```
package org.apache.ofbiz.order.shoppingcart.product;

import java.math.BigDecimal;
import java.math.MathContext;
import java.util.Collections;
import java.util.Comparator;
import java.util.HashMap;
import java.util.Iterator;
import java.util.LinkedList;
import java.util.List;
import java.util.Map;

import javax.servlet.ServletRequest;
import javax.servlet.http.HttpServletRequest;

import org.apache.ofbiz.base.util.Debug;
import org.apache.ofbiz.base.util.UtilGenerics;
import org.apache.ofbiz.base.util.UtilMisc;
import org.apache.ofbiz.base.util.UtilNumber;
import org.apache.ofbiz.base.util.UtilValidate;
import org.apache.ofbiz.entity.Delegator;
import org.apache.ofbiz.entity.GenericEntity;
import org.apache.ofbiz.entity.GenericEntityException;
import org.apache.ofbiz.entity.GenericValue;
import org.apache.ofbiz.entity.util.EntityQuery;
import org.apache.ofbiz.order.shoppingcart.ShoppingCart;
import org.apache.ofbiz.order.shoppingcart.ShoppingCartItem;
import org.apache.ofbiz.product.catalog.CatalogWorker;
import org.apache.ofbiz.product.category.CategoryWorker;
import org.apache.ofbiz.product.product.ProductWorker;


public final class ProductDisplayWorker {

    public static final String module = ProductDisplayWorker.class.getName()
        ;

    private ProductDisplayWorker() {}
```

```
/*
    ==============================================================
    =====================*/

/* ============================= Special Data Retrieval Methods
    ===========================*/

public static List<GenericValue> getRandomCartProductAssoc
    (ServletRequest request, boolean checkViewAllow) {
    Delegator delegator = (Delegator) request.getAttribute("delegator");
    HttpServletRequest httpRequest = (HttpServletRequest) request;
    ShoppingCart cart = (ShoppingCart) httpRequest.getSession().
        getAttribute("shoppingCart");

    if (cart == null || cart.size() <= 0) return null;

    List<GenericValue> cartAssocs = null;
    try {
        Map<String, GenericValue> products = new HashMap<String,
            GenericValue>();

        Iterator<ShoppingCartItem> cartiter = cart.iterator();

        while (cartiter != null && cartiter.hasNext()) {
            ShoppingCartItem item = cartiter.next();
            // since ProductAssoc records have a fromDate and thruDate,
                we can filter by now so that only assocs in the date
                range are included
            List<GenericValue> complementProducts = EntityQuery.use
                (delegator).from("ProductAssoc").where("productId", item
                .getProductId(), "productAssocTypeId",
                "PRODUCT_COMPLEMENT").cache(true).filterByDate().
                queryList();

            List<GenericValue> productsCategories = EntityQuery.use
                (delegator).from("ProductCategoryMember").where("product
                Id", item.getProductId()).cache(true).filterByDate().
                queryList();
            if (productsCategories != null) {
                for (GenericValue productsCategoryMember :
                    productsCategories) {
                    GenericValue productsCategory =
                        productsCategoryMember.getRelatedOne("ProductCat
                        egory", true);
                    if ("CROSS_SELL_CATEGORY".equals(productsCategory.
                        getString("productCategoryTypeId"))) {
                        List<GenericValue> curPcms = productsCategory.
                            getRelated("ProductCategoryMember", null,
                            null, true);
                        if (curPcms != null) {
                            for (GenericValue curPcm : curPcms) {
                                if (!products.containsKey(curPcm.
                                    getString("productId"))) {
                                    GenericValue product = curPcm.
                                        getRelatedOne("Product", true);

                                        products.put(product.getString("
                                        productId"), product);
                                }
                            }
                        }
                    }
                }
            }

            if (UtilValidate.isNotEmpty(complementProducts)) {
                for (GenericValue productAssoc : complementProducts) {
                    if (!
                        products.containsKey(productAssoc.getString("pro
                        ductIdTo"))) {
                        GenericValue product = productAssoc.
                            getRelatedOne("AssocProduct", true);
                        products.put(product.getString("productId"),
                            product);
                    }
                }
            }
        }

        // remove all products that are already in the cart
        cartiter = cart.iterator();
        while (cartiter != null && cartiter.hasNext()) {
            ShoppingCartItem item = cartiter.next();
            products.remove(item.getProductId());
        }

        // if desired check view allow category
        if (checkViewAllow) {
            String currentCatalogId = CatalogWorker.getCurrentCatalogId
                (request);
            String viewProductCategoryId = CatalogWorker.
                getCatalogViewAllowCategoryId(delegator,
                currentCatalogId);
            if (viewProductCategoryId != null) {
                List<GenericValue> tempList = new LinkedList<
                    GenericValue>();
                tempList.addAll(products.values());
                tempList = CategoryWorker.filterProductsInCategory
                    (delegator, tempList, viewProductCategoryId,
                    "productId");
                cartAssocs = new LinkedList<GenericValue>();
                cartAssocs.addAll(tempList);
            }
        }

        if (cartAssocs == null) {
            cartAssocs = new LinkedList<GenericValue>();
            cartAssocs.addAll(products.values());
        }

        // randomly remove products while there are more than 3
        while (cartAssocs.size() > 3) {
            int toRemove = (int) (Math.random() *  cartAssocs.size());
            cartAssocs.remove(toRemove);
        }
    } catch (GenericEntityException e) {
        Debug.logWarning(e, module);
    }

    if (UtilValidate.isNotEmpty(cartAssocs)) {
        return cartAssocs;
    } else {
        return null;
    }
}

public static Map<String, Object> getQuickReorderProducts(ServletRequest
    request) {
    Delegator delegator = (Delegator) request.getAttribute("delegator");
    HttpServletRequest httpRequest = (HttpServletRequest) request;
    GenericValue userLogin = (GenericValue) httpRequest.getSession().
        getAttribute("userLogin");
    Map<String, Object> results = new HashMap<String, Object>();

    if (userLogin == null) userLogin = (GenericValue) httpRequest.
        getSession().getAttribute("autoUserLogin");
    if (userLogin == null) return results;

    try {
        Map<String, GenericValue> products = UtilGenerics.checkMap
            (httpRequest.getSession().getAttribute("_QUICK_REORDER_PRODU
            CTS_"));
        Map<String, BigDecimal> productQuantities = UtilGenerics.
            checkMap(httpRequest.getSession().getAttribute("_QUICK_REORD
            ER_PRODUCT_QUANTITIES_"));
        Map<String, Integer> productOccurances = UtilGenerics.checkMap
            (httpRequest.getSession().getAttribute("_QUICK_REORDER_PRODU
            CT_OCCURANCES_"));

        if (products == null || productQuantities == null ||
            productOccurances == null) {
            products = new HashMap<String, GenericValue>();
            productQuantities = new HashMap<String, BigDecimal>();
            // keep track of how many times a product occurs in order to
                find averages and rank by purchase amount
            productOccurances = new HashMap<String, Integer>();

            // get all order role entities for user by customer role
                type : PLACING_CUSTOMER
            List<GenericValue> orderRoles = EntityQuery.use(delegator).
                from("OrderRole").where("partyId",
                userLogin.get("partyId"), "roleTypeId",
                "PLACING_CUSTOMER").queryList();
            Iterator<GenericValue> ordersIter = UtilMisc.toIterator
                (orderRoles);

            while (ordersIter != null && ordersIter.hasNext()) {
                GenericValue orderRole = ordersIter.next();
                // for each order role get all order items
                List<GenericValue> orderItems =
                    orderRole.getRelated("OrderItem", null, null, false)
                    ;
                Iterator<GenericValue> orderItemsIter = UtilMisc.
                    toIterator(orderItems);

                while (orderItemsIter != null && orderItemsIter.hasNext
                    ()) {
                    GenericValue orderItem = orderItemsIter.next();
                    String productId = orderItem.getString("productId");
                    if (UtilValidate.isNotEmpty(productId)) {
```

```
            // for each order item get the associated
                product
            GenericValue product =
                orderItem.getRelatedOne("Product", true);

            products.put(product.getString("productId"),
                product);

            BigDecimal curQuant = productQuantities.get
                (product.get("productId"));

            if (curQuant == null) curQuant = BigDecimal.ZERO
                ;
            BigDecimal orderQuant =
                orderItem.getBigDecimal("quantity");

            if (orderQuant == null) orderQuant = BigDecimal.
                ZERO;

                productQuantities.put(product.getString("pro
                ductId"), curQuant.add(orderQuant));

            Integer curOcc = productOccurances.get(product.
                get("productId"));

            if (curOcc == null) curOcc = Integer.valueOf(0);

                productOccurances.put(product.getString("pro
                ductId"), Integer.valueOf(curOcc.intValue()
                + 1));
        }
    }
}

// go through each product quantity and divide it by the
    occurances to get the average
for (Map.Entry<String, BigDecimal> entry : productQuantities
    .entrySet()) {
    String prodId = entry.getKey();
    BigDecimal quantity = entry.getValue();
    Integer occs = productOccurances.get(prodId);
    BigDecimal nqint = quantity.divide(new BigDecimal(occs),
        new MathContext(10));

    if (nqint.compareTo(BigDecimal.ONE) < 0) nqint =
        BigDecimal.ONE;
    productQuantities.put(prodId, nqint);
}


    httpRequest.getSession().setAttribute("_QUICK_REORDER_PR
    ODUCTS_", new HashMap<String, GenericValue>(products));

    httpRequest.getSession().setAttribute("_QUICK_REORDER_PR
    ODUCT_QUANTITIES_", new HashMap<String, BigDecimal>
    (productQuantities));

    httpRequest.getSession().setAttribute("_QUICK_REORDER_PR
    ODUCT_OCCURANCES_", new HashMap<String, Integer>
    (productOccurances));
```

```
        } else {
            // make a copy since we are going to change them
            products = new HashMap<String, GenericValue>(products);
            productQuantities = new HashMap<String, BigDecimal>
                (productQuantities);
            productOccurances = new HashMap<String, Integer>
                (productOccurances);
        }

        // remove all products that are already in the cart
        ShoppingCart cart = (ShoppingCart) httpRequest.getSession().
            getAttribute("shoppingCart");
        if (UtilValidate.isNotEmpty(cart)) {
            for (ShoppingCartItem item : cart) {
                String productId = item.getProductId();
                products.remove(productId);
                productQuantities.remove(productId);
                productOccurances.remove(productId);
            }
        }

        // if desired check view allow category
            String currentCatalogId = CatalogWorker.getCurrentCatalogId
                (request);
            String viewProductCategoryId = CatalogWorker.
                getCatalogViewAllowCategoryId(delegator,
                currentCatalogId);
            if (viewProductCategoryId != null) {
                for (Map.Entry<String, GenericValue> entry : products.
                    entrySet()) {
                    String productId = entry.getKey();
                    if (!CategoryWorker.isProductInCategory(delegator,
                        productId, viewProductCategoryId)) {
                        products.remove(productId);
                        productQuantities.remove(productId);
                        productOccurances.remove(productId);
                    }
                }
            }

        List<GenericValue> reorderProds = new LinkedList<GenericValue>()
            ;
        reorderProds.addAll(products.values());

        // sort descending by new metric...
        BigDecimal occurancesModifier = BigDecimal.ONE;
        BigDecimal quantityModifier = BigDecimal.ONE;
        Map<String, Object> newMetric = new HashMap<String, Object>();
        for (Map.Entry<String, Integer> entry : productOccurances.
            entrySet()) {
            String prodId = entry.getKey();
            Integer quantity = entry.getValue();
            BigDecimal occs = productQuantities.get(prodId);
            //For quantity we should test if we allow to add decimal
                quantity for this product an productStore : if not then
                round to 0
            if(! ProductWorker.isDecimalQuantityOrderAllowed(delegator,
                prodId, cart.getProductStoreId())){
                occs = occs.setScale(0, UtilNumber.
                    getBigDecimalRoundingMode("order.rounding"));
                }
                else {
                    occs =
                        occs.setScale(UtilNumber.getBigDecimalScale("order.d
                        ecimals"),
                        UtilNumber.getBigDecimalRoundingMode("order.rounding
                        "));
                }
                productQuantities.put(prodId, occs);
                BigDecimal nqdbl = quantityModifier.multiply(new BigDecimal
                    (quantity)).add(occs.multiply(occurrencesModifier));

                newMetric.put(prodId, nqdbl);
            }
            reorderProds = productOrderByMap(reorderProds, newMetric, true);

            // remove extra products - only return 5
            while (reorderProds.size() > 5) {
                reorderProds.remove(reorderProds.size() - 1);
            }

            results.put("products", reorderProds);
            results.put("quantities", productQuantities);
        } catch (GenericEntityException e) {
            Debug.logWarning(e, module);
        }

        return results;
    }

    public static List<GenericValue> productOrderByMap(List<GenericValue>
        values, Map<String, Object> orderByMap, boolean descending) {
        if (values == null)  return null;
        if (values.size() == 0) return UtilMisc.toList(values);

        List<GenericValue> result = new LinkedList<GenericValue>();
        result.addAll(values);

        Collections.sort(result, new ProductByMapComparator(orderByMap,
            descending));
        return result;
    }

    private static class ProductByMapComparator implements Comparator<Object
        > {
        private Map<String, Object> orderByMap;
        private boolean descending;

        ProductByMapComparator(Map<String, Object> orderByMap, boolean
            descending) {
            this.orderByMap = orderByMap;
            this.descending = descending;
        }

        public int compare(java.lang.Object prod1, java.lang.Object prod2) {
            int result = compareAsc((GenericEntity) prod1, (GenericEntity)
                prod2);

            if (descending) {
                result = -result;
            }
            return result;
        }

        @SuppressWarnings("unchecked")
        private int compareAsc(GenericEntity prod1, GenericEntity prod2) {
            Object value = orderByMap.get(prod1.get("productId"));
            Object value2 = orderByMap.get(prod2.get("productId"));

            // null is defined as the smallest possible value
            if (value == null) return value2 == null ? 0 : -1;
            return ((Comparable<Object>) value).compareTo(value2);
        }

        @Override
        public boolean equals(java.lang.Object obj) {
            if ((obj != null) && (obj instanceof ProductByMapComparator)) {
                ProductByMapComparator that = (ProductByMapComparator) obj;

                return this.orderByMap.equals(that.orderByMap) && this.
                    descending == that.descending;
            } else {
                return false;
            }
        }
    }
}
```