Lab 5 Shopping Basket - Play Framework

[Introduction 1](#_Toc446026290)

[1. Description 2](#_Toc446026291)

[2. Entity Object Model 3](#_Toc446026292)

[3. Object Relationship Mapping 4](#_Toc446026293)

[3.1 OrderItem 5](#_Toc446026294)

[3.2 ShopOrder 6](#_Toc446026295)

[3.3 Customer 7](#_Toc446026296)

[3.4 Mapping Notes 7](#_Toc446026297)

[3.5 Generated DB tables 8](#_Toc446026298)

[4. Controller and Views 9](#_Toc446026299)

[4.1 Adding items to the basket 9](#_Toc446026300)

[4.2 Viewing the shopping basket 12](#_Toc446026301)

[4.3 Increment and Decrement item quantities 13](#_Toc446026302)

[4.4 Removing all items from the basket 15](#_Toc446026303)

[4.5 Placing an Order 16](#_Toc446026304)

[5. To Do 17](#_Toc446026305)

## Introduction

Download Lab5ShoppingCart.zip from Moodle, extract and open in IntelliJ.

This tutorial describes how to add a shopping basket and product ordering to the products web application. The application described here is based on examples from prerequisite labs and tutorials which should be attempted before continuing.

Note that the application is a prototype with basic functionality, many features of a fully functional e-commerce/ shopping web application are not implemented.

## 1. Description

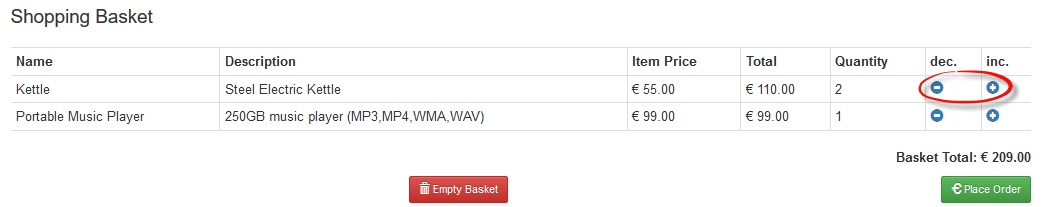
Administrative functionality was described previously - this is a description of how the application works from a customer perspective.

On visiting the site, the user will see a list of products and categories. All products are displayed by default but the user can choose to see products for a single category only. A product may be included in more than one category.

The user has the option of adding a product to his/ her shopping basket. To do this the user must be registered as a customer and logged in. Attempts by non-customer users to view or add a product to the bask will be redirected to the login view.



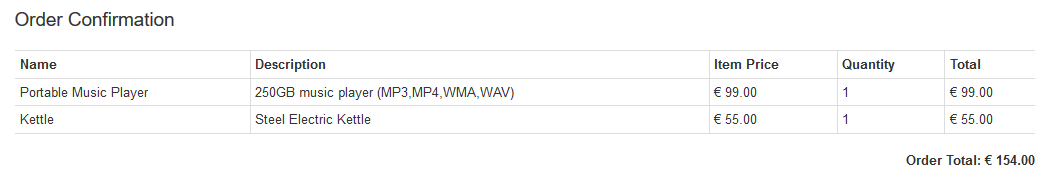
When an item is added to the basket, the shopping basket view is displayed. This view shows each item added, quantities, and prices. The basket (and order items) are stored in the database and so persist across customer sessions.



The customer may increase or decrease the quantity of each item in the basket. Totals are adjusted accordingly.

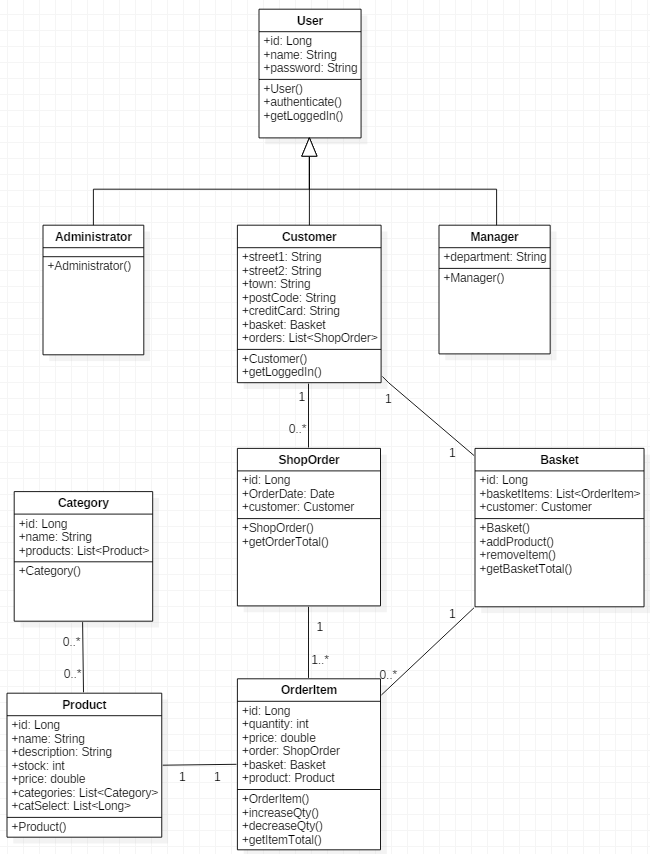
The Empty Basket button removes all items from the basket (and the database).

The place order button also empties order items from the basket but does not delete them. Instead an order is created and the items are associated with that. After placing an order, the order confirmation view is displayed.



## 2. Entity Object Model

This is the entity object model used in the application. Note the relationships between the classes.



## 3. Object Relationship Mapping

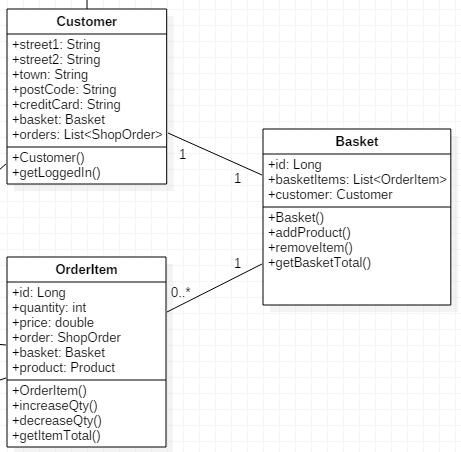
As the application uses code-first ORM, all database tables, relationships, constraints, sequences, indices, and keys are created based on the object model and mapping definitions. When using code-first you should not manually alter the database or evolution SQL as this can cause conflicts.

This section will explain the entity models and mapping definitions in detail, and show the resulting database tables.

This application adds some new classes (models.shopping):

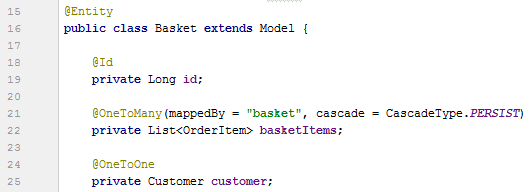
#### Basket

There is one basket for each customer. The basket may contain zero to many order items.



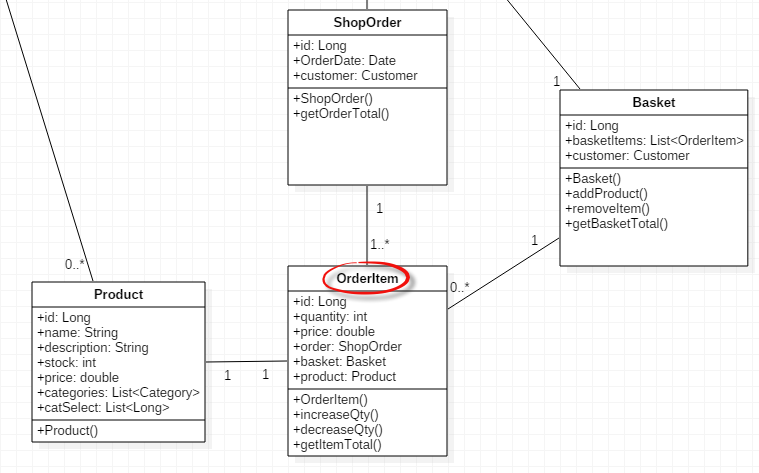
The Basket class: Add the following mappings to Basket.java

1. The OneToMany mapping is mappedBy basket in the OrderItem class. mappedBy here indicates that Basket is the owner side of the mapping (the ‘one side’ of the one to many) and that a foreign key will be included in the Order\_Item table in the database.
2. CascadeType.PERSIST ensures that basketItems will be updated (persisted) but not removed. This is important as they must not be removed if the items are ordered.
3. A OneToOne mapping to Customer.



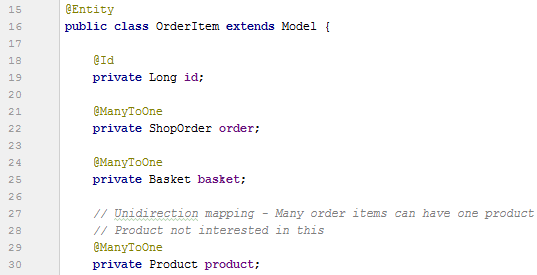
### 3.1 OrderItem

Order items are created when a product is added to a basket. In addition to product details, order items records the price at time of order and quantity ordered. The class is mapped to three others, Basket, ShopOrder, and Product.



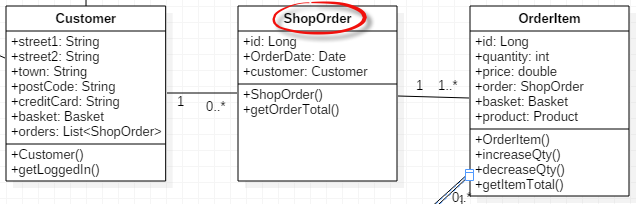
The OrderItem class: Add the following mappings to OrderItem.java

1. ManyToOne mapping to ShopOrder.
2. ManyToOne mapping to Basket.
3. The diagram shows a one to one relationship but that would create a table constraint requiring unique product ids – the result – duplicate key errors. A unidirectional ManyToOne mapping to Product avoids that problem.



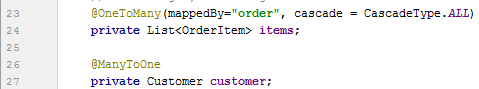
### 3.2 ShopOrder

When an order is created, order items are moved from the basket and added to the new ShopOrder instance. A customer has just one basket but may place many orders.



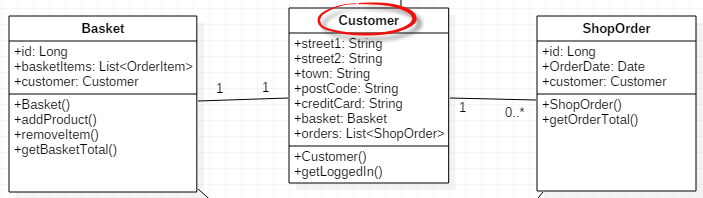
The ShopOrder class: Add the following mappings to ShopOrder.java

1. ShopOrder is the ‘owning side’ of the mapping with OrderItem. During mapping a foreign key will be added to the Order\_ITEM table. All changes, including deletes, are cascaded.
2. ManyToOne mapping to Customer.



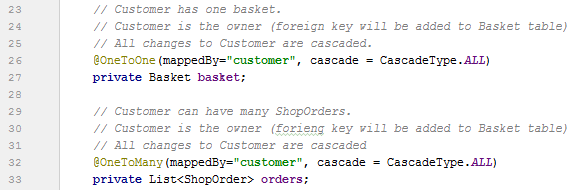
### 3.3 Customer

Customer is mapped to a Basket and ShopOrders



The Customer class: Add the following mappings to Customer.java

1. Customer is the ‘owning side’ of the OneToOne mapping with Basket. During mapping a foreign key will be added to the BASKET table. All changes, including deletes, are cascaded.
2. The OneToMany mapping to ShopOrder is also denoted as the owning side.



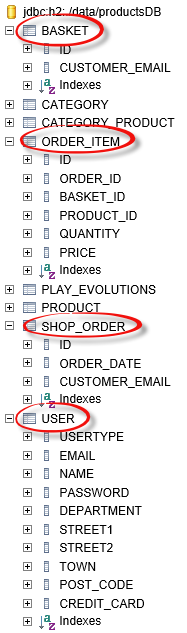
### 3.4 Mapping Notes

The mappedBy attribute described here is optional, however explicitly setting it ensures that mapping is performed as expected.

For more cascade options see: <http://stackoverflow.com/questions/13027214/jpa-manytoone-with-cascadetype-all>

### 3.5 Generated DB tables

This is a snapshot of the tables generated by the ORM. Note the table names and foreign keys generated.

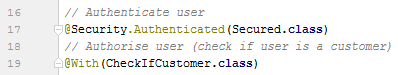


## 4. Controller and Views

This section looks at how the shopping part of the application works, highlighting important functionality in the Shopping controller and views.

The shopping controller class is defined in controllers/ShoppingCtrl.java. The functionality of this class is available only to users of type Customer (e.g. [customer@products.com](mailto:customer@products.com) password: **password**) – enforced by the annotations shown below.

Add these two annotations to ShoppingCtrl.java. Take a look at the CheckIfCustomer.java class.



### 4.1 Adding items to the basket

The list products view displays a shopping trolley icon next to each product.



Clicking this calls the addToBasket(Long id) method in ShoppingCtrl, passing the product id as a parameter. This line of code is already added to listProducts.scala.html.



addToBasket(Long id) in ShoppingCtrl

1. Finds the product (by id)
2. Gets the basket for this customer
3. Create a basket if the customer doesn’t have one
4. Call the basket addProduct(Product) method

Write this method in ShoppingCtrl.java

@Transactional

public Result addToBasket(Long id) {

// Find the product

Product p = Product.find.byId(id);

// Get basket for logged in customer

Customer customer = (Customer)User.getLoggedIn(session().get("email"));

// Check if item in basket

if (customer.getBasket() == null) {

// If no basket, create one

customer.setBasket(new Basket());

customer.getBasket().setCustomer(customer);

customer.update();

}

// Add product to the basket and save

customer.getBasket().addProduct(p);

customer.update();

// Show the basket contents

return ok(basket.render(customer));

}

#### addProduct(Product P) in Basket

1. This method (see Basket.java) first looks for an existing basketItem (An OrderItem list) for the product being added.
2. If found, then increment the item quantity.
3. If not found, then add a new OrderItem to the basket

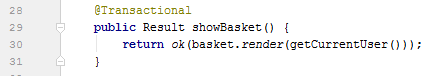
Write this method in Basket.java

*// Add product to basket  
// Either update existing order item or ad a new one.***public void** addProduct(Product p) {  
   
 **boolean** itemFound = **false**;  
 *// Check if product already in this basket  
 // Check if item in basket  
 // Find orderitem with this product  
 // if found increment quantity* **for** (OrderItem i : **basketItems**) {  
 **if** (i.getProduct().getId() == p.getId()) {  
 i.increaseQty();  
 itemFound = **true**;  
 **break**;  
 }  
 }  
 **if** (itemFound == **false**) {  
 *// Add orderItem to list* OrderItem newItem = **new** OrderItem(p);  
 *// Add to items* **basketItems**.add(newItem);  
 }  
}

### 4.2 Viewing the shopping basket

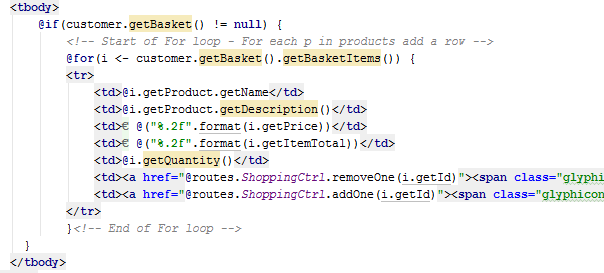
Displayed when the menubar link is clicked or after an item has been added to the basket. The showBasket() method in ShoppingCtrl renders the basket.scala.html view – passing the logged in customer.

Write this method in ShoppingCtrl.java



1. The view includes a loop which iterates through the customer’s basketItems, displaying each.
2. Note the currency formatting and the call to getItemTotal()
3. Links to increment and decrement item quantities
4. Call getBasketTotal() to display the total price of all basket items

This code is already provided in the scala file.



### 4.3 Increment and Decrement item quantities

The shopping basket links allow quantities to be adjusted up or down by one.



The requests are handled by the addOne() and removeOne() methods in ShoppingCtrl.

Add these methods to ShoppingCtrl.java

// Add an item to the basket

@Transactional

public Result addOne(Long itemId) {

// Get the order item

OrderItem item = OrderItem.find.byId(itemId);

// Increment quantity

item.increaseQty();

// Save

item.update();

// Show updated basket

return redirect(routes.ShoppingCtrl.showBasket());

}

@Transactional

public Result removeOne(Long itemId) {

// Get the order item

OrderItem item = OrderItem.find.byId(itemId);

// Get user

Customer c = getCurrentUser();

// Call basket remove item method

c.getBasket().removeItem(item);

c.getBasket().update();

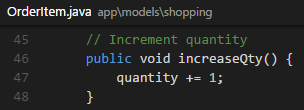
// back to basket

return ok(basket.render(c));

}

To add an item, call the OrderItem method increaseQty() which simply increments the objects quantity attribute.

This method is already provided in OrderItem.java



1. Decrementing quantity is a little more complex as if the quantity is decreased to zero then the item must be removed completely. The Basket method, removeItem(), does this:
   1. Use Java Iterator to safely navigate list objects. Simply using a list index is not considered safe as items could be added or removed elsewhere, which could lead to removal of the wrong object. The Iterator uses an object reference which is consistent.
   2. If quantity > 1 simply decrement
   3. If quantity == 1 then delete the object from DB then use Iterator instance to remove from the basket.

Write this method in Basket.java

public void removeItem(OrderItem item) {

// Using an iterator ensures 'safe' removal of list objects

// Removal of list items is unreliable as index can change if an item is added or removed elsewhere

// iterator works with an object reference which does not change

for (Iterator<OrderItem> iter = basketItems.iterator(); iter.hasNext();) {

OrderItem i = iter.next();

if (i.getId().equals(item.getId()))

{

// If more than one of these items in the basket then decrement

if (i.getQuantity() > 1 ) {

i.decreaseQty();

}

// If only one left, remove this item from the basket (via the iterator)

else {

// delete object from db

i.delete();

// remove object from list

iter.remove();

break;

}

}

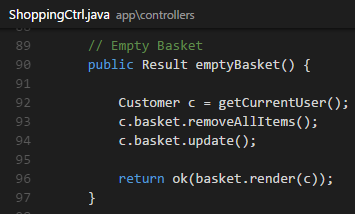
}

}

### 4.4 Removing all items from the basket

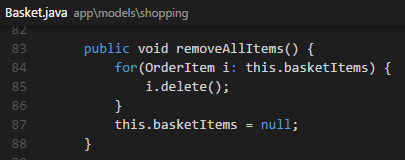
Clicking the Empty Basket button in the basket view causes all items in the basket to be deleted by calling the emptyBasket() method in the shopping controller:

This method is already provided in ShoppingCtrl.java



1. This in turn calls the Basket removeAllItems() method which calls delete() for each OrderItem instance in the basket.
2. It then sets the basketItems list = null

This method is already provided in Basket.java

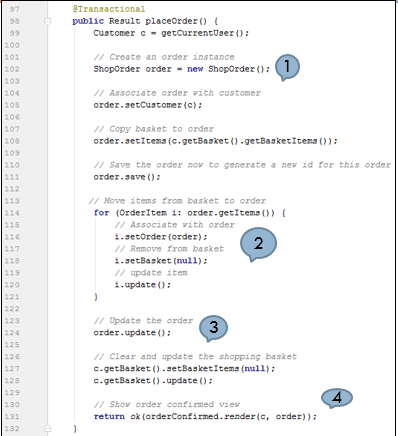


### 4.5 Placing an Order

When an order is placed (via the button in the basket view):

1. A new ShopOrder instance is created
2. OrderItems are removed from the basket and added to the order
3. Update order item objects
4. An order confirmation is displayed.

Add this method to ShoppingCtrl.java



Log in as a customer and try and add/delete items from the shopping basket and place an order.