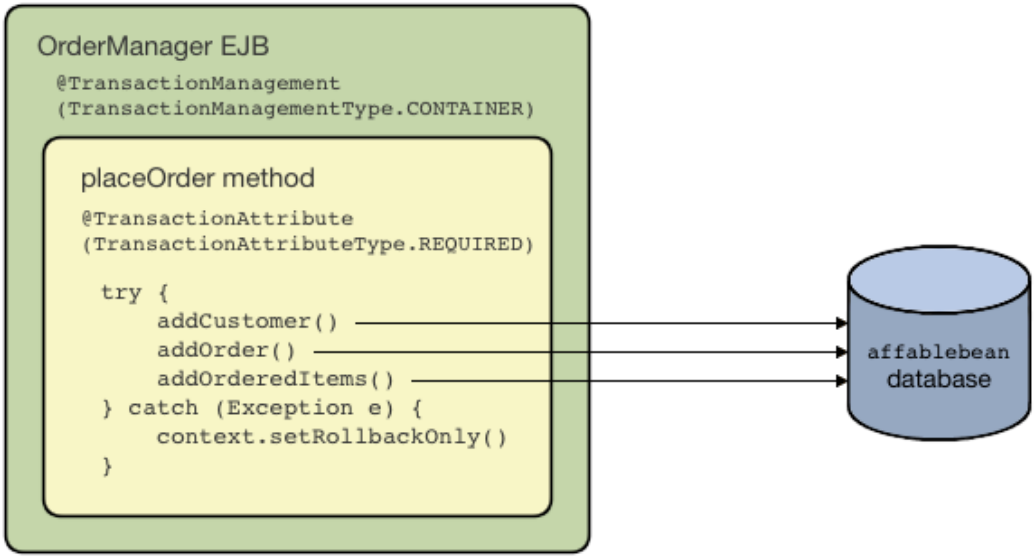
Overview of the Transaction

In order to process the data from the checkout form as well as the items contained in the customer's shopping cart, you create an `OrderManager` EJB. The `OrderManager` uses the provided data and performs the following write actions to the database:

- A new `Customer` record is added.
- A new `CustomerOrder` record is added.
- New `OrderedProduct` records are added, according to the items contained in the `ShoppingCart`.

We'll implement this by creating a `placeOrder` method which performs the three write actions by sequentially calling private helper methods, `addCustomer`, `addOrder`, and `addOrderedItems`. We'll also implement the three helper methods in the class. To leverage EJB's container-managed transaction service, we only require two annotations. These are:

- `@TransactionManagement(TransactionManagementType.`*`CONTAINER`*`)` : Used to specify that any transactions occurring in the class are container-managed.
- `@TransactionAttribute(TransactionAttributeType.`*`REQUIRED`*`)` : Used on the method that invokes the transaction to specify that a new transaction should be created (if one does not already exist).



Because we are implementing the transaction within a larger context, we'll approach this exercise by dividing it into several easily-digestible tasks.

- Examining the Project Snapshot
- Creating the OrderManager EJB
- Handling Request Parameters
- Implementing `placeOrder` and Helper Methods
- Utilizing JPA's EntityManager
- Synchronizing the Persistence Context with the Database
- Setting up the Transaction Programmatically

Examining the Project Snapshot

Begin by examining the project snapshot associated with this tutorial unit.

1. Open the project snapshot for this tutorial unit in the IDE. Click the Open Project ( ) button and use the wizard to navigate to the location on your computer where you downloaded the project. If you are proceeding from the previous tutorial unit, note that this project snapshot is identical to the state of the project after completing the previous unit, but with the following exceptions:
   - The `confirmation.jsp` page is fully implemented.
   - The `affablebean.css` stylesheet includes rules specific to the `confirmation.jsp` page implementation.
2. Run the project ( ) to ensure that it is properly configured with your database and application server.

   If you receive an error when running the project, revisit the setup instructions, which describe how to prepare the database and establish connectivity between the IDE, GlassFish, and MySQL.
3. Test the application's functionality in your browser. In particular, step through the entire business process flow. When you click the submit an order from the checkout page, the confirmation page currently displays as follows:

   No data related to the order is displayed on the confirmation page. In fact, in its current state the application doesn't do anything with the data from the checkout form. By the end of this tutorial unit, the application will gather customer data and use it to process an order. In its final state, the application will display a summary of the processed order on the confirmation page, remove the user's `ShoppingCart` and terminate the user session. (Snapshot 8 completes the request-response cycle when a checkout form is submitted.)

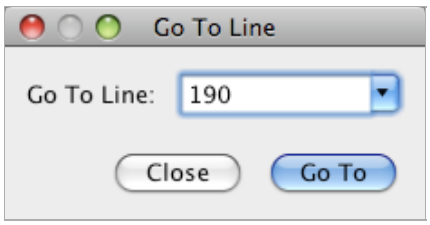
Creating the OrderManager EJB

1. Click the New File ( ) button in the IDE's toolbar. (Alternatively, press Ctrl-N; ⌘-N on Mac.) In the New File wizard, select the Java EE category, then select Session Bean.
2. Click Next. Name the EJB '`OrderManager`', place the EJB in the `session` package, and accept other default settings. (Create a stateless session bean, and do not have the wizard generate an interface for the bean.)
3. Click Finish. The new `OrderManager` class is generated and opens in the editor.

Handling Request Parameters

1. Open the project's `ControllerServlet`. (Either select it from the Projects window, or press Alt-Shift-O (Ctrl-Shift-O on Mac) and use the Go to File dialog.)
2. Locate the area in the `doPost` method where the `/purchase` request will be implemented (line 190).

   Press Ctrl-G to use the Go To Line dialog.



3. Implement code that extracts the parameters from a submitted checkout form. Locate the `TODO: Implement purchase action` comment, delete it, and add the following:

```
// if purchase action is called
} else if (userPath.equals("/purchase")) {

    if (cart != null) {

        // extract user data from request
        String name = request.getParameter("name");
        String email = request.getParameter("email");
        String phone = request.getParameter("phone");
        String address = request.getParameter("address");
        String cityRegion = request.getParameter("cityRegion");
        String ccNumber = request.getParameter("creditcard");
    }

    userPath = "/confirmation";
}
```

Implementing placeOrder and Helper Methods

1. In the `ControllerServlet`, add a reference to the `OrderManager` EJB. Scroll to the top of the class and add a reference beneath the session facade EJBs that are already listed.

```
public class ControllerServlet extends HttpServlet {

    private String userPath;
    private String surcharge;
    private ShoppingCart cart;

    @EJB
    private CategoryFacade categoryFacade;
    @EJB
    private ProductFacade productFacade;
    @EJB
    private OrderManager orderManager;
```

2. Press Ctrl-Shift-I (⌘:-Shift-I on Mac) to allow the editor to add an import statement for `session.OrderManager`.
3. Use the extracted parameters, as well as the session `cart` object, as arguments for the `OrderManager.placeOrder` method. Add the following code:

```
// if purchase action is called
} else if (userPath.equals("/purchase")) {

    if (cart != null) {

        // extract user data from request
        String name = request.getParameter("name");
        String email = request.getParameter("email");
        String phone = request.getParameter("phone");
        String address = request.getParameter("address");
        String cityRegion = request.getParameter("cityRegion");
        String ccNumber = request.getParameter("creditcard");
```

```
        int orderId = orderManager.placeOrder(name, email, phone, address,
cityRegion, ccNumber, cart);
    }

    userPath = "/confirmation";
}
```

Note that we haven't created the `placeOrder` method yet. This is why the editor flags an error. You can use the tip that displays in the left margin, which allows you to generate the method signature in the appropriate class.

---

4. Click the tip. The IDE generates the `placeOrder` method in the `OrderManager` class.

```
@Stateless
public class OrderManager {

    public int placeOrder(String name, String email, String phone, String address,
String cityRegion, String ccNumber, ShoppingCart cart) {
        throw new UnsupportedOperationException("Not yet implemented");
    }

    ...
}
```

The import statement for `cart.ShoppingCart` is also automatically inserted at the top of the file.

5. In the new `placeOrder` method, use the method arguments to make calls to the (yet nonexistent) helper methods. Enter the following:

```
public int placeOrder(String name, String email, String phone, String address, String
cityRegion, String ccNumber, ShoppingCart cart) {

    Customer customer = addCustomer(name, email, phone, address, cityRegion,
ccNumber);
    CustomerOrder order = addOrder(customer, cart);
    addOrderedItems(order, cart);
}
```

Note that we need to follow a particular order due to database constraints. For example, a `Customer` record needs to be created before the `CustomerOrder` record, since the `CustomerOrder` requires a reference to a `Customer`. Likewise, the `OrderedItem` records require a reference to an existing `CustomerOrder`.

6. Press Ctrl-Shift-I (⌘:-Shift-I on Mac) to fix imports. Import statements for `entity.Customer` and `entity.CustomerOrder` are automatically added to the top of the file.

7. Use the editor hints to have the IDE generate method signatures for `addCustomer`, `addOrder`, and `addOrderedItems`. After utilizing the three hints, the `OrderManager` class looks as follows.

```
@Stateless
public class OrderManager {

    public int placeOrder(String name, String email, String phone, String address,
String cityRegion, String ccNumber, ShoppingCart cart) {

        Customer customer = addCustomer(name, email, phone, address, cityRegion,
ccNumber);
```

```
        CustomerOrder order = addOrder(customer, cart);
        addOrderedItems(order, cart);
    }

    private Customer addCustomer(String name, String email, String phone, String
address, String cityRegion, String ccNumber) {
        throw new UnsupportedOperationException("Not yet implemented");
    }

    private CustomerOrder addOrder(Customer customer, ShoppingCart cart) {
        throw new UnsupportedOperationException("Not yet implemented");
    }

    private void addOrderedItems(CustomerOrder order, ShoppingCart cart) {
        throw new UnsupportedOperationException("Not yet implemented");
    }

}
```

Note that an error is still flagged in the editor, due to the fact that the method is currently lacking a return statement. The `placeOrder` signature indicates that the method returns an `int`. As will later be demonstrated, the method returns the order ID if it has been successfully processed, otherwise `0` is returned.

8. Enter the following return statement.

```
public int placeOrder(String name, String email, String phone, String address, String
cityRegion, String ccNumber, ShoppingCart cart) {

    Customer customer = addCustomer(name, email, phone, address, cityRegion,
ccNumber);
    CustomerOrder order = addOrder(customer, cart);
    addOrderedItems(order, cart);
    return order.getId();
}
```

At this stage, all errors in the `OrderManager` class are resolved.

9. Begin implementing the three helper methods. For now, simply add code that applies each method's input parameters to create new entity objects.

addCustomer

Create a new `Customer` object and return the object.

```
private Customer addCustomer(String name, String email, String phone, String address,
String cityRegion, String ccNumber) {

    Customer customer = new Customer();
    customer.setName(name);
    customer.setEmail(email);
    customer.setPhone(phone);
    customer.setAddress(address);
    customer.setCityRegion(cityRegion);
    customer.setCcNumber(ccNumber);

    return customer;
}
```

addOrder

Create a new `CustomerOrder` object and return the object. Use the `java.util.Random` class to generate a random confirmation number.

```
private CustomerOrder addOrder(Customer customer, ShoppingCart cart) {

    // set up customer order
    CustomerOrder order = new CustomerOrder();
    order.setCustomer(customer);
    order.setAmount(BigDecimal.valueOf(cart.getTotal()));

    // create confirmation number
    Random random = new Random();
    int i = random.nextInt(999999999);
    order.setConfirmationNumber(i);

    return order;
}
```

addOrderedItems

Iterate through the `ShoppingCart` and create `OrderedProducts`. In order to create an `OrderedProduct`, you can use the `OrderedProductPK` entity class. The instantiated `OrderedProductPK` can be passed to the `OrderedProduct` constructor, as demonstrated below.

```
private void addOrderedItems(CustomerOrder order, ShoppingCart cart) {

    List<ShoppingCartItem> items = cart.getItems();

    // iterate through shopping cart and create OrderedProducts
    for (ShoppingCartItem scItem : items) {

        int productId = scItem.getProduct().getId();

        // set up primary key object
        OrderedProductPK orderedProductPK = new OrderedProductPK();
        orderedProductPK.setCustomerOrderId(order.getId());
        orderedProductPK.setProductId(productId);

        // create ordered item using PK object
        OrderedProduct orderedItem = new OrderedProduct(orderedProductPK);

        // set quantity
        orderedItem.setQuantity(scItem.getQuantity());
    }
}
```

10. Press Ctrl-Shift-I (⌘-Shift-I on Mac) to fix imports. A dialog opens to display all classes that will be imported. Note that the dialog correctly guesses for `java.util.List`.

11. Click OK. All necessary import statements are added, and the class becomes free of any compiler errors.

Utilizing JPA's EntityManager

As was mentioned in Adding Entity Classes and Session Beans, the `EntityManager` API is included in JPA, and is responsible for performing persistence operations on the database. In the `AffableBean` project, all of the EJBs employ the `EntityManager`. To demonstrate, open any of the session facade beans in the editor and note that the class uses the `@PersistenceContext` annotation to express a dependency on a container-managed `EntityManager` and its associated persistence context (`AffableBeanPU`, as specified in the `persistence.xml` file). For example, the `ProductFacade` bean looks as follows:

```
@Stateless
public class ProductFacade extends AbstractFacade<Product> {
    @PersistenceContext(unitName = "AffableBeanPU")
    private EntityManager em;

    protected EntityManager getEntityManager() {
        return em;
    }

    ...

    // manually created
    public List<Product> findForCategory(Category category) {
        return em.createQuery("SELECT p FROM Product p WHERE p.category = :category").
               setParameter("category", category).getResultList();
    }

}
```

To be able to write to the database, the `OrderManager` EJB must take similar measures. With an `EntityManager` instance, we can then modify the helper methods (`addCustomer`, `addOrder`, `addOrderedItems`) so that the entity objects they create are written to the database.

1. In `OrderManager`, apply the `@PersistenceContext` annotation to express a dependency on a container-managed `EntityManager` and the `AffableBeanPU` persistence context. Also declare an `EntityManager` instance.

   ```
   @Stateless
   public class OrderManager {

       @PersistenceContext(unitName = "AffableBeanPU")
       private EntityManager em;

       ...
   }
   ```

2. Press Ctrl-Shift-I (⌘:-Shift-I on Mac) to fix imports. Import statements for `javax.persistence.EntityManager` and `javax.persistence.PersistenceContext` are added to the top of the class.
3. Use the `EntityManager` to mark entity objects to be written to the database. This is accomplished using the `persist` method in the `EntityManager` API. Make the following modifications to the helper methods.

   addCustomer

   ```
   private Customer addCustomer(String name, String email, String phone, String address,
   ```

```
String cityRegion, String ccNumber) {

    Customer customer = new Customer();
    customer.setName(name);
    customer.setEmail(email);
    customer.setPhone(phone);
    customer.setAddress(address);
    customer.setCityRegion(cityRegion);
    customer.setCcNumber(ccNumber);

    em.persist(customer);
    return customer;
}
```

addOrder

```
private CustomerOrder addOrder(Customer customer, ShoppingCart cart) {

    // set up customer order
    CustomerOrder order = new CustomerOrder();
    order.setCustomer(customer);
    order.setAmount(BigDecimal.valueOf(cart.getTotal()));

    // create confirmation number
    Random random = new Random();
    int i = random.nextInt(999999999);
    order.setConfirmationNumber(i);

    em.persist(order);
    return order;
}
```

addOrderedItems

```
private void addOrderedItems(CustomerOrder order, ShoppingCart cart) {

    List<ShoppingCartItem> items = cart.getItems();

    // iterate through shopping cart and create OrderedProducts
    for (ShoppingCartItem scItem : items) {

        int productId = scItem.getProduct().getId();

        // set up primary key object
        OrderedProductPK orderedProductPK = new OrderedProductPK();
        orderedProductPK.setCustomerOrderId(order.getId());
        orderedProductPK.setProductId(productId);

        // create ordered item using PK object
        OrderedProduct orderedItem = new OrderedProduct(orderedProductPK);

        // set quantity
        orderedItem.setQuantity(String.valueOf(
```