Cookbook - Low Stock Indicator

Overview

It is sometimes important to the business to communicate stock levels to their customers when they are browsing the shop.

Some common reasons for this:

- Encourage sales ("Buy this item now, it's almost gone")
- Warn of longer delivery times ("Stock is low, next delivery time is X")

API

The platform provides **ProductAvailabilityModel** interface in order to distill information relevant to specific Product or ProductSku. This model is available in runtime for both domain objects and search index objects.

This model provides information whether the product is available (both in stock and on preorder), in stock (i.e. has available inventory) and type of its availability.

Wicket App

Wicket web app already uses ProductAvailabilityModel in order to switch between "Preorder" and "Add to Cart" buttons.

For example examining **ProductInListView** you come across this snippet:

```
     final long browsingShopId = getCurrentCustomerShopId();
       final ProductAvailabilityModel skuPam =
productServiceFacade.getProductAvailability(product, browsingShopId);
       final ShoppingCart cart = getCurrentCart();
       final PriceModel model = productServiceFacade.getSkuPrice(cart,
null, skuPam.getDefaultSkuCode(), BigDecimal.ONE);

       final boolean ableToAddDefault = !model.isPriceUponRequest() &&
skuPam.isAvailable() &&
skuPam.getDefaultSkuCode().equals(skuPam.getFirstAvailableSkuCode());

        add(links.newAddToCartLink(ADD_TO_CART_LINK,
skuPam.getDefaultSkuCode(), null, getPage().getPageParameters())
                       .add(new Label(ADD_TO_CART_LINK_LABEL,
skuPam.isInStock() || skuPam.isPerpetual() ?
                                   getLocalizer().getString("addToCart", this)
:
                                   getLocalizer().getString("preorderCart",
this)))
                       .setVisible(ableToAddDefault)
       );

        add(links.newProductLink(PRODUCT_LINK_ALT, product.getId(),
getPage().getPageParameters())
                       .add(new Label(VIEW_ALT_LABEL,
getLocalizer().getString("viewAllVariants", this)))
                       .setVisible(!ableToAddDefault)
       );
```

skuPam is the product availability model and by checking its properties, specifically:

```
skuPam.isAvailable() &&
skuPam.getDefaultSkuCode().equals(skuPam.getFirstAvailableSkuCode())
```

we are able to determine if the product is in stock. Further we can detect if it is preorder of regular products by checking:

```
skuPam.isInStock() || skuPam.isPerpetual()
```

Now that we are familiar with how product availability model we can explore how we can use this to display low stock indicators. And in fact it is fairly simple. Product availability model has **getAvailableToSellQuantity(sku)** method that return exact quantity of the inventory available. Thus we can use some business rules say "if quantity is less than 3 then display the message".

```
     final BigDecimal qty =
skuPam.getAvailableToSellQuantity(skuPam.getDefaultSkuCode());
       if (qty == null || qty.compareTo(new BigDecimal(3)) < 0) {
            warn(getLocalizer().getString("lowStockMessage", this));
       }
```

The snippet is self explanatory - we check the SKU quantity and if it is less than 3 we issue a warn message. This message will appear in the feedback component of the page. However you can use a label near the "Add to Cart" button, or have a special zone in the page were this information is displayed.

However checking for hard coded quantity is not particularly useful since well it is hard coded. However we could have an attribute say on SKU or PRODUCT to have this set dynamically. The snippet could be improved to the following:

```
  final Pair<String, I18NModel> lowStockAv =
product.getAttributes().getValue("LOW_STOCK_QTY");
       final BigDecimal lowStock = new
BigDecimal(NumberUtils.toDouble(lowStockAv != null ? lowStockAv.getFirst()
: "3", 3d));
       final BigDecimal qty =
skuPam.getAvailableToSellQuantity(skuPam.getDefaultSkuCode());
       if (qty == null || qty.compareTo(lowStock) < 0) {
            warn(getLocalizer().getString("lowStockMessage", this));
       }
```

Improved version uses "LOW_STOCK_QTY" attribute to determine the low stock for this particular product, with a fallback to value 3 if it is not available.

> When creating LOW_STOCK_QTY attribute make sure you set "stored" flag to *true* to make it available in the search index.

We can further improve to have a shop specific setting for low stock:

```
    final String shopLowStockAv =
getCurrentShop().getAttributeValueByCode("SHOP_LOW_STOCK_QTY");
       final double shopLowStock = NumberUtils.toDouble(shopLowStockAv);
       final Pair<String, I18NModel> lowStockAv =
product.getAttributes().getValue("LOW_STOCK_QTY");
       final BigDecimal lowStock = new
BigDecimal(NumberUtils.toDouble(lowStockAv != null ? lowStockAv.getFirst()
: shopLowStockAv, shopLowStock));
       final BigDecimal qty =
skuPam.getAvailableToSellQuantity(skuPam.getDefaultSkuCode());
       if (qty == null || qty.compareTo(lowStock) < 0) {
            warn(getLocalizer().getString("lowStockMessage", this));
       }
```

Now we have a solution which has product specific low stock setting and a shop fallback setting to display low stock message. Of course using the values in snipped you can build even more complex solution with several levels or even display a message which is an attribute of the product e.g.:

```
    final String shopLowStockAv =
getCurrentShop().getAttributeValueByCode("SHOP_LOW_STOCK_QTY");
       final double shopLowStock = NumberUtils.toDouble(shopLowStockAv);
       final Pair<String, I18NModel> lowStockAv =
product.getAttributes().getValue("LOW_STOCK_QTY");
       final BigDecimal lowStock = new
BigDecimal(NumberUtils.toDouble(lowStockAv != null ? lowStockAv.getFirst()
: shopLowStockAv, shopLowStock));
       final BigDecimal qty =
skuPam.getAvailableToSellQuantity(skuPam.getDefaultSkuCode());
       if (qty == null || qty.compareTo(lowStock) < 0) {
           final Pair<String, I18NModel> lowStockMsgAv =
product.getAttributes().getValue("LOW_STOCK_QTY_MSG");
           if (lowStockMsgAv != null) {
                 warn(lowStockMsgAv);
           }
       }
```

Similar approach can be applied on product details page and in any other components that renders product or SKU information.

REST API

REST API exposes **ProductAvailabilityModelRO** object which has all the information which is available in **ProductAvailabilityModel**. Consumers of API response can apply similar concepts in order to provide messaging to customers or alter application behaviour.

Groovy App SaaS

Groovy App relies on the same concepts as already mentioned in previous sections. Additional MO (model) abstraction level exposes **ProductAvailabilityModelMO** object.

It is recommended to work with CMS content includes in order to render messaging in page.

For example if we define an include: **category_product_low_stock_include** we can dynamically render it injecting model as an attribute like so on **ProductList.groovy**:

```
div('class': 'attr-info') {
   mkp.yieldUnescaped(ctx.dynamicContent('category_product_low_stock_inclu
de', ['product': _product]));
}
```

Above snippet includes a dynamic content include and supplies product model as an argument. The model already contains the ProductAvailabilityModelMO as a property. Thus the CMS content that displays the messaging could be:

```
div('class': 'attr-info') {
   mkp.yieldUnescaped(ctx.dynamicContent('category_product_low_stock_inclu
de', ['product': _product]));
}
```

And accompanying CMS content that will render the information:

```
<ul class="text-left">
   <%
    if (product.productAvailabilityModel.available) {

       if (product.productAvailabilityModel.inStock) {

       def _qty =
product.productAvailabilityModel.availableToSellQuantity[product.productAv
ailabilityModel.defaultSkuCode];

       if (_qty != null) {
           def _enough = _qty > 3;
           def _color = _enough ? 'success' : 'warning';
    %>
    <li><b>Stock</b> <span class="label label-pill
label-${_color}">${_qty.setScale(0,
java.math.RoundingMode.DOWN).intValueExact()}</span></li>
           <% if (!_enough) { %><li class="attr-grid-view"><span>Delivery
in 2-3days</span></li><% } %>
   <%
    } else { %>
    <li><b>Stock</b> <span class="label label-pill
label-danger">0</span></li>
    <li class="attr-grid-view"><span>Delivery in 2-3days</span></li>
   <% }

    } else { %>

    <li><b>Stock</b> <span class="label label-pill
label-danger">0</span></li>
    <li class="attr-grid-view"><span>Item is not longer
available</span></li>
   <% } %>
</ul>
```

It maybe somewhat complex snippet but it does a lot! We check if the model is available and if so we render an LI element with Stock label, exact stock quantity as bootstrap pill and then if stock is low an additional message "Delivery in 2-3days". If the SKU is not available at all we render "Item is not longer available".

All of this is done in the view template and then delegated to the content include, thus you can easily change the messaging or how it behaves from the CMS. We also do not need shop specific attributes since content element belongs to specific shop, so we can just use concrete (hardcoded) numbers like "3".