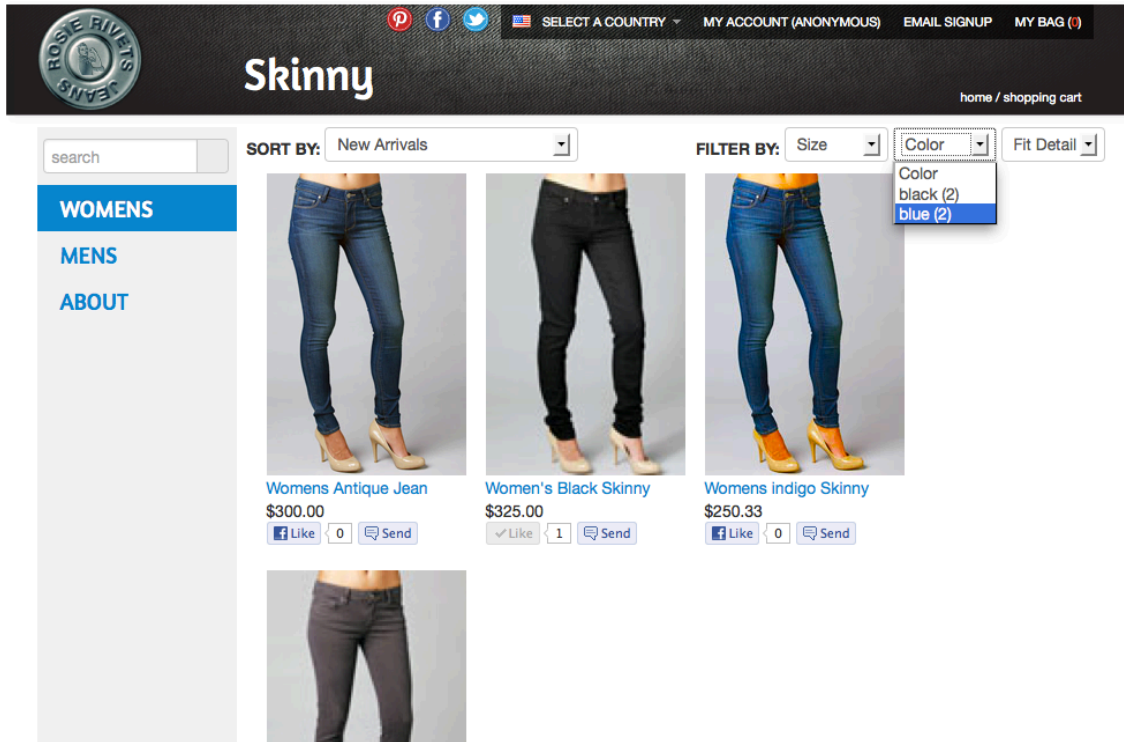


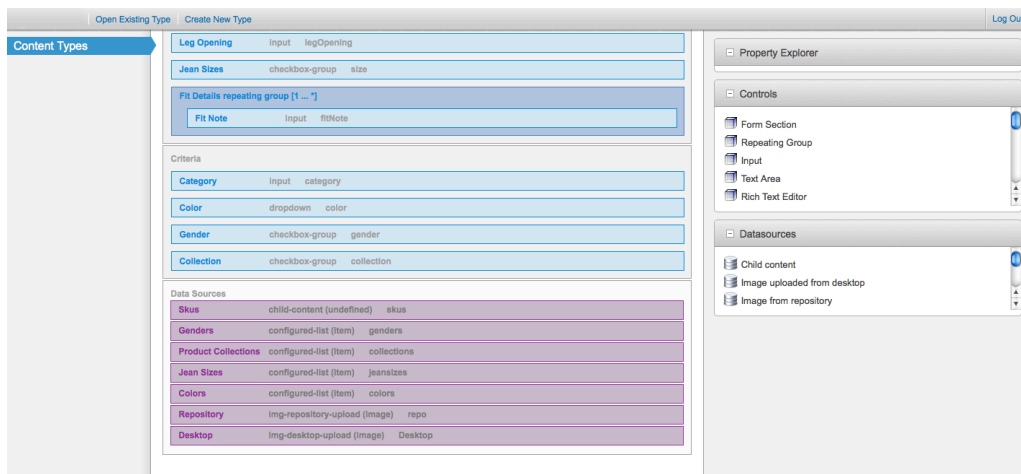
# Implementing Faceted Search

Crafter Engine, the delivery component of [Crafter CMS](#) provides powerful out-of-the-box search capabilities based on [Apache Solr](#). Solr is extremely fast and provides a wide range of capabilities that include fuzzy matching, full-text indexing of binary document formats, match highlighting, pagination, “did you mean”, typed fields and of course [faceted search](#). Faceted search (aka faceted navigation) is an ability of the search interface to break down a search in to categories that allow a user to filter and narrow down the number of results by selecting only those category values that are relevant.



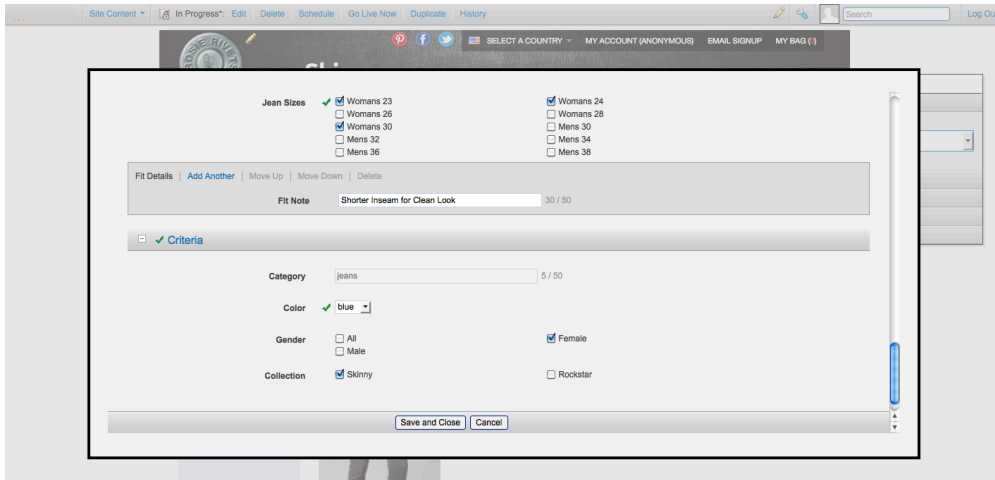
Before we get in to the construction of a faceted search let's take a quick step back and look at some basic architecture.

The first thing to think about is the type of thing we're going to be searching on. From a web content management perspective, this is often referred to as the content model. A content model in its most basic form is just the description of an entity like an article and its properties such as title, author, publish date, body and so on. In the figure above we see a search-driven UI that allows the user to narrow down a collection of jeans by size, color and fit. In order to enable this we have to “model” the jeans. These filters are criteria that must be associated with each instance of the content type. Each field (color, size, fit) has many possible values that are selected by an author when a jean object is created.



In the figure above you can see just a small portion of the Jeans product content type in the Crafter Studio drag and drop content type management tool. Note the fields for size, color and the data sources that pull values for these fields from managed taxonomies.

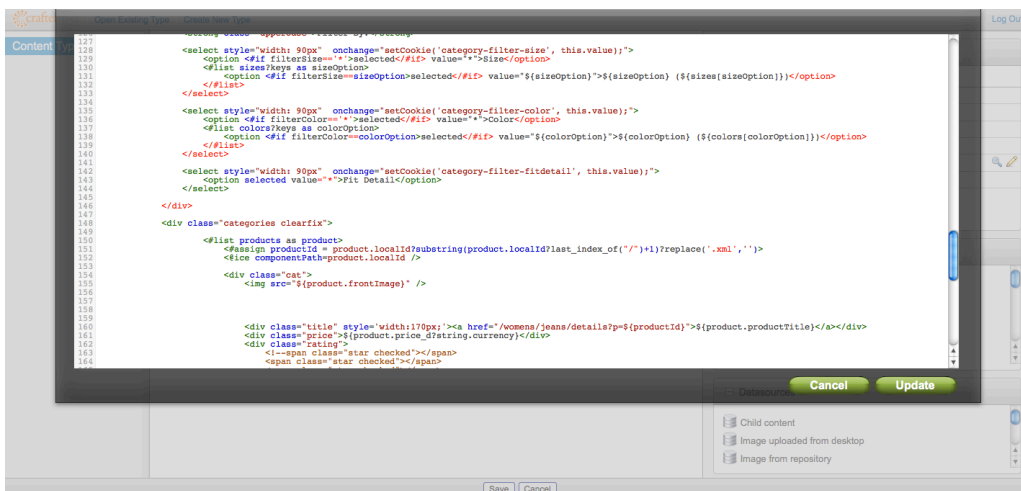
Once we've created our content type we can now create instances of jeans, provide the details for the product and select the criteria that correctly categorizes the pair of jeans.



Whenever an object is published from Crafter Studio (the content authoring environment) to Crafter Engine (the delivery platform), it is immediately indexed by Solr with the help of Crafter Engine's deployment Solr plug-in. Once published Solr is aware of each category and selected values for that category.

Now that we have content indexed in Solr we can build a search page. We're going to build the Jeans category page from the first figure. All of the coding will be done in the [Freemarker template language](#) supported by Crafter Engine. For our example we'll keep the implementation very straightforward without any abstraction. Advanced coders may choose to factor and encapsulate the code differently.

To begin, create or navigate to your category page content type (standard fields are fine) and then open the template editor. For a more in-depth tutorial on basic content modeling click [here](#).



Now that we have our template editor open and we're ready to begin coding. Let's start with a review some basic requirements.

- We need to maintain or store the user's selections for the various filters so that they persist from one search execution to another.
- We need allow the user to simultaneously filter all three categories (color, size, fit)
- We want to provide the user with a count of the number of items available for each category value
- We need to provide sorting (in our case price high to low, price low to high, and by arrival date)
- We need to provide pagination (showing n results per page)

## Maintaining the user's selection

How you choose to maintain the user's selections so that they are available across search executions is largely a function of a few factors:

- How long do the values need to persist: Only so long as the user is on the page? For the session? Whenever they visit the site?
- How sensitive is the value being stored?

- How are you refreshing the results: page reload or Ajax?

You have many options from simple JavaScript values that will be maintained only as long as the user does not leave or refresh the page to cookies, sessions and profiles each of which have their own life-cycle and security attributes.

For our example we're going to store the values in a cookie. This requires no additional configuration and persists across several visits. To do this we'll need the following code:

## Create template variables with current cookie values

As you can see, the code simply creates a template value for each user selection based on the value from the cookie. If no cookie is found a default value (specified by !"FOO") is provided. This code would typically appear close to the top of the template.

```
<#assign sort = (Cookies["category-sort"]!"")?replace("-", " ")>
<#assign filterSize = (Cookies["category-filter-size"]!"")>
<#assign filterColor = (Cookies["category-filter-color"]!"")>
```

## Render controls with values selected from cookies

Now we need to build the filter controls for our users so that they can narrow their searches. In the code below we're iterating over the available options (we'll show how these are acquired in just a moment) and creating the options for the select component. For each option we look to see if it is the currently selected item and if so we mark it as selected.

```
<select style="width: 90px"  onchange="setCookie('category-filter-color', this.value);">
  <option <#if filterColor=='*>selected</#if> value="*">Color</option>
  <#list colors?keys as colorOption>
    <option <#if filterColor==colorOption>selected</#if> value="{colorOption}">{colorOption}
  (<#list colors[colorOption]>)</option>
  </#list>
</select>
```

## Provide a mechanism to save a selected value to our cookie and force a refresh

In the code above you can see a simple JavaScript function on the "onChange" method for the select control. Again you can see here we're keeping the code as abstraction free as possible to make the example clear. Below is the simple JavaScript function:

```
<script>
  var setCookie = function(name, value) {
    document.cookie = name + "=" + value + "; path=/;";
    document.location = document.location;
    return false;
  }
</script>
```

## Building the Query and Filter Options

Now that we have a mechanism for choosing criteria it's time to use those values to create and execute a query. In the section below we'll look at how queries are built and executed through the Solr-powered Crafter Search interface.

## Construct a query that is NOT constrained by filters.

We will use the results of this query to get the possible values and counts for our filters.

Below you can see we're building up a simple query for the jeans content type, gender and collection.

```
<#assign queryStatement = 'content-type:"/component/jeans" ' />
<#assign queryStatement = queryStatement + 'AND gender.item.key:"' + gender + '" ' />
<#assign queryStatement = queryStatement + 'AND category:"' + category + '" ' />
<#assign queryStatement = queryStatement + 'AND collection.item.key:"' + collection + '" ' />
```

## Construct a query based on the first but with additional filter constraints

We will use the results of this query to display the results to the user.

```
<#assign filteredQueryStatement = queryStatement />
<#assign filteredQueryStatement = filteredQueryStatement + 'AND size.item.value:' + filterSize + ' ' />
<#assign filteredQueryStatement = filteredQueryStatement + 'AND color:' + filterColor + ' ' />
```

## Execute the unfiltered query

Here you can see we're declaring the facets we want the counts on.

```
<#assign query = searchService.createQuery()>
<#assign query = query.setQuery(queryStatement) />
<#assign query = query.addParam("facet","on") />
<#assign query = query.addParam("facet.field","size.item.value") />
<#assign query = query.addParam("facet.field","color") />
<#assign executedQuery = searchService.search(query) />
```

## Execute the filtered query

Here you can see we're declaring the pagination and sorting options.

```
<#assign filteredQuery = searchService.createQuery()>
<#assign filteredQuery = filteredQuery.setQuery(filteredQueryStatement) />
<#assign filteredQuery = filteredQuery.setStart(pageNum)>
<#assign filteredQuery = filteredQuery.setRows(productsPerPage)>

<#if sort?? && sort != "">
  <#assign filteredQuery = filteredQuery.addParam("sort","" + sort) />
</#if>

<#assign executedFilteredQuery = searchService.search(filteredQuery) />
```

## Assign the results to template variables

Below you can see the how we're getting the matching jean objects, and number of results returned from the filtered query response. You can also see how we're getting the available options and counts from the unfiltered query response.

```
<#assign productsFound = executedFilteredQuery.response.numFound>
<#assign products = executedFilteredQuery.response.documents />
<#assign sizes = executedQuery.facet_counts.facet_fields['size.item.value'] />
<#assign colors = executedQuery.facet_counts.facet_fields['color'] />
```

## Displaying the Results

### Display the products

In the code below, we're iterating over the available products and simply displaying the details for it.

```

<#list products as product>
  <#assign productId =
product.localId?substring(product.localId?last_index_of("/")+1)?replace('.xml','')>
  <@ice componentPath=product.localId />

  <div>
    
    <div style='width:170px;'><a
href="/womens/jeans/details?p={productId}">${product.productTitle}</a></div>
    <div>${product.price_d?string.currency}</div>
    <div>
      <@facebookLike contentUrl='http://www.rosiesrivets.com/womens/jeans/details?p={productId}'
width="75" faces="false" layout="button_count"/>
    </div>
  </div>
</#list>

```

## Construct pagination

Given the number of items found and our productsPerPage value we can determine the number of pages to show to the user.

```

<div>
  <ul>
    <#assign pages = (productsFound / productsPerPage)?round />
    <#if pages == 0><#assign pages = 1 /></#if>
    <#list 1..pages as count>
      <li <#if count=(pageNum+1) >class="active"</#if>><a href="{uri}?p={count}">${count}</a></li>
    </#list>
  </ul>
</div>

```

