ORACLE®

ATG WEB COMMERCE

Search

Version 10.0.2

Query Guide

Oracle ATG
One Main Street
Cambridge, MA 02142
USA

ATG Search Query Guide

Document Version

Search10.0.2 SEARCHQUERYv1 4/15/2011

Copyright

Contents

1 Introduction

The ATG Search product allows end-users to search indexed data on a site and receive useful answers. Implementing ATG Search on your site involves the following steps:

1. Specify repository items and properties to index through an XML definition file.
2. Based on the information in the XML definition file, transform these repository items into XHTML documents.
3. Index the XHTML documents in ATG Search.
4. Use search form handlers to allow site visitors to search the indexed documents.

The first three steps are described in the *ATG Search Administration Guide*. This book explains how to set up search pages and form handlers to help your users achieve the search results they want from your site. For most sites, the content searched is a product catalog; therefore most of the examples in this book are geared toward this common situation, but you can adapt them for other purposes.

The ATG form handlers provide access to a wide number of options that can affect how searches are processed and search results are displayed. These options are ultimately passed to the search engine in XML form. In general, this book provides Java API-level information regarding these settings; however, at times it may be necessary to explore the XML level, since this is the language understood by the engine. In addition, an appendix reference on the XML is included to help in debugging search request and response strings.

This chapter includes the following sections:

Audience

This guide is intended for JSP developers and system integrators. Users should have a thorough knowledge of the ATG platform, other ATG products they are planning to integrate ATG Search with, JSPs, search technology, and XML syntax.

Document Conventions

This guide uses the following conventions:

- `<ATG10dir>` refers to your ATG installation directory.
- *Server* or *ATG server* refers to an ATG application instance that is deployed to your application server in the form of an EAR file. See the *ATG Installation and Configuration Guide* for information. ATG documentation commonly refers to the following server types:
    - Asset management server—Where ATG Content Administration, Search Administration, and other such applications run. Sometimes called just the management server.
    - Agent-facing server—Used for ATG Knowledge and other agent-oriented applications
    - Customer-facing server or production server—Where your ATG application runs.

More Information

In addition to this guide, you may also want to consult the resources mentioned below.

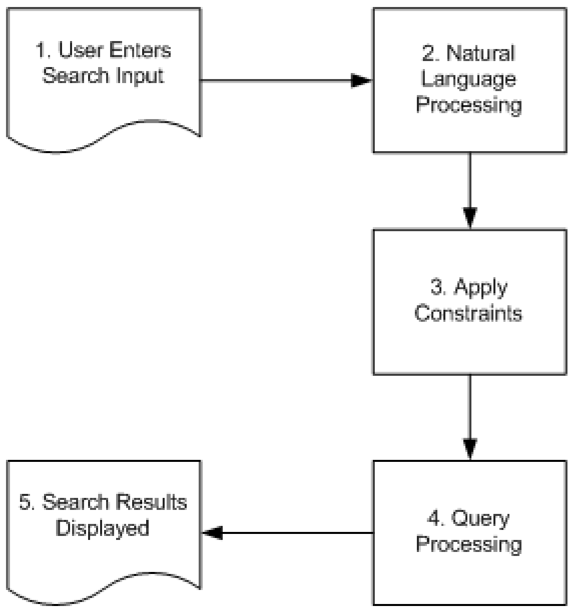
See the following documents for additional information on ATG Search and related products:

- *ATG Search Installation and Configuration Guide*
- *ATG Search Administration Guide*

2 Query Processing Overview

This chapter describes the process by which search queries are turned into responses, and highlights some of the important parameters that you might want to address in your form handlers.

This diagram provides a high-level overview of the processes used in answering search requests:


1. User Enters Search Input
2. Natural Language Processing
3. Apply Constraints
4. Query Processing
5. Search Results Displayed

Search Request Processing

The sections that follow provide detailed information on each request processing step:

User Enters Search Input

Natural Language Processing

Apply Constraints

Query Processing

Search Results Displayed

User Enters Search Input

ATG Search can accept several different types of search input. Two common types are:

- Text—The user enters one or more terms and clicks a submit button.
- Browse—The user clicks a link that contains search request information encoded in a URL. This information could include a product category.

The examples in this chapter use text searches.

The search form handlers can also add parameters to the user input, for example to identify the language the user is searching in, or to restrict the search to certain parts of the index. Most of this guide addresses this subject, but see the Search Form Handlers chapter to get started.

The text the user enters is packaged up into a query request. The request is a Java object that is sent to ATG Search when the user clicks a submit button, follows a link, etc. The request is then translated into XML for processing by the search engine.

Natural Language Processing

Once it receives the submitted request, ATG Search analyzes it to determine what terms it has to search for. To do this, it performs a series of analyses on the text the user entered. Note that it is the final result of *all* of these steps that is used by the search engine to conduct the search.

This is similar to the process used to index content, and it relies on a combination of the core search dictionary, any supplemental dictionaries Search Administration users have created, and language-specific information.

This step includes the following processes:

1. Identify the words in the request.
2. Identify the root form of each of the identified words.
3. Map variant word forms and common misspellings to a single form of the word.
4. Identify synonyms for each of the words, if any are defined.
5. Apply a weight to each of the terms in the request. This is a proprietary process by which ATG Search decides how important each term in the request is.

The final result of natural language processing is a list of terms that includes the spell-corrected root forms of the original search terms, any synonyms for these terms, and weighting information.

Apply Constraints

The next step of search processing is to apply constraints that were entered along with the request. Constraints can include the following:

- Site, catalog, category, or other property-based constraints—See the Constraining Queries chapter of this guide
- Operators such as quote marks and the ! symbol—See the User-Entered Operators chapter of this guide
- Query rules—See the *ATG Search Administration Guide*
- Search Merchandising rank configurations—See the Search Merchandising chapter of this guide

The constraints are processed and added to the query information.

Query Processing

The compiled query information is compared against the index. The engine returns XML that includes the resulting matching products or statements, taking constraints into account, along with categorization, spelling, and query feedback information specified in the request.

The results are sorted in relevance order by default, but this behavior can be changed using request parameters. The results can be grouped to reduce duplication, and you can apply paging to prevent slowing down your site with large result lists. See the Paging Search Results chapter of this guide for information.

Search Results Displayed

The final results are displayed to the user. See the Handling Results section of this guide for information.

3 Search Form Handlers

The ATG platform includes form handlers that you can use to build forms to issue queries for search data indexed by ATG Search. You can use these form handlers to build a wide variety of user interfaces to ATG Search, and to make the full power of the ATG platform available for working with ATG Search data.

This chapter discusses ATG platform and ATG Search classes and components. If you need more information about a particular class, see the *ATG API Reference*.

This chapter includes the following sections:

Form Handler Classes and Architecture

The main form handler for submitting queries to ATG Search is `atg.search.formhandlers.QueryFormHandler`. This form handler issues a search request of class `atg.search.routing.command.search.QueryRequest`, which represents a query of type `<query>`.

`QueryFormHandler` has as an ancestor the `atg.search.formhandlers.BaseSearchFormHandler` abstract class, which implements generic functionality: error handling, URL redirection, maintaining state information, query submission, etc. `QueryFormHandler` subclasses `PagedRequestFormHandler`, which is a subclass of `BaseFormHandler` that adds support for paging of results.

`QueryFormHandler` constructs search queries using the SearchClient API. This API uses an instance of Routing to direct those queries to the search engine. This instance of Routing can be running locally (on the same ATG instance as the SearchClient) or remotely (on a different ATG instance). Information about the connection to Routing is maintained in a session-scoped component of class `atg.search.formhandlers.SearchContext`. The form handler and request classes themselves are designed to be request-scoped.

QueryFormHandler has a handleSearch method that issues the search query. Typically, you associate this method with a form button, so the query is issued when the user clicks the button. For example:

```
<dsp:input type="submit" bean="QueryFormHandler.search" value=" Search "
   action="query.jsp"/>
```

QueryFormHandler includes a number of other handler methods that you may find useful in building search forms. See the *ATG API Reference* for more information.

Form Handler and Request Components

The DAF.Search.Base module includes a /atg/search/formhandlers/QueryFormHandler component and a corresponding request component, /atg/search/routing/command/search/QueryRequest. The DCS.Search.Query module adds QueryFormHandler and QueryRequest components that are preconfigured for querying the ATG Commerce product catalog. These components are found in the /atg/commerce/search/catalog Nucleus folder. You can use these components for creating search forms and displaying facets on commerce sites.

Configuring the Form Handler Component

Each instance of QueryFormHandler must be configured by setting, at a minimum, these two properties:

searchRequest
The associated request component of class atg.search.routing.command.search.QueryRequest.

searchContext
The session-scoped component (of class atg.search.formhandlers.SearchContext) that maintains information about the connection to ATG Search.

The form handler components mentioned in the previous section are preconfigured with settings for these properties. You should not need to change these values. However, if you create your own form handler components, be sure to set these properties on them.

In addition, the form handlers have properties that you can set to enable client-side processing of the request and response objects. These properties are described in Processing the Request and Response.

Configuring the QueryRequest Component

The form handler's searchRequest property points to a component of class atg.search.routing.command.search.QueryRequest. This component, which should be request-scoped, is the Java representation of the search query. The QueryRequest has a large number of properties that correspond to elements and attributes of the query XML.

Several sections of this chapter describe how to set these properties on the QueryRequest component. Note that the /atg/search/routing/command/search/QueryRequest component included in the DCS.Search.Query module has many of these properties preconfigured for querying the ATG Commerce catalog and performing faceted search.

Configuring the SearchContext Component

The form handler's `searchContext` property points to a component of class `atg.search.formhandlers.SearchContext`. This component, which should be session-scoped, maintains information about the connection to Routing. By default, the `searchContext` property of each form handler component is set to `/atg/search/formhandlers/SearchContext`.

Routing

The `SearchContext` component's `primaryConnection` property specifies the location of the `DAF.Search.Routing` module that the search client is accessing. Accessing this module on the same ATG instance as the search client is called *local Routing*; accessing it on a different ATG instance called *remote Routing*. If you are using local Routing, `primaryConnection` should be set to `local` (the default). If you are using remote Routing, set `primaryConnection` to:

```
rmi://hostname:port/atg/search/routing/SearchService
```

`port` is the RMI port used by ATG (typically 8860). For example:

```
primaryConnection=rmi://myHost1:8860/atg/search/routing/SearchService
```

For remote Routing, you can use the `SearchContext` component's `failoverConnections` property to specify an array of additional Routing instances to use if the primary instance fails. For example:

```
failoverConnections=rmi://myHost2:8860/atg/search/routing/SearchService,\
                    rmi://myHost3:8860/atg/search/routing/SearchService
```

The `maxAttemptsBeforeFailover` property specifies the maximum number of times to attempt to access a specific Routing instance before failover. The default value is 2.

The `SearchContext` component also has properties that control the firing of JMS messages when results are returned from a query. See JMS Event Handling for information.

SearchSession Object

The `SearchContext` component has a `searchSession` property that is set automatically to an object of class `atg.search.client.SearchSession`. This object is used to store information that needs to persist between search requests. For example, for paging of results, the `SearchSession` can keep track of the current page so that a subsequent request can determine what the next page is. The `SearchSession` can even store the entire previous request, which is useful for contexts (such as faceted search and some paging options) where you might want to reissue a request with only minor changes.

Sample Application

The ATG platform includes a sample Web application that demonstrates search query options. You can use the application for submitting test queries, or use the JSPs as starting points for building your own search pages.

To run the sample application, you need to assemble an EAR file that includes the `DAF.Search.Base.QueryConsole` module, and deploy this application on your application server. You can then access the application at:

```
http://hostname:port/atg/qc
```

See the *ATG Installation and Configuration Guide* for more information about the port number to use. For information about assembling applications, see the *ATG Programming Guide*.

Once the application is open in your browser, click the link for one of the query types. This opens a page that contains a form for constructing a query of that type. Fill in the form, and then click the Search button. The results of the search are displayed at the bottom of the page.

Note that neither the search forms nor the result displays are intended to be realistic examples of how you would use the form handlers on an actual site. The forms enable you set search attributes that are rarely set by individual queries. In addition, the forms allow you to set attributes directly that would typically be determined by logic implemented in the page or in methods of the form handler itself. For the results, each page simply displays a table of the fields of the results object and their values. On an actual site, the results would typically be displayed in a more usable (and selective) format.

Specifying the Content Labels and Target Type for Queries

ATG Search is organized by projects, as discussed in the *ATG Search Administration Guide*. There are two main types of relationships between projects:

- Independent projects have separate groupings of content for indexing. Two or more independent projects can index the same content or different content, but the indexes are totally separate and independent. A project's content sets are grouped and identified for querying by one or more *content labels*.
- Linked projects have identical content labels, but they differ by *target type*, which specifies the instance of the index that queries are directed to. In a set of linked projects, there is a main project whose target type is Production. Other projects are created as links to this project, and have different target types, such as Staging or Testing.

Each search project produces an index that comprises one or more logical partitions. The index has a separate logical partition for each content set in the project.

Understanding Content Labels and Target Types

Content labels are used in search projects as a way of grouping content sets for querying. A content label can refer to one or more content sets. So, for example:

```
Catalog content label --> Products content set
Articles content label --> Brochures content set, Manuals content set
```

Each content label must be unique to a single set of linked projects. Two independent projects cannot use the same content label (though they can use the same content and have different content labels to refer to it), because content labels are used to differentiate independent projects for querying.

Linked projects, on the other hand, *must* share content labels. Linked projects differ only in the target type (Production, Staging, etc.). Each target type must be unique to one project within a set of linked projects. Two linked projects cannot use the same target type, because the target type is used to determine which search environment within a set of linked projects to direct queries to.

To specify the content labels and target type for a query, you set the `contentLabels` and `targetType` properties on the `QueryRequest` component in a properties file or JSP. For example:

```
contentLabels=Catalog,Articles
targetType=Staging
```

By default, the `/atg/commerce/search/catalog/QueryRequest` component has the following configuration:

```
contentLabels=Catalog
targetType=Production
```

Site-Specific Content

In a multisite environment, you associate specific sites with specific content sets as a way of constraining indexing and querying. For example, suppose you have three sites, A, B, and C, and a content set that you want to make available for searching on sites A and C, but not on site B. To do this, you could associate sites A and C with this content set in ATG Site Administration. When the index is generated, the logical partition associated with this content set does not include data for site B.

When a query is issued, rather than directing it to all of the logical partitions of the index, Routing looks at the sites the query applies to, determines which content sets are associated with those sites, and directs the query only to the partitions for those content sets. So in this example, if a customer is searching site B, Routing will not direct the query to the logical partition for this content set. This restriction makes searching more efficient, because it reduces the number of logical partitions that need to be searched.

There may be times when you do not want to return data for a specific site, even though that data is included in the index. For example, if a site is disabled or otherwise unavailable, it is generally undesirable to return search results for that site. Therefore, ATG Search does not return results for sites that are currently unavailable.

Determining the Environment to Search

When a search request is submitted, Routing uses the `/atg/search/routing/DynamicTargetGenerator` component to determine the search environment, the set of logical partitions, and possible constraints for the request. The `DynamicTargetGenerator` takes into account the following properties of the `QueryRequest` component:

- `contentLabels` -- An array of the content labels to direct the request to. These content labels must all be associated with the same search project.
- `targetType` -- The target type to direct the request to.

- `dynamicTargetSpecifier` -- A component of class `atg.search.routing.command.search.DynamicTargetSpecifier`, which is used to specify site information; by default, `/atg/search/routing/command/search/DynamicTargetSpecifier`. If this property is not set, a `DynamicTargetSpecifier` is created automatically.
- `siteConstraints` -- A component of class `atg.search.routing.command.search.MultisiteConstraint`; by default, `/atg/search/routing/command/search/MultisiteConstraint`.

`DynamicTargetSpecifier` and `MultisiteConstraint` are described below.

DynamicTargetSpecifier

The `DynamicTargetSpecifier` component specifies the sites that the query should be applied to. This component has two main ways to specify the sites:

- To explicitly specify the sites, set the `siteIdsArray` property to an array of site IDs.
- If `siteIdsArray` is not set, `DynamicTargetSpecifier` uses the current site from the `SiteContext` object. In addition, if the `useContextAdditionalSites` property is `true`, `DynamicTargetSpecifier` also uses the sites specified in the `SiteContext` object's `additionalSites` property.

`DynamicTargetSpecifier` also has its own `additionalSites` property for specifying sites in addition to the ones specified in `siteIdsArray` or obtained from the `SiteContext` object.

MultisiteConstraint

Based on the properties set on the `QueryRequest` and the `DynamicTargetSpecifier`, the `DynamicTargetGenerator` may determine that the query should return results only from certain sites. If so, it can generate a constraint and set the value of the `MultisiteConstraint` component's `siteConstraints` property to the generated constraint. Note that `DynamicTargetGenerator` generates a site constraint only if necessary; in some cases, sites may already be excluded based on the content sets being searched, so they don't need to be excluded through a constraint.

`MultisiteConstraint` has a `siteConstraintsOperation` property for specifying whether the sites in the generated constraint should be combined using AND or OR Boolean operators. The default value of this property is `or`, so sites are combined using Boolean OR. This means that results will be returned from any site listed in the constraint. For example, this constraint means "return results that are associated with site A *or* site C":

```
<or>
  <strprop name="$siteId">A</strprop>
  <strprop name="$siteId">C</strprop>
</or>
```

If you set `siteConstraintsOperation` to `and`, the query returns only results that are associated with both site A *and* site C.

In addition to the generated site constraint, `MultisiteConstraint` has an `additionalConstraints` property that can be set to a `DocumentSetConstraint` component, which you can use to specify additional site constraints. For example, you could create a constraint that restricts unstructured content such as articles from being returned on a certain site.

`MultisiteConstraint` has an `additionalConstraintsOperation` property for specifying whether the generated constraint and the constraint specified through `additionalConstraints` should be combined using AND or OR Boolean operators. The default for this property is `and`.

Setting Request Properties

The `QueryRequest` class has properties that store the values that are used to construct the queries sent to ATG Search. Typically, these properties are set in JSPs (or have defaults set in properties files that can then optionally be overridden in JSPs), as their values may vary from query to query. Most of the properties of the request components correspond to attributes of the query XML understood by the search engine.

The XML query attributes are described in detail in the *ATG Search Query Reference Guide*. You can also find information about these attributes in the Javadoc for the request classes, in the *ATG API Reference*. This section describes how to set some of the more complex query attributes.

Setting the responseNumberSettings Property

The <`query`> query type has a `responseNumberSettings` attribute that comprises many subattributes. In the query XML, this attribute is a single long delimited String that encodes the values of all of the subattributes.

In the `QueryRequest` class, the corresponding `responseNumberSettings` property is a Map that stores these subattributes as a series of key/value pairs. This makes it possible to set the values of these subattributes individually. When the query is submitted, the values in the `responseNumberSettings` Map property are used to construct the String used in the query XML.

You can specify the values of all or some of the `responseNumberSettings` subattributes in a properties file. For example:

```
responseNumberSettings=\
     prop=10,\
     doc=100
```

You can also set these values individually in JSPs, overriding the values in the properties file. For example:

```
<dsp:input bean="${FH}.searchRequest.responseNumberSettings.doc"
   value="125"/>
```

Setting the relQuestSettings Property

The `<query>` query type has a `relQuestSettings` attribute that, like `responseNumberSettings`, is a delimited String that encodes the values of a number of subattributes.

In the `QueryRequest` class, the corresponding `relQuestSettings` property is an `atg.nucleus.ResolvingMap` that stores these subattributes as a series of key/value pairs. (The `atg.nucleus.ResolvingMap` class is a special type of Map that allows you to use the Nucleus `^=` syntax to link the value of an individual map key to the value of a property of another component.) When the query is submitted, the values in the `relQuestSettings` property are used to construct the String used in the query XML.

Setting activeSolutionsZones

The `activeSolutionZones` subattribute of `relQuestSettings` is a String containing a comma-separated list of the text properties in the index that are available for searching. Typically, you'll want to set this to a list of all of the text properties in the index (after all, that's why you included them). Rather than specifying the entire list explicitly, you can make all of the text properties available for searching by setting `activeSolutionZones` to an asterisk (*). For example, you might set this in the `QueryRequest.properties` file:

```
relQuestSettings=\
    activeSolutionZones=*
```

In some cases, however, you may want to make only a subset of the indexed text properties available for searching. For example, suppose your site has a Books section, and when customers search in this area, you want to search only the `title` and `author` properties. Rather than having two different indexes with different properties in them, you create a single index with the full set of desired properties for the overall site, but then override the value of `activeSolutionZones` in the pages of the Books section to restrict searches from those pages:

```
<dsp:input bean="${FH}.searchRequest.relQuestSettings.activeSolutionZones"
   value="role:author,role:title"/>
```

When a customer enters a search query in this part of the site, ATG Search searches only these two properties. The search is faster, returning fewer but more relevant results.

Setting the docProps and textProps Properties

The `QueryRequest` class has `docProps` and `textProps` properties that specify lists of the metadata and text properties to include in the response. If a property is specified in one of these lists, its values are included in the search results that are returned. This simplifies displaying these values in JSPs, since otherwise you'd need to retrieve them from the indexed repository.

The `textProps` property is an array listing the text properties to return. The names must include the `role:` prefix. For example, you might set `textProps` like this:

```
textProps+=role:longDescription,role:parentCategory.displayName
```

The docProps property is an array listing the metadata properties to return. For example, you might set docProps like this:

```
docProps+=childSKUs.color,childSKUs.salePrice
```

If you want to return all of the metadata properties in the index, you can set docProps to the special keyword all, rather than specifying the properties individually. For example:

```
docProps=all
```

Setting Grouping Options

If you index your product catalog by SKU, each result returned from a query represents an individual SKU. This means some products may appear multiple times, because the query may match multiple SKUs for an individual product.

If you want to return each product only once, you can group the results by product ID. ATG Search will return a single result for each unique product ID.

To enable grouping, you set the sorting property of the QueryRequest component to property (to enable grouping by property), and set the sortProperty to the index property to group by. For example:

```
sorting=property
sortProperty=string:$repositoryId:1
```

Note that in spite of their names, these QueryRequest properties control grouping, not sorting.

The format for the value of sortProperty is:

data-type:*index-property*:*default-value*

In the example above, the property is $repositoryId, which means the repository ID of the top-level item type in the IndexingOutputConfig definition file. The top-level item type is the product item, so the product ID is used as the grouping property. If a product ID value cannot be found for an item in the index, the default value (in this example, 1) is used for that item instead.

If you are using faceted search, you should also set the refineCount property to group, so the refinement counts reflect the number of products in each facet selection, rather than the number of SKUs:

```
refineCount=group
```

For more information about refinement counts in faceted search, see About Refinement Counts.

Setting the Parser Options

The QueryRequest class has a parserOptions property for specifying natural language processing options. These options are encoded as XML in the search request. To simplify the coding of your pages, the parserOptions property is set to a component of class atg.search.routing.command.search.ParserOptions; rather than constructing the XML directly, you specify the options by setting properties on this component, in either a properties file or a JSP. When a query is issued, the ParserOptions component takes care of constructing the XML and including it in the input sent to ATG Search.

The following is an example of setting a property of the ParserOptions component in a JSP:

```
<p>Maximum expansions: <dsp:input type="text"
    bean="${FH}.searchRequest.parserOptions.wildcardMax"/>
```

You can also specify the XML directly by setting the xml property of the ParserOptions component. This is useful if you want to initialize the XML in a properties file. For example:

```
xml=<parserOptions><language>en_US</language></parserOptions>
```

For information about the natural language processing options and their possible values, see the *ATG Search Query Reference Guide*.

Setting Constraint Properties

The QueryRequest object has several properties that point to components of class atg.search.routing.command.search.DocumentSetConstraint (or one of its subclasses). These components represent query constraints, which are encoded as XML in the search request. The XML can be quite complex , often with many levels of nested tags.

For example, suppose you want to specify the following constraint through the QueryRequest.documentSetConstraints property:

```
<and>
  <strprop op="equal" name="childSKU.color">
    rose
  </strprop>
  <numprop op="lesseq" name="childSKU.price">
    30.00
  </numprop>
</and>
```

This constraint limits search results to products whose childSKU.color property has a value of rose and whose childSKU.price property has a value less than or equal to 30.00. (These properties may have multiple values, since products can have multiple SKUs, but as long as one of the values of childSKU.color is rose and one of the values of childSKU.price is less than or equal to 30.00, the constraint is satisfied.)

To specify this constraint, you could create a component of class atg.search.routing.command.search.XmlDocumentSetConstraint and set its xmlConstraint property to:

```
<and>\
  <strprop op="equal" name="childSKU.color">\
     rose\
  </strprop>\
  <numprop op="lesseq" name="childSKU.price">\
     30.00\
  </numprop>\
</and>
```

Then set `QueryRequest.documentSetConstraints` to this component. For example.:

```
documentSetConstraints=/MyStuff/MyConstraints
```

Assembling the Constraint using Components

Specifying the XML directly is useful for initializing constraints, but it is not very flexible, because it doesn't allow you to modify the XML dynamically in JSPs. Therefore, the ATG platform also includes a number of subclasses of `DocumentSetConstraint` that allow you to specify elements of the XML as individual properties. When a query is issued, these constraint classes take care of constructing the XML and including it in the input sent to ATG Search.

To construct the constraint shown above, you'd create components of the `DocumentSetConstraint` subclasses `StringConstraint`, `NumericConstraint`, and `ConstraintsGroup`, from the `atg.search.routing.command.search` package.

To specify the first part of the constraint, create a `StringConstraint` component named `/MyStuff/MyStringConstraint` and set these properties:

```
property=childSKU.color
operation=equal
value=rose
```

To specify the second part, create a `NumericConstraint` component named `/MyStuff/MyNumericConstraint` and set these properties:

```
property=childSKU.price
operation=lesseq
value1=30
```

Now combine the two parts by creating a `ConstraintsGroup` component named `MyConstraintsGroup` and setting these properties:

```
operation=and
constraints=/MyStuff/MyStringConstraint,/MyStuff/MyNumericConstraint
```

Set the `QueryRequest.documentSetConstraints` property to `MyConstraintsGroup`:

```
documentSetConstraints=/MyStuff/MyConstraintsGroup
```

This approach makes it possible to modify the constraints dynamically in JSPs. For example:

```
<dsp:select bean="/MyStuff/MyStringConstraint.value">
  <dsp:option value="rose">rose</dsp:option>
  <dsp:option value="mauve">mauve</dsp:option>
  <dsp:option value="gray">gray</dsp:option>
  <dsp:option value="purple">purple</dsp:option>
  <dsp:option value="cobalt">cobalt</dsp:option>
</dsp:select>
```

Processing the Request and Response

The form handlers include three properties for specifying components for preprocessing the search request and postprocessing the response:

searchRequestProcessors
An array of Nucleus components for preprocessing the search request before it is submitted. Each component in the array must be of a class that implements the `atg.search.formhandlers.SearchRequestProcessor` interface.

searchResponseProcessors
An array of Nucleus components for postprocessing the search response after it is received. Each component in the array must be of a class that implements the `atg.search.formhandlers.SearchResponseProcessor` interface.

searchRedirectProcessors
An array of Nucleus components for determining the URL to redirect to, depending on the search results. Each component in the array must be of a class that implements the `atg.search.formhandlers.SearchRedirectProcessor` interface.

Note that the `/atg/commerce/search/catalog/QueryFormHandler` component is configured to use the `/atg/search/repository/FacetSearchTools` component as a request preprocessor and postprocessor. This component is of class `atg.commerce.search.refinement.custom.CustomCatalogFacetSearchTools`, which implements all three interfaces listed above.

Catalog Constraints

As mentioned above, `/atg/commerce/search/catalog/QueryFormHandler` component is configured to use the `/atg/search/repository/FacetSearchTools` component as a request preprocessor and postprocessor. In addition to faceted search-related processing (described in Issuing Faceted Search Queries), the `FacetSearchTools` component manages catalog constraints for all queries, not just faceted search queries.

By default, when a user enters a search query, `FacetSearchTools` checks the user's profile to see if he or she is assigned a catalog. If so, `FacetSearchTools` constrains the search to include only items in this catalog.

This behavior may not be desirable in certain situations. In a multisite environment that has a separate catalog for each site, this logic may interfere with cross-site searches, because the search will be constrained to a single catalog (and thus a single site). To prevent applying a constraint based on the catalog assigned to the user, set the `queryByCatalog` property of the `FacetSearchTools` component to `false`.

If `queryByCatalog` is `false`, you can still have `FacetSearchTools` constrain queries to specific catalogs by setting its `catalogIds` property to an array of the catalog IDs of the catalogs to include in the search. For example:

```
catalogIds=masterCatalog,homeStoreCatalog
```

Note that if `catalogIds` is not null, the value of `queryByCatalog` is ignored. In this case, even if `queryByCatalog` is `true`, the catalog assigned to the user is ignored and the catalogs in the `catalogIds` property are used for the catalog constraint.

Specifying a Price List in the Search Request

If your ATG Commerce catalog uses price lists, a single item may have multiple prices. A customer is assigned a price list, and when that customer accesses a product or SKU, ATG Commerce looks up the item's price in that price list. Another customer looking at that same item might see a different price, if that customer is assigned a different price list.

The *Indexing Price Data in Price Lists* section of the *ATG Search Administration Guide* describes how to create a special `price` property to represent the price data stored in price lists, using the `/atg/commerce/search/PriceListPropertyProvider` component. When you index your catalog, the item prices are read from the price lists and used to construct `meta` tags in the XHTML documents. A separate `meta` tag is created for each price list, and the property name in the tag identifies the price list the tag is associated with.

When the index is queried, the search client must determine which price list is assigned to the customer issuing the query, and map the `price` property to the `meta` tag associated with that price list. To do this, the search client determines which user profile properties identify the assigned price lists, and checks the value of those properties to determine the price lists to use. It then creates a property mapping between the `price` property in the indexing definition file and the price properties in the index associated with those price lists.

Creating the Property Mapping

When a search query is issued, the `/atg/commerce/search/PriceListPropertyMapping` component creates the property mapping dynamically, based on the user's profile and other settings that you specify through the following properties of the component:

price
The name of the price property created in the `PriceListPropertyProvider` component. The default is `price`.

priceLists
An ordered array of the profile properties that specify price lists assigned to the user. The order of the properties determines the order of precedence when querying the search index. The default is `priceList`.

pricePropertyPrefix
The prefix for the price property in the XHTML document. This must match the prefix specified in the `pricePropertyPrefix` property of the `PriceListMapPropertyAccessor` component. The default is `childSKUs.price@`.

The `PriceListPropertyMapping` component constructs the `propertyMapping` tag and sets the value of the `PriceListPropertyMapping` .`priceMapping` property to this tag. For example, `PriceListPropertyMapping` might set `priceMapping` to:

```
<propertyMapping>
  price,childSKUs.price@pl90002,childSKUs.price@pl90004
</propertyMapping>
```

To include the `propertyMapping` tag in the search request, set the `propertyMappings` property of the `/atg/commerce/search/ProductCatalogParserOptions` component by linking to the value of the `PriceListPropertyMapping` .`priceMapping` property:

```
propertyMappings^=\
   /atg/commerce/search/PriceListPropertyMapping.priceMapping
```

This maps the `price` property in the query to the `childSKUs.price@pl90002` and `childSKUs.price@pl90004` properties in the index. The index properties are listed in order of precedence. When the query is executed, ATG Search looks for the `childSKUs.price@pl90002` property for an item. If there is a property by that name, ATG Search uses the value of that property. If there is no `childSKUs.price@pl90002` property (which means that price list `pl90002` does not have a price for the item), it uses the price from the `childSKUs.price@pl90004` property. So for example, if `pl90002` contains sale prices and `pl90004` contains list prices, ATG Search uses the sale price if there is one, and otherwise uses the list price.

Specifying a Default Price List

If a user is not assigned a price list, constraints and facets based on price will not work properly for that user. This is generally not an issue for actual site customers, who should all have price lists assigned to them in their profiles, but it may affect internal users testing the site.

You can specify a default price list for users who are not assigned price lists in their profiles. To do this, set the `defaultPriceListId` property of the `PriceListPropertyMapping` component to the price list ID of the price list you want to use as the default. For example:

```
defaultPrice=pl90002
```

Invoking Multiple Form Handlers

Some sites may need to make multiple repositories available for searching. For example, suppose your site has, in addition to a product catalog, a repository with informational articles about the products. When a user searches for, say, mountain bikes, you want to return both products and articles. You could enable this by creating a single index that includes items from both repositories, and returning both types of items in the same set of results.

If, however, you want products and articles to appear as separate sets of results, displayed on different areas of the page, then including them in the same index is not desirable. Instead, you can create separate indexes for the two repositories, and use two form handler instances that submit the same query text simultaneously to the two indexes.

To invoke multiple form handlers simultaneously, you use a component of class `atg.search.formhandlers.MultipleSubmitHelper`. This is a special form handler class that invokes multiple `QueryFormHandler` components, using the same query text for all of them.

You configure a `MultipleSubmitHelper` component by setting its `queryFormHandlers` property to an array of the `QueryFormHander` components you want to invoke. For example, the `/atg/search/formhandlers/MultipleSubmitHelper` component (used by the `QueryConsole` sample application) is configured like this:

```
queryFormHandlers=/atg/search/formhandlers/QueryFormHandler1,\
          /atg/search/formhandlers/QueryFormHandler2
```

In your JSP, you create the search form as shown in the following example:

```
<%-- create the query text input field and associate it with the --%>
<%-- handleQuestion method of the MultipleSubmitHelper component --%>
<dsp:input type="text" id="question" size="30" converter="nullable"
           name="question" bean="MultipleSubmitHelper.question"/>

<%-- create the submit button and associate it with the handleSearch --%>
<%-- method of one of the QueryFormHandler components --%>
<dsp:input type="submit" bean="QueryFormHandler1.search" value=" Search "
           action="multiquerysubmit.jsp" priority ="-100"/>

<%-- create a hidden field on the other QueryFormHandler components --%>
<dsp:input type="hidden" bean="QueryFormHandler2.search" value=" Search "
           action="multiquerysubmit.jsp" priority ="-100"/>
```

Handling Results

The `QueryRequest` class has an inner `Response` class that stores the results returned for the query. The value of the `searchResponse` property of the form handler that issued the query is set to this `Response` object, so you can display the query results in JSPs. For example, the following JSP fragment creates a table that displays several properties of each category in the `suggestedCategories` List property of the `Response` object:

```
<dsp:getvalueof bean="QueryFormHandler.searchResponse" var="queryResponse"
  scope="request"/>

<table class=data style="width:100%" border=1 cellPadding=5 cellSpacing=0>
  <tr class="alt">
    <td>path</td>
    <td>id</td>
    <td>score</td>
    <td>label</td>
  </tr>
  <c:forEach items="${queryResponse.suggestedCategories}" var="cat">
    <tr>
      <td><c:out value="${cat.path}"/></td>
      <td><c:out value="${cat.id}"/></td>
      <td><c:out value="${cat.score}"/></td>
      <td><c:out value="${cat.label}"/></td>
    </tr>
  </c:forEach>
</table>
```

Notice that most of the JSP tags in this example are standard JSTL tags, rather than DSP tags, because the `Response` object is not a Nucleus component.

Indicating the Number of Results

You can indicate the number of results returned by rendering the value of the search response object's `groupCount` property. For example:

```
<c:out value="${queryResponse.groupCount}"/>
```

Note that this value is not affected by paging. It represents the total number of result groups returned by the search, regardless of the number on any given page. Note that if you are grouping results by property (see Setting Grouping Options), this number may be smaller than the number of documents returned, since a group can contain multiple documents.

Handling Repository Items

If the documents being searched represent repository items (such as products or SKUs in a product catalog), each result returned will contain the URL for the corresponding item. You can use the URL to display the repository item. For example:

```
<dsp:getvalueof bean="QueryFormHandler.searchResponse" var="queryResponse"
  scope="request"/>
```

```
<c:forEach items="${queryResponse.results}" var="result">
  <c:out value="${result.document.url}"/>

  <dsp:droplet name="/atg/targeting/RepositoryLookup">
     <dsp:param name="url" param="${result.document.url}"/>
     <dsp:oparam name="output">
       <dsp:valueof param="element.displayName"/>
     </dsp:oparam>
  </dsp:droplet>
</c:forEach>
```

Note, however, these URLs will not be recognized unless the repository is registered. To register a repository, add it to the list of repositories in the `initialRepositories` property of the `/atg/registry/ContentRepositories` component.

4 Paging Search Results

An individual search query can return a large number of results. Rather than displaying all of the results on a single page, you will typically want to break them up into multiple pages, with a certain number of items per page. Therefore, the query includes properties that you can use to specify the number of items to include per page and which page to display.

This chapter describes these pagination properties and their effects. It includes the following topics:

Specifying the Page Size

You use the `pageSize` request attributes to specify the number of items per page. For example, if `pageSize=10`, the results will include ten items per page.

The following JSP fragment creates a drop-down for selecting a value for the `pageSize` attribute:

```
Page Size
<dsp:select bean="${FH}.searchRequest.pageSize">
  <dsp:option value="100">100</dsp:option>
  <c:forEach begin="5" end="35" step="5" var="pageSize">
    <dsp:option value="${pageSize}">
      <c:out value="${pageSize}"/>
    </dsp:option>
  </c:forEach>
</dsp:select>
```

Handling Page Requests

When you render the initial page of results, you typically want to render links to other pages of results. Each of these links actually issues a new search query that in most respects is identical to the original query, but which specifies a different page of results. So paging involves a sequence of connected requests and responses.

To display a specific page of results, you set the form handler's `goToPage` property to a 1-based page number. The `goToPage` property has an associated handler method, `handleGoToPage`. When a user clicks a link that sets the value of `goToPage`, the `handleGoToPage` method issues the search for the specified page.

To ensure that all of the requests and responses in a sequence of page requests are associated with each other, the initial query generates a unique String identifier called a *request chain token*. This identifier is included in each query in the sequence of requests, and is returned in each response.

There are various paging options available, and the ones you use depend on the needs or your site. The following options are explained below:

- The type of paging to use: normal paging or fast paging
- Whether to save the request in the search session or not

Types of Paging

ATG Search supports two types of paging, normal paging and fast paging. The key differences between them relate to the information you get back from the search engine about the number of pages of results, and the navigation you can build into your pages:

- Normal paging is the default. In this mode, the search engine returns (in the form handler's `pagesAvailable` property) the total number of pages of results. You can create links that enable the customer to go directly to any page.
- Fast paging is specified by setting the `fastPaging` property of the search request to `true`. In this mode, the search engine does not return information about the total number of pages of results; `pagesAvailable` is set to the highest-numbered page that has been rendered so far. You can enable customers to go to the next page or to any page previously rendered.

For a single-partition index, normal paging is always enabled (the `fastPaging` property is ignored). For a multi-partition index, you can choose between normal paging and fast paging, but fast paging is recommended. Fast paging is much less resource-intensive than normal paging. On multi-partition indexes, normal paging can be very memory- and CPU-intensive, because results from the partitions must be merged.

Example of Normal Paging

The following example renders a list of the page numbers of all pages of results. Each page number is a link to the corresponding results page, except for the current page number, which is displayed without a link. For example, if `pagesAvailable` is 43 and the current page is page 10, this code will render the integers from 1 to 43, and all of these except 10 will be links. If the user clicks 22, for example, a request to display page 22 will be issued.

```
<!-- Indicate that the request should be saved in the search
     session so that initial request data, such as the question
     text, is available to subseqent paged requests. -->
<dsp:input bean="QueryFormHandler.searchRequest.saveRequest"
           value="true" type="hidden"/>

<!-- Display page numbers with links to take user to specified
     page -->
Go to Page:
<c:forEach var="page" begin="1" end="${formHandler.pagesAvailable}">
  <c:choose>
    <c:when test="${page == (1+formHandler.searchResponse.pageNum)}">
      ${page} <!-- The current page, don't display a link -->
    </c:when>
    <c:otherwise>
      <dsp:a href="normal-paging.jsp">
        ${page}
        <dsp:property bean="QueryFormHandler.searchRequest.requestChainToken"
                      value="${formHandler.searchResponse.requestChainToken}"/>
        <dsp:property bean="QueryFormHandler.searchRequest.saveRequest"
                      value="true"/>
        <dsp:property bean="QueryFormHandler.goToPage" value="${page}"/>
      </dsp:a>
    </c:otherwise>
  </c:choose>
</c:forEach>
```

Example of Fast Paging

The following example renders a list of the page numbers of the results pages that have been rendered so far. Each page number is a link to the corresponding results page, except for the current page number, which is displayed without a link. In addition, the example renders the word “more” as a link to the page following the current one.

```
<!-- Turn on fast paging -->
<dsp:input type="hidden" value="true"
           bean="QueryFormHandler.searchRequest.fastPaging"/>

<!-- Indicate that the request should be saved in the search
     session so that initial request data, such as the question
     text, is available to subseqent paged requests. -->
<dsp:input bean="QueryFormHandler.searchRequest.saveRequest"
           value="true" type="hidden"/>

<!-- Shortcut to the response object, which may be null -->
<c:set var="response" value="${formHandler.searchResponse}"/>
```

```
<!-- Display page numbers with links to take user to specified
     page -->
<c:if test="${response != null}">
  on page: ${1+response.pageNum}<br/>
  Go to Page:
  <c:forEach var="page" begin="1" end="${1+formHandler.pagesAvailable}">
    <c:choose>
      <c:when test="${page == (1+response.pageNum)}">
        ${page} <!-- current page -->
      </c:when>
      <c:otherwise>
        <dsp:a href="fast-paging.jsp">
          <c:choose>
            <c:when test="${page == (1+formHandler.pagesAvailable) &&
                            response.multiPartitionSearch &&
                            formHandler.searchRequest.fastPaging}">
              more
            </c:when>
            <c:otherwise>
              ${page}
            </c:otherwise>
          </c:choose>
          <dsp:property bean="QueryFormHandler.searchRequest.requestChainToken"
                        value="${formHandler.searchResponse.requestChainToken}"/>
          <dsp:property bean="QueryFormHandler.searchRequest.saveRequest"
                        value="true"/>
          <dsp:property bean="QueryFormHandler.goToPage" value="${page}"/>
        </dsp:a>
      </c:otherwise>
    </c:choose>
  </c:forEach>
</c:if>
```

Modifying and Resubmitting the Request

Since subsequent requests differ only in the requested page of results, it is most efficient just to retrieve the most recent search request, change the value of the `goToPage` property, and resubmit the request. There are two ways to do this:

- Modify properties on the form, and resubmit it. This avoids the memory use required to save the request in the `SearchSession`. The downside is that resubmitting the form is difficult if you are creating your links through anchor tags. In that case, it is generally easiest to write a JavaScript function that makes the necessary changes and submits the form.
- Save the request in the `SearchSession`. This allows you to retrieve the request, modify it, and reissue it; no JavaScript is necessary. The downside is that this approach

can use a lot of memory, especially if there are many users at your site issuing search queries.

Note that resubmitting a modified request is useful for faceted search as well as for paging.

Example of Resubmitting the Form

If you do not want to save the request in the `SearchSession`, you will need to resubmit the form. Create a JavaScript function like this:

```
function nextPage(pageNum, requestChainToken)
{
  document.searchForm.requestChainToken.value = requestChainToken;
  document.searchForm.goToPage.value = pageNum;
  document.searchForm.submit();
  return false;
}
```

You can then invoke the function when the user clicks on a link for a specific page:

```
<a href="#" onclick="return nextPage('<%=pageValue.toString()%>',
 '${formHandler.searchResponse.requestChainToken}');">
  <dsp:valueof param="count"/>
</a>
```

When the link is clicked, the page number associated with the link and the `requestChainToken` of the current search response are passed to the function. The function uses these values to set the `goToPage` property and the `requestChainToken` property of the form, which it then submits. In addition to specifying the results page to display, this ensures that the same `requestChainToken` value is associated with each subsequent search request.

Example of Saving the Request in the SearchSession

If you save the request in the `SearchSession`, you can avoid the use of JavaScript. Instead, when a user clicks on a link for a page, you set the necessary properties (including the `saveRequest` property) on the saved request through `dsp:property` tags, and then resubmit the request:

```
<dsp:a href="queryExampleFastSave.jsp#Paging">
  <dsp:valueof param="count"/>
  <dsp:property bean="QueryFormHandler.goToPage" paramvalue="count"
    name="fh_gtp" priority="29"/>
  <dsp:property bean="QueryFormHandler.searchRequest.saveRequest"
    value="true" name="fh_sr" priority="30"/>
  <dsp:property
    bean="QueryFormHandler.searchRequest.requestChainToken"
    value="${formHandler.searchResponse.requestChainToken}"
    name="fh_rct" priority="30"/>
</dsp:a>
```

The examples of normal paging and fast paging in the Types of Paging section use this approach.

5 Implementing Type-Ahead for Searches

ATG Search supports a type-ahead feature that attempts to automatically complete the query text as the user enters it in a search form. As the user types in the text box, a drop-down list is displayed listing possible query text that matches the characters entered. For example, if the user types “bla”, the dropdown may display “black”, “bland”, and “blank”. If the user then types “n”, the dropdown now displays only “bland” and “blank”, because the typed substring no longer matches “black”. At any point, the user can select a text string from the dropdown, and it will be entered in the text box. The user can then click the submit button to issue a query with the selected text, or can ignore the dropdown and type in the complete text to search for.

Implementing type-ahead involves creating two pages:

- A type-ahead page for submitting type-ahead queries and rendering the returned autocomplete strings.
- A search page containing an instance of `QueryFormHandler`, which is used to construct the search form and issue queries against (typically) catalog data. This page also includes a JavaScript autocompleter function that uses AJAX to render the type-ahead page, populate the drop-down list with the results, and fill in the text box with the user’s selection.

Note that the type-ahead data and the catalog data are included in the same index, so an additional search engine instance is not required. However, using separate indexes (and therefore separate search engine instances) is highly recommended for performance reasons.

The examples in this section use the `Ajax.Autocompleter` function from the script.aculo.us open source JavaScript library. For information about this library, see:

```
http://script.aculo.us
```

You can use an autocompleter function from a different JavaScript library instead, though you might then need to implement a different servlet bean or form handler to use on the type-ahead page.

The WAR file for the QueryConsole sample application includes the script.aculo.us JavaScript files. The WAR file is located at:

```
<ATG10dir>/DAF/Search/Base/QueryConsole/web-apps/queryconsole.war
```

This chapter includes the following sections:

Creating the Type-Ahead Page

Creating the Search Page

For information about configuring ATG Search to use type-ahead data, see the *ATG Search Administration Guide*.

Creating the Type-Ahead Page

The type-ahead page is a JSP that is rendered periodically as the user types text in the search box. The frequency of rendering the page is controlled by the JavaScript autocompleter function, which typically polls for changes several times a second.

The following is an example of a type-ahead page that uses the `atg.search.formhandlers.TypeAheadDroplet`. This servlet bean submits a request of class `atg.search.routing.command.search.TypeAheadRequest` and then renders the results. Most of the input parameters for the servlet bean are passed into the page as query parameters when the page is posted by the JavaScript function.

```
<dsp:page>
<ul>
<dsp:droplet name="/atg/search/formhandlers/TypeAheadDroplet">
  <dsp:param bean="/atg/search/formhandlers/SearchContext" name="context"/>
  <dsp:param name="text" param="q"/>
  <dsp:param name="environment" param="environment"/>
  <dsp:param name="language" param="language"/>
  <dsp:oparam name="itemformat">
    <li><dsp:valueof param="match"/></li>
  </dsp:oparam>
</dsp:droplet>
</ul>
</dsp:page>
```

Creating the Search Page

On the search page, you include a `<div>` element in the search form to provide a hook for associating the type-ahead drop-down list with the search text input:

```
<p>Search for: </p>
<dsp:input type="text" id="question" size="30" converter="nullable"
           name="question" bean="QueryFormHandler.searchRequest.question"/>
<div id="autocomplete_choices" class="autocomplete"></div>
```

The page also needs JavaScript to call the autocompleter function. The example below creates a JavaScript function called `buildQueryCallback`, which constructs the URL (including query parameters) for rendering the type-ahead page. The example also creates a JavaScript function called `AutoComp`, which is executed when the page loads. The `AutoComp` function:

- Calls the `Ajax.Autocompleter` function with arguments that associate it with the text field and `<div>` element in the search form shown above.
- Renders the type-ahead page using the URL constructed by the `buildQueryCallback` function, adding a query parameter q whose value is the string currently in the text box.

```
<script language="Javascript">
function buildQueryCallback(element, entry) {
  entry += "&environment=commerce";
  entry += "&language=english";
  return entry;
}

function AutoComp() {
  var myAutoCompleter1 = new Ajax.Autocompleter('question',
    'autocomplete_choices', 'testtypeget.jsp' , {frequency: 0.2,
     callback: buildQueryCallback, paramName: "q" });
}

document.onload = AutoComp();
</script>
```

For example, if the user types “fla”, the URL constructed is:

```
testtypeget.jsp?q=fla&environment=commerce&language=english
```

The query parameters are used to set the input parameters of the `TypeAheadDroplet` on the type-ahead page.

6 User-Entered Operators

Customers using your search page can by default include operators that can modify how ATG Search interprets query terms. Depending on your customers' level of knowledge, you may want to provide them with information on how to use these operators, or prevent them from using the operators.

This chapter includes the following sections:

Literal Operator

If a single word is surrounded by double-quotes (for example, *"books"*), ATG Search interprets this as a literal constraint, meaning that the user wants the query term to match that word form and nothing else. Note that double-quotes do not enforce an alphabetic case match.

Double-quotes disable term expansion for the quoted term, meaning that the term can only match the same index term. Double-quotes also require that the morphological information of the query term be identical to the retrieved index term. For example, the term *"books"* is the index term *book* plus a *+s* suffix, so the query will retrieve results with the index term *book*, but it will filter those that do not have the *+s* suffix.

If a sequence of terms is quoted (for instance, "book clubs"), the terms are required to appear in the same order in the retrieved result as they do in the query, and to be adjacent to each other. For example, the quoted string "book clubs" would mean the following:

- *book* must not be expanded and must match only this form of *book*, equivalent to the double-quoted query of "book"

- *clubs* must not be expanded and must match only this form of *club*, equivalent to the double-quoted query of “clubs”
- *book* and *clubs* must appear in that order in the retrieved result
- *book* and *clubs* must be next to each other

For the adjacency constraint, intervening stop-words or punctuation do not count. For example, *book – clubs* and *book a clubs* would satisfy the example query.

Required Terms

The operators described in the following sections can be used to require a search term to appear in the sentence or document result.

Required in Statement

By default, not all of the terms in a query are required to be in the statement result (Boolean OR). To require that a search term appear in the statement result, users can use the operator +. The + operator must immediately precede the query term, with no intervening white space, and there must be white space before the + operator, as in:

```
book +clubs
```

Note that since ATG Search applies morphology and term expansion by default, the + operator applies to the explicit query index term **plus** all of its expansions. In order to require a literal term in the statement results, use the double-quote operator with the + operator, as in the following example:

```
book +"clubs"
```

If the query includes multiple terms with the + operator, all of them must appear in each statement result.

Required in Document

The special operator ++ requires a term to appear in the document results, but not necessarily the sentence results. This is useful if a user wants to constrain the search to documents *about* a term, yet not require that the sentence results from those documents contain the term. The ++ operator must immediately precede the query term, with no intervening white space, and there must be white space before the ++ operator, as in:

```
book ++clubs
```

Note that since ATG Search applies morphology and term expansion by default, the ++ operator applies to the query index term **plus** all of its expansions. To require a literal term in the document results, the query can use the double-quote operator with the ++ operator, as in:

```
book ++"clubs"
```

Multiple terms with the ++ operator form a Boolean AND of those terms; all of them must appear in the documents of each result.

The operator +| requires a term to appear in the document results, but not necessarily the sentence results, in the same way as the ++ operator. Unlike the ++ operator, however, multiple terms with the +| operator form a Boolean OR; at least one of them must appear in the documents of each result. This is useful for situations where there are alternative terms that reflect the overall content that the user wishes to search within.

Note that a single query can have terms that use each of these operators. First, the terms with the ++ operator are used to constrain the document candidates to those which contain all of these terms. Next, the terms with the +| operator are also used to further constrain the document candidates to those which contain one of these terms. Finally, the terms with the + operator are used to restrict the sentence results to those that contain these terms.

Excluded Terms

Users can eliminate undesired search results by using ! and !! to exclude terms, in effect a Boolean NOT.

Excluded in Statement Results

The operator ! requires that a term not appear in the statement results. The ! operator must immediately precede the query term, with no intervening white space, and there must be white space before the ! operator, as in:

```
clubs !book
```

Because ATG Search applies morphology and term expansion by default, the ! operator applies to the query index term **plus** all of its expansions. To require a literal term not to appear in the statement results, use the double-quote operator with the ! operator, such as

```
clubs !"book"
```

Note: Other search engines use the - operator, but this operator can be mistaken for a hyphen; therefore ATG Search uses the exclamation mark, also a common symbol for a NOT operator.

Excluded in Document Results

The operator !! requires a term not to appear in the document results, and therefore all of the sentence results. This is useful if a user wants to exclude documents with a term from the search, no matter where they appear in those documents. The !! operator must immediately precede the query term, with no intervening white space, and there must be white space before the !! operator, as in:

```
clubs !!book
```

Since ATG Search applies morphology and term expansion by default, the !! operator applies to the original query index term plus all of its expansions. To exclude a literal term from the document results, use the double-quote operator with the !! operator, as in:

```
clubs !!"book"
```

Case Restriction

ATG Search interprets single quotes as a constraint on the case of the terms inside the quotes. ATG Search characterizes each query term into three types of case:

- Upper—All upper-case letters
- Lower—All lower-case letters
- Mixed—Both upper and lower-case letters

Mixed includes the common title case (for example, *The*) as well as non-standard case forms, such as *iPod* and *NeXT*. If a query term is single-quoted, then it is constrained to retrieve only index terms that match its case. For example, a query with *'IT'* will only retrieve results with the index term *it* in upper case.

Wildcards

A wildcard (*) is treated as a character pattern that matches many index terms at once. ATG Search uses asterisks in a term to denote a wildcard, where the asterisk can match zero or more other characters. At least one other non-asterisk character must appear in the query term. The wildcard can be used in the following ways:

- At the end of the term. For example, *book** matches any index term that begins with the substring *book*, such as *book*, *books*, *booking*, and *bookshelf*.
- At the beginning of the term, such as with **desk*, which matches terms such as *desk*, *workdesk*, and *computerdesk*.
- Within a term, such as *inst*ion*, which matches index terms such as *installation* and *institution*.
- Multiple wildcards, such as **install**, match any index term with *install* in the middle.

Wildcards are a form of term expansion, because a single query term is replaced with alternative terms. But in this case, the expansions are not from a thesaurus, but based solely on the characters of the terms in the index and the wildcard pattern.

Note: Wildcard patterns expand to index terms, **not** to morphological forms of words, so the results are not always obvious. For example, **desk* expands to *workdesk*, which retrieves all forms of *workdesk*, including *workdesks*, which doesn't really match the wildcard pattern due to the trailing *s*. This behavior is by design and consistent with the rest of ATG Search's query handling and search results. Users should understand that the wildcards are matching against the dictionary of index terms, not literally across the document text.

Wildcards can expand to hundreds or thousands of index terms with patterns like *s**. To prevent slowdowns and poor search results, ATG Search limits the number of expansions a wildcard can produce. This limit is configurable, as described in the `wildCardMax` section of Appendix B: Search XML Reference. The limit defaults to 20.

Regular Expressions

ATG Search uses a common regular expression pattern syntax consisting of the following operand types:

Operand	Description
.	Match any character
x	Match character *x*
\x	Match character *x*, which might otherwise have special meaning to the syntax, such as +, *, ?, (,), and /.
[*set*]	Match any character belonging to the given set, where hyphens denote a range
[^*set*]	Match any character that does not belong to the given set, where hyphens denote a range

The language allows each operand to have an optional operator immediately following it:

Syntax	**Description**
operand?	Match zero or one of the operand
operand*	Match zero or more of the operand
operand+	Match one or more of the operand

The language allows operands and operators to be grouped by parentheses to form a larger operand, which also can take operators.

To use a regular expression in a query, it must be denoted as shown:

```
re/regexp/
```

For efficiency, ATG Search requires that the *regexp* pattern must contain at least one non-optional, non-negative operand, which means either a literal (non-`.`) character or a `[set]` operand without a * or ? operator.

For example, *book.** is a wildcard term that matches any index term that starts with the sub-string book, such as *book*, *books*, *booking*, and *bookshelf*. An example of a set operand, the pattern `r[oa]m` will match the index terms *rom* and *ram* only.

Regular expressions are a form of term expansion, because a single query term is replaced with a disjunction of alternative terms. But in this case, the expansions are not from a thesaurus, but based solely on the characters of the terms in the index and the regexp pattern.

Note: The `regexp` patterns expand to index terms, not to morphological forms of words, so the results are not always intuitively obvious. For example, `.*desk` expands to *workdesk*, but this could retrieve all forms of *workdesk*, including *workdesks*, which doesn't really match the `regexp` pattern due to the trailing *s*. This behavior is by design, since it makes it consistent with the rest of ATG Search's query handling and search results. Users must understand that the regular expressions are matching against the ATG Search dictionary of index terms, not literally across the text of the collection.

Regular expressions can expand to hundreds or thousands of index terms with patterns like *s.**. To prevent slow searching and poor results, ATG Search limits the number of expansions a regular expression can produce. This limit is configured by the `wildCardMax` described in Appendix B: Search XML Reference. The default is 20.

Number Ranges

A number range is a query term that is treated as a numeric pattern that matches many index terms at once. ATG Search defines a number range as two numbers, separated by two periods, with no spaces (i..j). The first number is the minimum value, and the second number is the maximum value of the range. Any numeric index term that falls between the first and second numbers inclusively is retrieved. Integers and real numbers are not distinguished. For example, the number range 5..8 returns *6*, *5.7*, and *8.0*.

Number ranges can be viewed as a form of term expansion, because a single query term is replaced with a disjunction of alternative terms. Note that only numbers that appear in the collection are considered for these ranges. There is no limit to the number of included numbers.

Operator Combinations

Some query operators can be combined on a single term; others cannot, while some must be combined in a certain order. This section explains how to combine operators.

Natural language query terms can use the Boolean operators plus both quoted strings, as shown below. The Boolean operators must appear first, with no white space after them. Only one Boolean operator is allowed per term. The single quote operator can appear next, surrounding the term and optional double-quote operators. The double-quote operators are innermost.

```
[+,!,++,+|,!!][']["]term["][']
```

A wildcard query term can use the Boolean operators and only the single quote operator, as shown below. The double quote operator denotes a literal match constraint and therefore conflicts with the wildcard pattern operator.

```
[+,!,++,+|,!!][']wildcard[']
```

A number range query term can use only the Boolean operators, as shown below. The number range is a pattern and conflicts with the double quote operator. Because numbers do not have case, the single quote operator is not applicable.

```
[+,!,++,+|,!!]i..j
```

Fielded Search Operators

ATG Search stores features on each sentence term vector which represent different regions, also called *zones* or *fields*, of the text content. The search can be restricted to certain fields, therefore this functionality is called *fielded search*. Normally, the user interface would expose the controls for fielded search, which are passed as parameters. However, ATG Search also provides special query operators for specifying the fields within the query input. The operator syntax is:

```
zones:field,field,...

fields:field,field,...
```

The `zones` operator controls which fields are searched for normal unstructured content, and corresponds to the `/activeSentenceZones` setting (see the Search Fields section). By default, ATG Search searches only the body of the unstructured content, which has a field name of `doc`. However, unstructured content also has two other fields: a title field (named `role:title`) and a URL field (named `role:url`). The title field typically represents the metadata title element, such as the `<title>` element of HTML. The URL field is a special field that contains the words of the URL of the index item. The following shows how to search all three fields, plus each of the special fields individually:

```
zones:doc,role:title,role:url rest_of_query

zones:role:title rest_of_query

zones:role:url rest_of_query
```

The `fields` operator controls which fields are searched for structured content, and corresponds to the `/activeSolutionZones` setting (see the Search Fields section). By default, ATG Search searches only the goal, symptom, and id fields of the structured solution content, which have field names of `role:goal,role:symptom,role:id`.

However, structured content can have an unlimited number of fields, depending on the extracted structured content. Structured content does not necessarily have a title or URL field like unstructured content, but it can if its structured representation contains them. The following shows how to search on six fields for typical solution content, plus a search on just the goal and just the ID:

```
fields:role:goal,role:symptom,role:fact,role:fix,role:cause,role:change,
rest_of_query
```

```
fields:role:goal rest_of_query
```

```
fields:role:id rest_of_query
```

7 Constraining Queries

ATG Search queries can include constraints, which limit the search space within the index. This chapter describes the following topics:

Document set and item constraints can be arbitrarily combined into a single Boolean expression of individual constraints. Documents that satisfy the Boolean expression are accessible for searching; documents that do not satisfy the expression are inaccessible.

ATG Search uses XML to represent the query constraint expressions. Constraints are included in a query through the `expression` tag and its subtags (see the Combining Query Constraints section of this chapter for XML details). If you want to give your customers access to this feature, you must include the appropriate controls in the form handler.

The XML is constructed programmatically. Constraints can also be configured using metadata or query rules; see the *ATG Search Administration Guide* for information on these methods.

Document Set Constraints

A *document set* is a collection of source index items that are indexed and deployed together. Document sets can represent directories, metadata, or item categorization. During a query, ATG Search can restrict its search to a particular document set or a document set and all its child sets.

The `subdirs` attribute determines whether child sets are included `(true)` or excluded `(false)`. The *docset_path* is the pathname for the document set, beginning with an initial forward slash denoting the root.

The XML representation for this constraint is:

```
<set subdirs="true|false">docset_path</set>
```

For example, the following XML constrains the search to documents that are included in the `/Meta/fall_styles/shoes` document set and its subdirectories:

```
<set subdirs="true">/Meta/fall_styles/shoes</set>
```

In ATG Search, there are four root paths:

- `/Documents` – Physical directories of unstructured content
- `/Solutions` – Physical directories of structured content
- `/Topics` – Virtual directories for categorization results
- `/Meta` – Virtual directories for metadata

Collection Constraints

Collection constraints function much like document sets, and are ideal for use in B2B environments, where catalog-based document sets can be numerous enough to slow searching. Collections do not include hierarchies and cannot be browsed, but are a simple switch that indicates whether a given document is part of the collection.

```
<collection>/Meta/name/value</collection>
```

In order to use collection constraints, index your content with `store-as-collection=true` in the `IndexingOutputConfig` component.

Property Constraints

ATG Search represents index item metadata as properties stored with the index item. During a query, ATG Search can restrict its search to items with certain property values. Numeric property values can be tested for equality, less-than, and greater-than; string property values can test for sub-string matches; and both types can be tested for range matches. The XML representation for these constraints is:

```
<prop type="type" name="name" op="str_op|num_op" case="true|false">value
</prop>

<strprop name="name" op="str_op" case="true|false">value</strprop>

<numprop name="name" op="num_op" >value</numprop>
```

The constraints are:

- `type`—One of six possible property types, `enum`, `string`, `float`, `integer`, `boolean` and `date`. Note that for properties of type `boolean`, the value must be provided as a 1 or 0. For example:

  ```
  <prop type="boolean" name="onSale">1</prop>
  ```

  The `strprop` constraint is equivalent to the type values of `enum` and `string`, and the `numprop` constraint is equivalent to the other four type values.
- `name` —The name of the property.
- `value` —The operand value for the constraint.
- `op` —Contains the comparative operator for the constraint, which defaults to `equal`.

All constraints allow the following operators:

- `equal`—The index item must have a property value equal to the operand
- `greater`—The index item must have a property value greater than the operand
- `greatereq`—The index item must have a property value greater than or equal to the operand
- `less`—The index item must have a property value less than the operand
- `lesseq`—The index item must have a property value less than or equal to the operand
- `greater-less`—The index item must have a property value greater than the initial range value and less than the final range value, where the range operand is expressed as *initial-final*.
- `greatereq-lesseq`—The index item must have a property value greater than or equal the initial range value and less than or equal the final range value, where the range operand is expressed as *initial-final*.
- `greatereq-less`—The index item must have a property value greater than or equal the initial range value and less than the final range value, expressed as *initial-final*.
- `greater-lesseq`—The index item must have a property value greater than the initial range value and less than or equal to the final range, where the range operand is expressed as *initial-final*.

For `string` and `enum` property constraints, the comparisons are character byte comparisons. In addition, the `enum` and `string` property constraints allow three more operators:

- `contains`—The index item must have a property value that contains the operand
- `starts`—The index item must have a property value that starts with the operand
- `ends`—The index item must have a property value that ends with the operand

For `string` property constraints, the additional `case` attribute controls whether the operator should be case-sensitive (`true`) or case-insensitive (`false`). If the operator is a range operator, then the `value` is a range of values expressed as *initial-final*.

Index Item Constraints

This section discusses various types of index item constraints.

Index Item URL Constraints

Index items require a URL identifier, even if they are not physical files or accessible by Web URLs. This identifier is stored in the index, and queries can constrain against this URL, which means that only a single index item will satisfy this constraint. Typically, a set of these constraints are grouped into a Boolean OR to limit the search to a small subset of index items. The XML for this constraint is shown below:

```
<doc>URL</doc>
```

Index Item Format Constraints

Queries can be constrained to items of a particular format. The XML representation for these constraints is:

```
<format>format</format>
```

The `format` operand can be one any of the following:

- HTML – Simple HTML of any style
- Text – Raw text format
- PDF – Adobe format
- XHTML – w3.org XML schema for HTML
- XML – XML format
- Other – Any other format, typically from word or data processing software, such as Word, Power Point, or Excel
- All – Any type

Index Item Language

Content is analyzed and indexed with respect to a specified language, and this language is recorded in the index. Therefore, queries can be constrained to items of a certain language. The XML representation for these constraints is shown below:

```
<language> lang</language>
```

The *`lang`* operand can be any valid language name or code, such as `English` or en.

Index Item File Extension

Index items require a URL identifier, even if they are not actually physical files or accessible by Web URLs. This identifier is stored in the index, and queries can constrain against the file extension. For example, the URL *http://www.oracle.com/index.htm* has a file extension of *htm*. The XML for this constraint is:

```
<type>file_suffix</type>
```

The `file_suffix` operand is the file extension characters, not including the period.

Index Item Modified Date

Index items typically have a last modified date associated with them. Queries can be constrained to items with a certain date or a range of dates. The XML representation of this constraint is:

```
<date op="num_op">date_string</date>
```

This constraint has the same comparative operator values as the number property constraint (see Property Constraints). The `date_string` is either a valid date value or a range of valid values, expressed as *initial_date-final_date*. The valid forms of data values are:

- *YYYY* – A four digit year
- *YY* – A two digit year, assumes 1900.
- *MM YY* – Month number followed by two digit year
- *MM YYYY* – Month number followed by four digit year
- *MM DD YY* – Month number followed by day number followed by two digit year
- *MM DD YYYY* – Month number followed by day number followed by four digit year
- *month DD YY* – Month name followed by day number followed by two digit year
- *month DD YYYY* – Month name followed by day number followed by four digit year

For all forms, the month, day, and year components must be separated by one of the following delimiters: space, hyphen, period, comma, slash, or backslash.

Combining Query Constraints

ATG Search allows you to express arbitrary Boolean combinations of the index item constraints described previously. For example, to constrain a query to one of three document sets, you can construct a Boolean OR of three document set constraints; to constrain a query to documents with two property values, create a Boolean AND of two property constraints. To constrain a query to documents without a property value or not within a document set, a Boolean NOT of those constraints would be used.

The Backus Naur Form for the XML representation of the query constraint expressions is shown below.

```
expression := or_exp | and_exp | not_exp | set_exp | format_exp | type_exp
| doc_exp | date_exp | prop_exp | strprop_exp | numprop_exp ;
```

```
or_exp := "<or>" expression+ "</or>" ;

and_exp := "<and>" expression+ "</and>" ;

not_exp := "<not>" expression+ "</not>" ;

set_exp := "<set subdirs=\"true|false\">" docset_path "</set>"

format_exp := "<format>" format "</format>"

type_exp := "<type>" file_suffix "</type>"

doc_exp := "<doc>" URL "</doc>"

date_exp := "<date op=\"num_op\" >" date_string "</date>"

prop_exp := "<prop type=\"type\" name=\"name\" op=\"str_op\" >" value
"</prop>"

strprop_exp := "<strprop name=\"name\" op=\"str_op\" >" value "</strprop>"

numprop_exp := "<numprop name=\"name\" op=\"num_op\" >" value "</numprop>"

format := html|text|pdf|other|xhtml|xml|all

date_string := YYYY|YY|MMYY|MMYYYY|MMDDYY|MMDDYYYY

date_string := month DD YY|YYYY

num_op := greater|greatereq|less|lesseq|equal|between|within

str_op :=
greater|greatereq|less|lesseq|equal|between|within|contains|starts|ends
```

An <or> element represents a Boolean OR. The statement is `true` if one of its contained expressions is `true`, and otherwise evaluates to `false`.

The <and> element represents a Boolean AND, which is `true` if all of its contained expressions are `true`, and otherwise evaluates to `false`.

The <not> element represents a Boolean NOT, which is `true` if none of its contained expressions are `true`, and otherwise evaluates to FALSE.

Weighted Metadata Preference Expressions

The previous sections described query constraint expressions that represent arbitrarily complex Boolean expressions of metadata constraints, including document sets, properties and URL constraints. In some circumstances, a hard constraint is not appropriate, and a *preference* for certain content is desired. For example, to support personalization, an application might prefer certain content depending on the user's profile.

ATG Search supports this functionality by allowing a second expression to be included in the request XML. This second expression includes weights which will affect the relevancy calculation of the results. Results that satisfy the expression receive extra weight, and results that do not, receive no extra weight.

The metadata expressions use the same XML format as the query constraints, except that each basic element requires a `weight` attribute, such as:

```
<set subdirs="bool" weight="N"

<strprop name="name" weight="N"
```

The *N* value is arbitrary and will be relative to the maximum weight for any given expression. The three Boolean expressions do not use the weight attribute, for their weight is based on the weight of their operands, as follows:

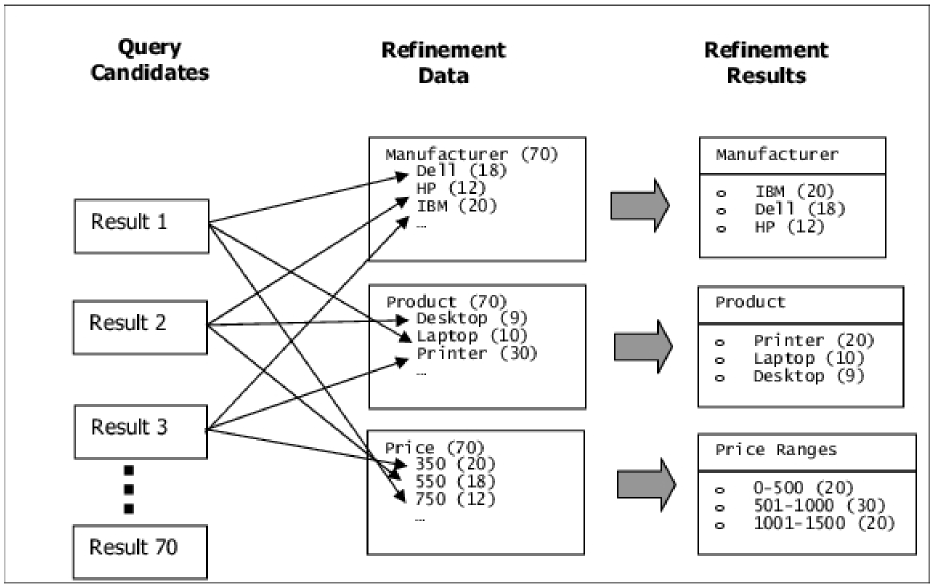
- Boolean `<and>` expression has a weight equal to the **sum** of its operands
- Boolean `<or>` expression has a weight equal to the **maximum** weight of its operands
- Boolean `<not>` expression has a weight equal to **zero minus the sum** of its operands

Thus, the entire metadata expression results in a total weight using the mathematical combination of the individual weights. This total weight is normalized into a relevancy value and factored into total relevancy.

Query Refinements

Another type of user feedback involves metadata properties of the retrieved index items. ATG Search can return properties and values which can segment the retrieval results. This refinement is controlled by configuration data defined by the administrator. Note that this feature is referred to as refinement configuration in ATG Merchandising; see the *ATG Merchandising User Guide* for information.

The configuration data specifies which properties to use in refinement, the order in which they should be used, and various settings to control how to construct the refinement. For example, the configuration data might specify that manufacturer, product type, and price should be used, where price should be returned in ranges and the other two in enumerated lists, limited to the top three values.



Query Refinement

As another example, the configuration data can specify that the `country` property will be used initially until all results are from the same country, then state, then county, and so on. The property refinement is dynamic depending on the result set.

8 JMS Event Handling

When a `QueryRequest` is issued by `QueryFormHandler` and a response is received, a JMS event (message) of class `atg.search.events.QueryMessage` is fired. Search events are used to create logs for reporting purposes. You can also add configuration to use them for other purposes, such as triggering scenarios.

This chapter discusses search events and how to configure them. It includes these sections:

For information about the properties of the `QueryMessage` class, see the *ATG API Reference*.

Search Messaging Components

The `/atg/search/formhandlers/SearchContext` component has two properties that control the firing of search messages:

Property	Description
`firingSearchEvents`	A Boolean which specifies whether firing of search events is enabled. If `false`, no search events are fired; if `true`, search events are enabled and are managed by the component specified in the `searchMessageService` property.
`searchMessageService`	A component of class `atg.search.events.SearchMessageTools`, which manages the firing of the search events. By default, the `SearchMessageService` property is set to `/atg/search/events/SearchMessageService`.

The `/atg/search/events/SearchMessageService` component is responsible for constructing and sending search messages. It has a number of properties that configure the firing of messages:

Property	Description
`spiderlikeTypes`	An array of `atg.servlet.BrowserType` components for which search events are not fired.
`browserTyper`	The `browserTyper` component used to determine the browser type the request is coming from.
`messageFilter`	A component of class `atg.search.events.MessageFilter` that is configured with a list of IP addresses and a list of login names for which search events are not fired. By default, the `messageFilter` property is set to `/atg/search/events/MessageFilter`.
`searchMessageSource`	The Patch Bay message source, of class `atg.search.messages.SearchMessageSource`, that fires JMS messages when results are received from ATG Search. The `SearchMessageService` component uses the data in the `Response` object to construct a message object, which is then sent off by the `SearchMessageSource` component.

Suppressing Search Messages

The `SearchMessageService` component includes logic for determining, when a response is received, whether to send a JMS message. The purpose of this logic is to prevent sending of messages that will distort the search reporting results. For example, if a query has been initiated by a Web spider or by the Search Testing feature in ATG Merchandising rather than by a site visitor, you typically will not want to take this query into account in reports.

`SearchMessageService` suppresses messages in the following situations:

- If the request is determined to come from browser type that is considered a Web spider
- If the request comes from a specified IP address or user account

Detecting Web Spiders

Web spiders (also called robots) crawl the Web and create indexes for Web search services such as Google. This activity is generally benign, but it can skew reporting results. For example, if a spider issues a search query, that query will be reflected in search reports you run. This is generally undesirable, since you typically want the reports to reflect only queries issued by actual site visitors. Therefore, the `SearchMessageService` has a mechanism for determining whether a search query is being issued by a spider, and if it is, suppressing the firing of search events.

To enable this mechanism, you set the `SearchMessageService` component's `spiderlikeTypes` property to an array of the `atg.servlet.BrowserType` components that you consider to be spiders. When a search query is issued, the `SearchMessageService` examines the `userAgent` property of the `QueryRequest` component. `SearchMessageService` compares the value of the `userAgent` property with the values of the patterns properties of the spiderlikeTypes components, and if it finds a match, suppresses the events.

Typically, the search request's userAgent property is set to the value of the User-Agent property of the HTTP request. You can override this value by explicitly setting the userAgent property in the properties file of the search request component. This is what happens in the ATG Merchandising Search Testing environment. When you use Search Testing, the userAgent property of the QueryRequest component is set to SearchTesting. By default, one of the spiderlikeTypes components is /atg/dynamo/servletpipeline/BrowserTypes/Robot, and one of the entries in this component's patterns array is SearchTesting, so events are not fired.

Filtering by IP Address or User Account

The SearchMessageService has a messageFilter property that points to a component of class atg.search.events.MessageFilter. The MessageFilter component determines whether to suppress search messages, based on the IP address or the user account associated with the request. By default, the SearchMessageService component's messageFilter property is set to /atg/search/events/MessageFilter.

To configure this component, set the following properties:

Property	Description
IPAddressList	A array of the IP addresses that search messages should not be fired for. Note that the entries can include wildcards (e.g., 10.1.4.*).
loginList	A list of usernames that search messages should not be fired for.

Patch Bay Configuration

The DAF.Search.Base module includes a dynamoMessagingSystem.xml (Patch Bay configuration) file that declares the /atg/search/events/SearchMessageSource component as a message source, and also defines a number of message sinks.

If you want other message sinks to listen for search events, you can add your own dynamoMessagingSystem.xml file to your CONFIGPATH. You can also configure the ScenarioManager to listen for search events, if you want to use search events to trigger scenario actions. For more information about configuring Patch Bay, see the *ATG Programming Guide*.

9 Caching Search Query Data

You can configure ATG Search to cache entire user queries and their responses. This can improve performance by removing load from the search engines, and is most useful when many users are doing browse-type searches (such as faceted navigation) that result in repeated identical queries.

The cache stores the environment name, the query XML, and the response XML. By default, the cache stores most recently used queries and their responses in memory, while older responses are stored on disk.

Any ATG instance running the `DAF.Search.Routing` module can cache search data; caches are created as needed, with one cache per search environment.

This chapter includes these sections:

Search caching is implemented through the `atg.service.cache.persistent.ehcache.EHCacheService` class, which invokes the Ehcache open source library. For more information about the `EHCacheService` class, see the *ATG API Reference*. For information about the Ehcache library, see `http://ehcache.sourceforge.net/`.

Configuring Search Caching

Search caching is enabled by default. You can disable caching by setting the `cacheService` property of the `RoutingSearchService` component to null. Note that if you disable caching, ATG Search does not clear out any existing cached queries, but it no longer retrieves existing cached queries or caches new queries.

If caching is enabled, you can configure its behavior by setting properties of the `/atg/search/routing/CacheService` component, which is of class `EHCacheService`. By default, this component includes these settings:

```
defaultMaxElementsInMemory=100
defaultOverflowToDisk=true
defaultMemoryStoreEvictionPolicy=LRU
```

```
defaultMaxElementsOnDisk=10000000
defaultDiskPersistent=true
```

These setting configure the cache to store a maximum of 100 queries in memory. When this number is exceeded, the least recently used queries are moved to disk, up to a total of 10,000,000. Cache contents are maintained on disk during ATG server restarts.

Disabling Cache Invalidation

By default, a search cache is invalidated each time the index is deployed. This ensures that out-of-date queries are removed from the cache.

Unfortunately, this results in valid queries being flushed as well. As a result, load on the search engines tends to spike after a deployment, then decrease over time as more and more queries are cached. This pattern is repeated after each deployment, resulting in large fluctuations in the load.

To avoid these fluctuations, you can disable cache invalidation, and instead have individual queries evicted based on how long they have been in the cache. If you do this, it is a good idea to also disable storing any queries on disk, because looking for queries to evict on disk is very slow. You will also need to increase the maximum number of queries to keep in memory (since none will be stored on disk), and specify how long to keep a query before evicting it from the cache. So you might configure the `CacheService` component like this:

```
defaultDisableInvalidation=true
defaultTimeToLiveSeconds=3600
defaultMaxElementsInMemory=10000
defaultOverflowToDisk=false
defaultMemoryStoreEvictionPolicy=LRU
defaultMaxElementsOnDisk=0
defaultDiskPersistent=false
```

Note that if query invalidation is disabled, the setting for `defaultTimeToLiveSeconds` involves a tradeoff between keeping queries in the cache that are no longer valid (because the index has changed and has been redeployed) and evicting ones that still are (because they have been present longer than the configured value).

Controlling the Caching of Individual Queries

Caching queries is most useful if the same queries are used repeatedly. Faceted search queries are particularly good candidates for caching, because the same query is likely to be issued by different users selecting the same facet values. Searches that include search text are less likely to be repeated exactly, so they may benefit less from caching.

If you know in advance that certain queries are unique, it is a good idea to disable caching of them. Otherwise your cache will grow quickly and provide little performance benefit. To disable caching of a query, you set the `cacheable` property of the `QueryRequest` component to `false`. For example:

```
<dsp:property bean="QueryFormHandler.searchRequest.cacheable"
    value="false"/>
```

Queries that Include Timestamps

Queries are especially likely to be unique if they have constraints that include timestamps. For example, suppose you configure queries to include a constraint that removes items whose start date is in the future (i.e., items that are not yet available). To do this, you might create a component of class `atg.search.routing.command.search.PropConstraint` and configure it so that an item is returned only if the value of its `startDate` property is less than or equal to the current date and time. Unfortunately, this constraint results in every query being unique, because the time changes continuously. So no benefit can be realized by caching the queries.

To prevent these kinds of queries from being unique, the ATG platform includes the class `atg.service.util.ChunkedTimeInterval`. This class divides up time into discrete intervals of a specified length, and during an individual interval always returns the same time. If you use this class to include timestamps in search queries, all queries created during an individual interval will have identical timestamps, so queries that are otherwise identical will remain identical. At the end of one interval, a new interval begins and the timestamp changes, but then remains constant for the duration of the new interval.

`ChunkedTimeInterval` has `intervalUnitName` and `intervalCount` properties for specifying the length of the interval. For example, to specify an interval of 4 hours:

```
intervalUnitName=hour
intervalCount=4
```

`ChunkedTimeInterval` has `startTime` and `endTime` properties that it sets to the time at the beginning and end of the interval (in milliseconds, using Coordinated Universal Time). So, for example, if an interval is 4 hours long and begins at 4:00 am on July 3, 2010, then `startTime` is set to 4:00 am of that day and `endTime` is set to 8:00 am. `ChunkedTimeInterval` also has a number of other properties that it sets to the interval start and end times in different formats. For search constraints, use either the `startTimeSecondsAsString` or the `endTimeSecondsAsString` property, as these properties return the time in a format that is most suited to inclusion in search queries.

The `DCS.Search.Query` module includes a `ChunkedTimeInterval` component named `/atg/commerce/search/catalog/ProductAvailabilityTimeWindow` which is configured to use intervals of one day. You can change this interval or create your own component of this class. The following example shows a `PropConstraint` properties file that uses this component to include a timestamp in a constraint:

```
type=integer
name=startDate
operation=lesseq
value^=/atg/commerce/search/catalog/\
   ProductAvailabilityTimeWindow.startTimeSecondsAsString
```

For more information about the `ChunkedTimeInterval` class, see the *ATG API Reference*.

Using the Cache Administration Page

To administer search caching, navigate to `/atg/search/routing/CacheService` in the Component Browser of the ATG Dynamo Server Admin. The Cache Statistics section provides information on current cache configuration and effectiveness.

Note that the accuracy of many of these statistics may be affected by the value of the `statisticsAccuracy` property of `EHCacheService`. There is tradeoff between the accuracy of the values and the goal of minimizing the system resources required to gather the values. You can increase the accuracy by changing the setting of this property, as described in the *ATG API Reference*.

The page also provides buttons for the following actions:

- Clear Statistics -- Resets to zero the statistics that appear in the `CacheService` charts (see below).
- Disable/Enable -- Turns the caching service off or on. (Note that this does not disable searching.)
- Flush to Disk -- Saves the portion of the cache currently in memory to disk.
- Clear Cache -- Deletes all cached information.
- Disable Invalidation -- Disables cache invalidation.

The page includes the following charts, which show how caching affects your site:

- Hit Rates -- The percentage of queries that have been satisfied by cached data. A high hit rate means that caching is working well.
- Maintenance Time -- The percentage of time the cache spent purging itself vs. serving requests. If this number is more than a few percent, try increasing the value of `defaultMaxElementsInMemory` to decrease the amount of time the system spends moving items between disk and memory.
- Requests/Hits per Second -- The number of search requests, the number of those requests that were served by cached data, and the number of items evicted from the cache.

Note: If you are using the `QueryConsole` sample application to test search behavior, the Cache Responses setting is `false` by default; change this value before testing.

10 Faceted Search

The Faceted Search feature enables ATG Commerce sites to provide a navigational structure that is not strictly based on the catalog hierarchy. Facets are like virtual categories that are populated by the results of search queries. Facets are implemented as search refinements, which are used to narrow down search results by searching within those results for only the items that fulfill a certain constraint. For example, a search might return men's shirts, and then the customer might select a facet value that narrows the results to men's shirts that are made of cotton.

This chapter describes how to write pages that implement Faceted Search on an ATG Commerce site. It includes the following sections:

Note that Faceted Search extends the Search querying and response-handling mechanisms described in the Search Form Handlers chapter. Be sure to familiarize yourself with the information in that chapter before reading this chapter.

Overview of Faceted Search

A facet is a search refinement element that corresponds to a property of a product or SKU. The property is referred to as a faceting property. The values of this property are broken down into selections that can be either ranges or specific values. For example, you might define a price facet whose faceting property is the `salePrice` property of a product's SKUs. The selection ranges, which can either be determined dynamically or specified explicitly, might be $100 to $200, $200 to $500, $500 to $1000, etc. Or you might define a manufacturer facet with selection values of Acme, Cogswell, and Spacely.

You specify facets and the logic for determining the selections in ATG Merchandising as part of creating your product catalog. Each facet corresponds to a property of a commerce item, and can be associated with one or more categories or catalogs. Each facet is stored in the `RefinementRepository` as a separate `refineElement` repository item. When you deploy your catalog to your production site, the

`RefinementRepository` is deployed as well. When you index your catalog, the data in the `RefinementRepository` is used to create the refinement configuration files used by ATG Search. These are the XML files that define sets of facets and the facet values or ranges. These files are generated and submitted to ATG Search after an indexing operation. When ATG Commerce issues a query to ATG Search, it determines which refinement configuration file should be applied, and specifies it in the query.

When you write JSPs for displaying facets, you render the facet selection values as hyperlinks. When a customer clicks one of these links, a query is issued to ATG Search, using the selection range or value as a refinement criterion. For example, a customer might issue a text query that returns a set of products that are displayed on the page. If the customer then clicks the Sale Price facet's "$100 to $200" link, a new query is issued that specifies "return only the products in this set whose sale price is between $100 and $200." The results of this query are then displayed on the page, and the facet selections are updated.

See the *ATG Merchandising Guide for Business Users* for information about creating facets.

Building Pages that Include Facets

To use facets on your site, you create JSP pages that display the facet selections as hyperlinks. When a customer clicks one of these links, a query is issued to ATG Search. The search results returned are then displayed on the page. These results include only those items whose faceting property value is the selected value or falls within the selected range. The available facet selections are also updated to reflect the selections previously made. This process continues as the customer clicks further links.

To create pages that enable Faceted Search, you use the following classes:

- The form handler `atg.search.formhandlers.QueryFormHandler` issues the search queries that include the refinement information, and receives the responses that include the refinements. The `/atg/commerce/search/catalog/QueryFormHandler` component of this class is configured to work with faceted search.
- The `atg.commerce.search.refinement.custom.CustomCatalogFacetSearchTools` class processes the refinement data in the search request and response objects. The `/atg/commerce/search/catalog/QueryFormHandler` component is configured to use a component of this class, `/atg/search/repository/FacetSearchTools`.
- The `atg.repository.search.refinement.FacetTrail` class keeps track of the facet selections made by the customer.

The pages you write should not be associated with specific facets or selections. Instead, they should be written in a generic way, to be able to handle any set of facets defined in ATG Merchandising. Typically you write one primary page that deals with displaying the facets and manipulating the facet trail, and that page is re-rendered each time new results are returned. The remaining sections in this chapter describe how to do this.

Note: In ATG 2007.1 and earlier, faceted search queries and responses were handled through the `atg.repository.search.refinement.FacetSearchDroplet` and `atg.commerce.search.refinement.CommerceFacetSearchService` classes. These classes were deprecated in ATG 9.0, and `atg.search.formhandlers.QueryFormHandler` was modified to support faceted search queries and responses. As of ATG 10, the `FacetSearchDroplet`, `CommerceFacetSearchService`, and related classes have been removed from the ATG distribution. If your faceted search implementation is based on these classes, you must rewrite your JSPs to use `QueryFormHandler` instead.

Issuing Faceted Search Queries

You issue faceted search queries with the `QueryFormHandler`, which takes the `FacetTrail` object and various ATG Search query attributes and constructs a search query that specifies:

- Constraints created based on the facets and categories in the facet trail
- A refinement configuration, which is determined based on the entries in the facet trail
- Search refinement query attributes
- Pagination query attributes

The `QueryFormHandler` submits the search request to ATG Search and receives back a search response. The `FacetSearchTools` class handles the processing of the search request by creating constraints and specifying the refinement configuration to use, based on the entries in the facet trail. `FacetSearchTools` also processes the search response by converting the `Refinements` object into an array of facets that can be manipulated individually. The page developer can then use servlet beans or JSP tags to iterate through the response and display the resulting facets and selections and the products returned. The selection ranges or values can be displayed as hyperlinks which, when clicked, pass the new facet trail String and modification instructions as query parameters to the linked page.

To configure the request and response processing, a component of class `FacetSearchTools` is added to the `QueryFormHandler` component's `searchRequestProcessors` and `searchResponseProcessors` array properties. For example:

```
searchRequestProcessors+=/atg/search/repository/FacetSearchTools
searchResponseProcessors+=/atg/search/repository/FacetSearchTools
```

Note that the `/atg/commerce/search/catalog/QueryFormHandler` component is configured to use `/atg/search/repository/FacetSearchTools`.

The following example creates a search form where a user can enter text and then click a button to submit the query. The results are returned as a `QueryRequest.Response` object (stored in the `QueryRequest.searchResponse` property). The `FacetSearchTools` component converts the raw facet data (found in the `Refinements` object stored in the `QueryFormHandler.searchResponse.refinements` property) and stores the converted data in the `FacetSearchTools.facets` property for use on the page.

```
<dsp:form id="searchForm" name="searchForm" formid="searchForm"
            method="post" action="simpleFacet.jsp">
```

```
  <!-- the text field is linked to the searchRequest's question property -->
  <p>question:  <dsp:input type="text" id="question" size="60"
                name="question" bean="QueryFormHandler.searchRequest.question"/>

  <!-- submit button invokes the handleSearch method on the QueryFormHandler  -->
  <p><dsp:input type="submit" bean="QueryFormHandler.search" value="Search"/>

  <!-- get the search response, facets, and facet trail string    -->
  <!-- from the initial request                                    -->
  <dsp:getvalueof bean="QueryFormHandler.searchResponse"
     var="queryResponse" scope="request"/>
  <dsp:getvalueof bean="FacetSearchTools.facets" var="facetHolders"
     scope="request"/>
  <dsp:getvalueof param="trail" var="trailString"/>

. . .
</dsp:form>
```

Specifying the Category for the Query

When a customer navigates a site, facets are not returned until a search query is issued. If you want facets to appear without a customer explicitly entering a search query, you can silently issue a textless query when the customer selects a category. This is sometimes referred to as *category navigation*.

For example, suppose a site has top-level categories of Shoes, Hats, and Gloves. When the customer clicks the link for the Hats category, the products or SKUs in that category are displayed along with the appropriate facets for the category. The products or SKUs can be obtained using standard ATG Commerce catalog navigation or through a search query, but a search query must be issued to return the facets. The query is submitted when the page for the category is loaded, and specifies the category to return results and facets for.

To specify the category, you set the `startCategory` property of the `QueryRequest` component. For example:

```
<dsp:setvalue bean="QueryFormHandler.searchRequest.startCategory"
   param="/Meta/ancestorCategories.catalogSpecificId/${categoryId}"/>
<dsp:setvalue bean="QueryFormHandler.search" value="submit"/>
```

This code assumes the category ID has been passed to the page through a query parameter on the selected link. For example, the anchor tag for the link to the Hats category might look like this:

```
<a title="Hats" href="/store/browse/category.jsp?categoryId=cat50001">
```

Note that setting `startCategory` this way requires that the `ancestorCategories.catalogSpecificId` property be included in the index. This property is an array of the catalog-specific category IDs of a product's ancestor categories, which are determined by the `/atg/commerce/search/CustomCatalogCategoriesPropertyAccessor` component. The definition file of the `/atg/commerce/search/ProductCatalogOutputConfig` component includes this property by default:

```
<item is-multi="true" property-name="ancestorCategories">
  <meta-properties>
    <property store-as-docset="true" name="catalogSpecificId" type="string"
       property-accessor=\
         "/atg/commerce/search/CustomCatalogCategoriesPropertyAccessor"
       output-name="ancestorCategories.catalogSpecificId"
       is-non-repository-property="true" filter="unique"/>
  </meta-properties>
</item>
```

Restricting the Set of Facets and Selections

In some situations, you may not want your sites to display all available facets and selections. For example:

- If you have a large number of facets, you may want to display only the most important ones.
- If a facet has a large number of selection values or ranges, you may not want to display all of them. For example, if a Color facet has 20 values, you could display only the ones with the most results.
- If a facet returns only one selection value, you may not want to display that facet. For example, if the value of the `color` property for every result returned is `red`, you could suppress display of the Color facet, since it would have only a single selection value that would return the same set of results already being displayed.

You can restrict the set of facets and selection values that ATG Search returns by setting the following properties on the `/atg/commerce/search/catalog/QueryRequest` component:

refineMax
The maximum number of facets to return (configured value: 5). If the number of facets in the refinement configuration is greater than this number, only the top *n* facets (where `refineMax=`*n*) in terms of priority order are returned. (See Ordering Facets by Priority.) Note that if two or more facets have a nesting relationship, ATG Search returns only one facet in that nesting hierarchy.

refineTop
The maximum number of values or ranges to return for a facet (configured value: 5). If the number of available values is greater than this number, only the top *n* values (where `refineTop=`*n*) in terms of sort order are returned. The sort order for a facet's selection values is specified when the facet is defined in ATG Merchandising; you can choose to sort in descending order of the number of results in the selection, so that only the facet selections with the fewest results are eliminated.

refineMin
The minimum number of results that a facet value or range must have for that value or range to be returned. The configured value is 0, which means even facet values with no results are returned. For example, if a Size facet has values of Small, Medium, and Large, selections are displayed for all three values, even if the results include no items whose size is Small. Set the value of this property to 1 to return only facet values that include results.

refineMinVal
The minimum number of selection values or ranges that a facet must have in order for that facet to be returned. The default value is 1, which means all facets are returned, since a facet always has at least one selection value or range. Set the value of this property to 2 eliminate facets for which all results have the same value. For example, if every result has `color=red`, setting `refineMinVal` to 2 prevents ATG Search from returning the Color facet.

Using a Facet Trail

Because facet selections are cumulative, it is necessary to keep track of each facet selection a customer has made so far. This is done through a *facet trail*, which is stored as an object of class `atg.repository.search.refinement.FacetTrail`.

A facet trail is similar to a navigational breadcrumb trail, where each entry consists of a facet and a selection value or range for that facet. For example, a facet trail might be rendered on the page like this:

Manufacturer:Cogswell > Price:$300-$500 > Voltage:12-24

The first time the page is displayed, you can set the facet trail explicitly, or leave it empty. When the user makes a facet selection, you append entries to or delete entries from the facet trail, and include the updated facet trail in the new search query.

You can update the facet trail using the `CommerceFacetTrailDroplet`. This servlet bean takes as input a String representation of the current `FacetTrail` object plus instructions for modifying the facet trail based on selections chosen by the customer. (Typically these inputs are passed through HTTP request query parameters.) From these inputs, the `CommerceFacetTrailDroplet` constructs a new `FacetTrail` object.

For example, suppose a site visitor navigates by first selecting the Televisions category, and then selecting the LCD Televisions subcategory. (This assumes that a property whose values are categories is defined as a facet; see the ATG Merchandising documentation.) Next, she chooses the $1000 to $2000 selection range for the price facet, and then chooses Acme as the selection value for the manufacturer facet. The facet trail String would now look something like this:

```
1:cat444323:1:cat333222:2:1000-2000:32:Acme
```

In this example, `cat444323` is the repository ID of the Televisions category, and `cat333222` is the repository ID of the LCD Televisions category. The example also assumes the following facets have been defined in ATG Merchandising:

```
1 = category
2 = price
32 = manufacturer
```

The number identifying the facet is the repository ID of the corresponding `refinementElement` in the refinement repository, which is also used as the ID of the `refinementElement` XML attribute in the refinement configuration.

Your pages should display the current facets and selections, along with links for removing the selections. (Typically a link for removing a facet is an image of an X, or the word "remove.") Suppose the site visitor now clicks the link to remove the Televisions facet. When a category selection value is removed from the facet trail, all of that category's subcategories are also removed, so in this case the LCD Televisions category is removed as well. (In addition, any category-specific facets and selection values that no longer apply are also removed. For example, if the Televisions category has a Screen Size facet associated with it, removing the Televisions facet from the trail also removes the Screen Size facet and selections.)

The new facet trail String is therefore:

```
2:1000-2000:32:Acme
```

The displayed search results now consist of all products priced between $1000 and $2000 whose manufacturer is Acme. So if Acme also manufactures stereo systems, the ones in this price range will now be displayed.

Last Range Indicator

The constraint generated for a facet's last selection range is slightly different from the constraints for the other ranges. For example, suppose you have a price facet with three selection ranges: $1000 to $2000, $2000 to $3000, and $3000 to $4000. The first range includes items whose price is great than or equal to $1000 and less than $2000. The second range includes items whose price is great than or equal to $2000 and less than $3000. But the last range includes items whose price is great than or equal to $3000 and less than **or equal to** $4000.

If a site visitor chooses the last selection range for a facet, the facet trail must indicate this, so that the correct constraint can be constructed. The last range is indicated by appending |LAST to the entry for the range in the trail. In this example, if the site visitor selects the $3000 to $4000 range, the facet trail String would look something like this:

```
1:cat444323:2:3000-4000
```

Note that although |LAST is included in the facet trail String (and therefore may appear in URLs), it is not part of the label for the selection range, and therefore does not appear on the page itself. Also, you do not need to code your JSPs in any special way to deal with this selection range.

For more information about the `CommerceFacetTrailDroplet`, see Appendix A: Commerce Search Servlet Beans.

Supporting Multiple Selection Values

Many sites that use faceted search support selecting only a single value or range for a given facet. When a user clicks a link to make a selection, a search query is issued that specifies that selection as a constraint. So, for example, for a Color facet, when a customer clicks the link for the Red selection value, a search query is issued that returns only items whose `color` property is `red`.

Some sites allow users to make multiple selections for certain facets. For these facets, rather than encoding the selections as hyperlinks, each selection typically has an associated checkbox, so the user can choose multiple selections by checking their checkboxes; a separate button is provided to issue the query once the selections are chosen. (Encoding the individual selections as hyperlinks is also possible, but results in more queries being issued.)

Multiple selections can be combined either with Boolean OR or Boolean AND logic:

- Combining with Boolean OR returns results that match any of the selected values. For example, for a Color facet, if the user selects Yellow and Orange, a given item is returned if its `color` property is `yellow` **or** if it is `orange`.
- Combining with Boolean AND returns results that match all of the selected values. For example, suppose a product representing a laser printer has a `paperSizes` property that is an array of the paper sizes the printer accepts, and you have a Paper Sizes facet based on this property. If the user selects A4 and Letter for this facet, a given item is returned only if its `paperSizes` property includes `letter` **and** `a4`.

If multiple facet values are selected, you need a way to encode them in the facet trail. To specify multiple selections combined with Boolean OR, use the pipe (`|`) character. For example, to encode the Yellow and Orange selections for a Color facet, the facet trail entry for this facet would look something like this:

```
3:Yellow|Orange
```

To specify multiple selections combined with Boolean AND, use the dollar sign ($) character. For example, to encode the Letter and A4 selections for a Paper Size facet, the facet trail entry for this facet would look something like this:

```
3:Letter$A4
```

You can use the `/atg/commerce/search/refinement/FacetTrailString` component to construct facet trails that include facets with multiple selection values. This component, which is of class `atg.repository.search.refinement.FacetTrailString`, provides methods for adding facet values, including multiple selection values combined with Boolean OR or Boolean AND, to the facet trail string.

Working with the FacetTrail Object

The `FacetTrail` object stores information about the facets in its `facetValues` property, which holds an array of `atg.repository.search.refinement.FacetValue` objects. Each `FacetValue` object represents a single entry in the facet trail – a facet and its value. To render the facet trail on a page, you will typically need to access the following properties of the `FacetValue` objects:

facet
An object of class `atg.repository.search.refinement.Facet`. This is an abstract base class whose subclasses represent various types of refinement elements. A `Facet` object has `id` and `label` properties that hold the ID of the refinement element and some associated text. For standard facets, these value are the repository ID and the display name of the `refinementElement` in the Refinement Repository. For the special SRCH search text facet, `id` is `SRCH` and `label` defaults to `Search`. See Incorporating Search Text as a Facet.

value
The facet selection value. Depending on the facet type, this can be text or numeric,

and either a single value or a range. If the facet has multiple selection values, then this property is an array of values or ranges.

matchingDocsCount
The number of items with this facet value or in this range.

The examples in the Rendering the Facets section illustrate using these properties in JSPs.

Rendering the Facets

When a faceted search query is issued, ATG Search returns the facets as part of the `QueryRequest.Response` object. The `FacetSearchTools` component converts the raw facet data and stores the converted data in the `FacetSearchTools.facets` property as an array of objects of class `atg.repository.search.refinement.FacetHolder`. Each `FacetHolder` object represents a single facet and its values.

The facet is stored in the `FacetHolder.facet` property as an object of class `atg.repository.search.refinement.Facet`. (See Working with the FacetTrail Object.) The facet values are stored in the `FacetHolder.facetValueNodes` property as an array of `atg.repository.search.refinement.FacetValueNode` objects.

The following example illustrates rendering the facets and facet values. It accesses the `FacetSearchTools` object and iterates through the facets in its `facets` property. For each facet, it displays the available selections and indicates the number of results for each selection value or range.

Each value or range is rendered as a hyperlink. When a user clicks one of these links, the JSP:

- Adds the facet and selection value to the facet trail.
- Retrieves the current search request object (which has been saved in the `SearchSession` object), updates it, and resubmits the request. (See Modifying and Resubmitting the Request for more information.)
- Re-renders the page with the facets and selections returned in the new search response. (This fragment does not include the code for rendering the search results themselves.)

```
<dsp:getvalueof bean="QueryFormHandler.searchResponse" var="queryResponse"
   scope="request"/>
<dsp:getvalueof bean="FacetSearchTools.facets" var="facetHolders"
   scope="request"/>
<dsp:getvalueof param="trail" var="trailString"/>

<!-- iterate through the returned facets -->
<c:forEach items="${facetHolders}" var="facetHolder">
  <tr>
    <!-- display the name of the facet -->
    <td><c:out value="${facetHolder.facet.label}"/></td>
```

```
<!-- iterate through the values of the current facet -->
<td><c:forEach items="${facetHolder.facetValueNodes}" var="facetValueNode">

  <!-- use the CommerceFacetTrailDroplet to construct the facet trail   -->
  <dsp:droplet name="CommerceFacetTrailDroplet">

  <!-- pass in facet trail from previous request -->
  <dsp:setvalue param="trail" value="${trailString}"/>

  <!-- add the facet selection to the trail; also include the SRCH facet -->
  <!-- if there is search text and the SRCH facet hasn't been added yet  -->
  <c:if test="${ empty trailString and ! empty queryResponse.question }">
    <dsp:setvalue param="addFacet"
      value="SRCH:${queryResponse.question}:${facetValueNode.facetValue}"/>
  </c:if>
  <c:if test="${ ! empty trailString or empty queryResponse.question }">
    <dsp:setvalue param="addFacet" value="${facetValueNode.facetValue}"/>
  </c:if>

    <dsp:oparam name="output">

    <!-- expose the facetTrail string as a jstl variable -->
    <dsp:getvalueof param="facetTrail" var="facetTrail"/>

    <!-- display facet value as link that issues a new search when -->
    <!-- clicked and re-renders this page                          -->
    <dsp:a href="simpleFacet.jsp">

      <!-- re-use the previously saved search request     -->
      <dsp:property bean="QueryFormHandler.searchRequest.requestChainToken
          value="${queryResponse.requestChainToken}"
          name="qfh_rct" priority="30"/>

      <!—- save this request so it can be re-used       -->
      <dsp:property bean="QueryFormHandler.searchRequest.saveRequest"
          value="true" name="fh_sr" priority="30"/>

      <!—- specify that this is a faceted search request    -->
      <dsp:property bean="QueryFormHandler.searchRequest.facetSearchRequest"
          value="true" name="qfh_fsr" priority="31"/>

      <!-- set the facetTrail property on the FacetSearchTools component -->
      <dsp:property bean="FacetSearchTools.facetTrail"
          value="${facetTrail.trailString}" name="qfh_ft" priority="27"/>

      <!-- set the facet trail as a query parameter -->
      <dsp:param name="trail" param="facetTrail.trailString"/>

      <!-— submit the request by invoking the handleSearch method -->
      <!—- on the QueryFormHandler when the link is clicked       -->
```

```
          <dsp:property bean="QueryFormHandler.search" value="submit"
              name="qfh_s_s"/>

          <!-- display each facet value and the number of results -->
          <c:out value="${facetValueNode.facetValue.value}"/>
            ( <c:out value="${facetValueNode.facetValue.matchingDocsCount}"/> )
          <br>
        </dsp:a>

. . .
```

Ordering Facets by Priority

When defining a facet in ATG Merchandising, a merchandiser can optionally specify a priority value for the facet. The facet priority value must be zero (0) or a positive integer. The higher the value is, the lower the priority.

The priority values determine the order in which the facets appear in refinement configurations, which determines the order in which facets are returned by ATG Search. So, for example, if a refinement configuration has a Color facet with priority 0 and a Size facet with priority 3, the Color facet is returned before the Size facet. If the page doesn't do any reordering of the facets, the Color facet is displayed above the Size facet.

If a facet is not assigned a priority value in ATG Merchandising, its priority is set to the value of the `FacetSearchTools` component's `defaultFacetPriority` property. The default value of this property is 100, so that facets whose priority value is null are assigned a low priority. You can change this value in the `FacetSearchTools` component's properties file. For example:

```
defaultFacetPriority=50
```

Filtering Facets

When a user selects a facet value, a new query is issued with this selection applied as a constraint. The search engine returns the results of this query and a new set of facets. By default, the new facet set does not include the other selection values for the selected facet. So, for example, if the selection values for the Color facet are Red, Green, and Blue, and the user selects Red, the facet set returned does not include the Green and Blue selection values, because the green and blue items have already been removed from the search results. Removing these selection values is called *filtering* the facet.

There are a certain cases where filtering the facet may not be the desired behavior:

- If you have a facet based on the `product` item's `ancestorCategories` property, when a user selects a category, you typically will want lower-level category selections to still be available for further navigation. See Skipping Facet Values in the Facet Trail for information about removing the higher-level category selections.
- If your faceted search implementation supports multiple selection values, you may want the unselected facet values to be returned, so the user can further refine on the same facet. This is particularly true if the facet values are combined using Boolean

AND, so you can allow the user to further restrict the current result set. See Supporting Multiple Selection Values for more information.

The search engine filters all facets unless the refinement configuration specifies otherwise. To disable filtering of an individual facet, you add the faceting property to the `filterProperties` array property of the `/atg/commerce/search/refinement/RefinementConfigurationXMLGenerator` component. By default, this property is set to:

```
filterProperties=\
        ancestorCategories.$repositoryId,\
        ancestorCategories.displayName
```

If you have additional facets that you do not want to be filtered, add the faceting properties to this array. For example:

```
filterProperties+=paperSizes
```

Skipping Facet Values in the Facet Trail

As mentioned above, when a faceted search query is issued, ATG Search returns the facets as part of the `QueryRequest.Response` object, and the `FacetSearchTools` component converts the raw facet data into the appropriate format for display. The facets and values returned by ATG Search include ones that the user has already selected, as well as ones that are available for selection.

Most faceted search implementations do not need the already-selected facet values, since these values are typically not displayed, except as part of the facet trail. Therefore, the `FacetSearchTools` component omits these values by default.

This behavior is controlled through the following `FacetSearchTools` properties:

- `skipValuesAlreadyInTrail` -- When set to `true` (the default), omits any facet values that are already present in the facet trail.
- `skipAncestorsToCategoriesInTrail` -- When set to `true` (the default), omits any facet values that represent categories that are ancestors of categories present in the facet trail.

You should leave these properties set to `true` unless your faceted search implementation requires otherwise. If you change the values of these properties, you may see unexpected facet values displayed. For example, if you have a facet based on the `product` item's `ancestorCategories` property, the available facet selections will include the current category and categories above it in the catalog hierarchy, rather than just categories below it in the hierarchy.

Note that the values of these properties have no effect on the facet trail itself. The trail still includes all of the selected values, so the appropriate constraints can be applied when a query is submitted.

Removing Facet Selections

The example in Rendering the Facets creates links for facet selection values. When an user clicks a link, the selection is added to the facet trail and a new search request is issued.

Your pages should also include links for removing facet selections. For example, if a customer chooses a “$100 to $200” selection range for the price facet, only items in that price range are displayed. To display all items regardless of price, the customer can click a link that removes this facet selection.

The following example creates a “remove” link for each facet selection. When a user clicks one of these links, the JSP removes the facet and selection value from the facet trail, along with any dependent facet selections. It also retrieves, updates, and resubmits the current search request.

```
<!-- use CommerceFacetTrailDroplet to transform the  -->
<!-- facet trail string into a FacetTrail object     -->
<dsp:droplet name="CommerceFacetTrailDroplet">

  <!-- expose the facet values as a jstl variable -->
  <dsp:getvalueof param="facetTrail.facetValues" var="facetValues"/>

  <dsp:oparam name="output">

  <c:forEach items="${facetValues}" var="currentFacetValue">
  <!-- Output the facet name and selection value separated by a colon; -->
  <!-- skip over the facet value containing the search question text   -->
  <c:set var="srchFacetLabel" value="SRCH"/>
  <c:if test="${currentFacetValue.facet.id != srchFacetLabel}">
  <c:out value="${currentFacetValue.facet.label}:"/>
  <c:out value="${currentFacetValue.value}"/>

    <!-- create a remove link for each facet value; use the       -->
    <!-- CommerceFacetTrailDroplet to construct a new facet trail -->
    <dsp:droplet name="CommerceFacetTrailDroplet">
      <dsp:setvalue param="trail" value="${trailString}"/>
      <dsp:setvalue param="removeFacet" value="${currentFacetValue}"/>

      <!-- expose the facetTrail string as a jstl variable -->
      <dsp:getvalueof param="facetTrail" var="facetTrail"/>

      <dsp:oparam name="output">

        <!-- create a link back to this page that submits a     -->
        <!-- search request using the updated facet trail       -->
        <dsp:a href="simpleFacet.jsp">
          <dsp:property bean="QueryFormHandler.searchRequest.requestChainToken"
              value="${queryResponse.requestChainToken}"
              name="qfh_rct" priority="30"/>
          <dsp:property bean="QueryFormHandler.searchRequest.saveRequest"
              value="true" name="fh_sr" priority="30"/>
          <dsp:property bean="QueryFormHandler.searchRequest.facetSearchRequest"
              value="true" name="qfh_fsr" priority="31"/>
          <dsp:property bean="FacetSearchTools.facetTrail"
              value="${facetTrail.trailString}" name="qfh_ft" priority="27"/>
          <dsp:param name="trail" param="facetTrail.trailString"/>
          <dsp:property bean="QueryFormHandler.search" value="submit"
              name="qfh_s_s"/>
          remove
        </dsp:a>
      </dsp:oparam>
    </dsp:droplet>
  </c:if>
  </c:forEach>
  </dsp:oparam>
</dsp:droplet>
```

Rendering Multiple Selection Values

If your site supports multiple selection values for facets, you need to code your JSPs accordingly. For example, consider these lines that display a facet and its selection value:

```
<c:out value="${currentFacetValue.facet.label}:"/>
<c:out value=" ${currentFacetValue.value}"/>
```

This code assumes that the `value` property of a `FacetValue` object is a single value. If a facet actually has multiple selection values, the code will display only one value. So, for example, if the current selections for the Color facet are Blue, Green, and Red, this code would display:

```
Color: Blue
```

To display all of the selection values, your JSP code should treat the `FacetValue.value` property as an array. For example:

```
<c:out value="${currentFacetValue.facet.label}:"/>
<c:forEach items="${currentFacetValue.value}" var="currentMultiValue">
  <c:out value=" ${currentMultiValue}"/>
</c:forEach>
```

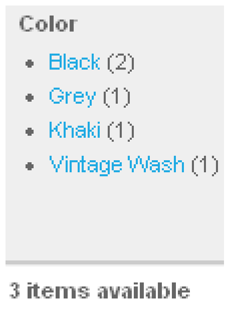
For the Color facet, this will display:

```
Color: Blue Green Red
```

Note that code for handling multiple selection values will work even when there is only a single selection value for a facet, so it is not necessary to write separate code to handle each case.

About Refinement Counts

Pages that display facets typically include refinement counts, which show the number of items in each selection. For example, the selection values displayed for a color facet might look like this:


Color

- Black (2)
- Grey (1)
- Khaki (1)
- Vintage Wash (1)

3 items available

The values in parentheses are the refinement counts. If the user clicks on the Black selection value, for example, 2 items will be displayed.

Note that these refinement counts may not always match the number of search results. For example, in this figure, the number of search results is 3, but if you add up the refinement counts, you get 5. This is because some search results appear in multiple facet selections, because the index is based on products but SKU properties are used for faceting. In this example, one of the products is available in Black, Grey, and Khaki, so it is included in the counts for all three of those colors.

There are various other situations where refinement counts won't match the number of search results. For example, if you use SKU-based indexing but group results by product, the refinement counts will reflect the number of SKUs returned, but the items displayed when a facet value is selected will be products. To make the refinement counts reflect the number of products, set the following property on the `/atg/commerce/search/catalog/QueryRequest` component:

```
refineCount=group
```

For more information about grouping results by product, see Setting Grouping Options.

Incorporating Search Text as a Facet

Using a search form created with the `QueryFormHandler` class, you can constrain the set of items accessed through facet selections to ones that also match the search query text. For example, suppose a site visitor at a clothing store begins by searching for "belt," and then chooses the "Brown" selection value for the color facet. The site would now display only brown belts, not all brown items in the store.

To enable this behavior in your pages, Faceted Search allows you to use a special SRCH facet whose selection value is the search term entered by the site visitor. In this example, the facet trail String would look something like this:

```
SRCH:belt:12:Brown
```

Constructing the Facet Trail String

When a customer submits a text query through the `QueryFormHandler`, the search text is stored in the `question` property of the `QueryRequest` component, and returned (with possible modifications) in the question property of the QueryRequest.Response object. Submitting search text creates a new search request with an empty facet trail. When the customer makes a facet selection, it is added to the facet trail that is included in the subsequent search request.

The following JSP code implements the logic for building up the facet trail. If the facet trail is empty and the question property of the response object is not empty, this means that the user has entered search text as the most recent search request. In this case, the code sets the facet trail to SRCH:*text* plus whatever facet selection the customer makes.

If, however, there is already a non-empty facet trail or the question property of the response object is empty, this means that either the SRCH facet has already been added to trail, or there is no SRCH facet (because there is no search text). So in either of these cases only the facet selection the customer makes is added to the trail.

```
<dsp:droplet name="CommerceFacetTrailDroplet">
  <dsp:setvalue param="trail" value="${trailString}"/>

  <c:if test="${ empty trailString and ! empty queryResponse.question }">
    <dsp:setvalue param="addFacet"
      value="SRCH:${queryResponse.question}:${facetValueNode.facetValue}"/>
  </c:if>
  <c:if test="${ ! empty trailString or empty queryResponse.question }">
    <dsp:setvalue param="addFacet" value="${facetValueNode.facetValue}"/>
  </c:if>

. . .
```

Selecting the Refinement Configuration

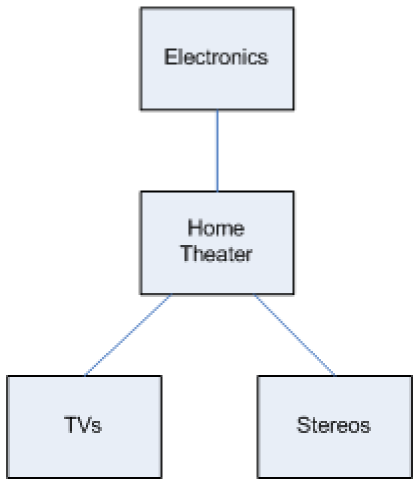
When a customer submits a text query, the QueryFormHandler has no information for selecting a refinement configuration. In this situation, ATG Search can determine the refinement configuration based on the items that are returned from the query. This behavior is configured in the /atg/commerce/search/catalog/QueryRequest component through the following settings:

```
refineConfig=$map
refineConfigMapProperty=ancestorCategories.catalogSpecificId
```

Setting the refineConfig property to $map instructs ATG Search to select the refinement configuration by finding a metadata property value common to all of the results; refineConfigMapProperty specifies which metadata property to use. The value of the specified property, ancestorCategories.catalogSpecificId, is an array of the catalog-specific category IDs of a product's ancestor categories. (Each ID is formed by combining the category ID with the catalog ID.) The value of ancestorCategories.catalogSpecificId is determined by the /atg/commerce/search/CustomCatalogCategoriesPropertyAccessor component, which is a property accessor of class atg.commerce.search.producer.CustomCatalogCategoriesPropertyAccessor.

These settings specify that ATG Search should use the refinement configuration associated with the lowest-level catalog-specific ancestor category that is common to all of the returned items. This refinement configuration includes the global facets plus any facets specific to that category. If there is no ancestor category common to all of the results, a refinement configuration that includes only global facets is used.

For example, suppose your site has several root categories, including Electronics, Shoes, Books, etc. The hierarchy of the Electronics category looks like this:



Let's say a customer searches for "Acme". The search results consist only of TVs and stereos manufactured by Acme Audiovisual. So the lowest-level ancestor category that is common to all of the returned items is Home Theater. ATG Search uses the refinement configuration associated with this category.

Another customer searches for "Cogswell". This time, the results consist of TVs and stereos manufactured by Cogswell Inc., but also books written by an author named Russell Cogswell. The results therefore do not have a common ancestor category, so ATG Search uses only the global facets.

Formatting Facet Values

In many cases, when you display a facet value on a page, it is desirable to reformat the value, because the format used to store the values in the index is not optimized for display purposes. For example:

- Boolean values are stored in the index as 0 (false) or 1 (true).
- Dates are encoded as long integers.
- Price data is stored as raw numbers with no currency symbol or other formatting. For example, the value representing $87,109.00 might be stored in the index as 87109.0.

To reformat these values for displaying on pages, the ATG platform includes a servlet bean, `atg.repository.search.refinement.RefinementValueDroplet`. This servlet bean takes as input a facet selection value and the repository ID of the `refinementElement` that represents the facet, and outputs the value in a more human-readable form.

ATG Commerce includes a component of this class, which you can use in your pages like this:

```
<dsp:droplet name="/atg/commerce/search/refinement/RefinementValueDroplet">
  <dsp:param name="refinementId" value="${facetHolder.facet.id}"/>
  <dsp:param name="refinementValue" value="${facetValueNode.facetValue.value}"/>
  <dsp:oparam name="output">
    <dsp:valueof param="displayValue"/>
  </dsp:oparam>
</dsp:droplet>
```

To perform the formatting, `RefinementValueDroplet` uses a class that implements the `atg.repository.search.MetaPropertyValueFormatter` interface. By default, this class is `atg.repository.search.DefaultMetaPropertyValueFormatter`. The `RefinementValueDroplet` component has a `defaultValueFormatter` property that points to a component of this class, `/atg/commerce/search/refinement/DefaultMetaPropertyValueFormatter`.

`DefaultMetaPropertyValueFormatter` can reformat a variety of data types:

- For a repository ID property (e.g., `childSKUs.$repositoryId`), it returns the item display name.
- For a date property (e.g., `creationDate`), it returns a locale-specific date string.
- For an enumerated property (e.g. `stockAvailabilityStatus`), it returns the display name of the enumerated value.
- For a Boolean property (e.g., `onSale`) it looks in a resource bundle for a key of the format *`property-name`*`_0` (for `false`) or *`property-name`*`_1` (for `true`), and returns the string associated with that key. For example, for an `onSale` property, the keys would be `onSale_0` and `onSale_1`. If the key is not found, the value is returned unchanged. The resource bundle is specified by the `resourceBundle` property of the `DefaultMetaPropertyValueFormatter` component.

You can write a custom implementation of the `MetaPropertyValueFormatter` interface to use for specific faceting properties. ATG Commerce includes one such class, `atg.commerce.search.PriceMetaPropertyValueFormatter`, for formatting price data. It also includes a component of this class, `/atg/commerce/search/PriceMetaPropertyValueFormatter`.

To specify a custom formatter for a property, you create a component of your class and then register the component with the `/atg/search/repository/MetaPropertyValueFormatterRegistry` component. `MetaPropertyValueFormatterRegistry` has a `valueFormatterMap` property which maps property names to their associated formatter components. For example, to specify that the `PriceMetaPropertyValueFormatter` should be applied to the `price` property, ATG Commerce adds this configuration:

```
valueFormatterMap+=\
   price=/atg/commerce/search/PriceMetaPropertyValueFormatter
```

When it formats a value, `RefinementValueDroplet` first checks the `MetaPropertyValueFormatterRegistry` to see if there is a custom `MetaPropertyValueFormatter` registered for the property name, and if there is, uses that formatter. If there no custom formatter registered for the property, it uses the `DefaultMetaPropertyValueFormatter`.

11 Search Merchandising

The Search Merchandising feature enables commerce sites to customize search results based on site initiatives and customer purchasing patterns. A merchandiser uses ATG Merchandising to create *search configurations*, which are sets of rules that determine the order of search results and which items are excluded from the results.

Search configurations are stored in the `RefinementRepository` (along with the refinement configurations used for Faceted Search), and are included by ATG Search when the product catalog is indexed. When a site visitor issues a search query, ATG platform components determine which search configuration to use, and include this information with the query. When ATG Search returns the results from the query, it applies the rules in the search configuration to those results.

This chapter describes the query-side services involved in Search Merchandising. It includes the following sections:

For more information about Search Merchandising, including details about creating search configurations, see the *ATG Merchandising Guide for Business Users* and the *ATG Merchandising Administration Guide*.

Determining the Search Configuration for a Query

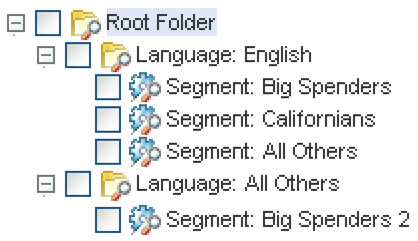
When a visitor enters a search term on a site that uses Search Merchandising, the software determines which search configuration to apply, and includes this information in the query it sends to ATG Search. The logic used to select the search configuration is based on the tree structure created in ATG Merchandising. This structure can take into account three dimensions: user segment, site, and locale (referred to in ATG Merchandising as language).

The dimension tree structure is stored as repository items in the `RefinementRepository`. When the product catalog is deployed from the ATG Merchandising environment to the target site, this repository is deployed as well.

The dimension services that create the tree structure in ATG Merchandising, `/atg/search/config/LanguageDimensionService`, `/atg/commerce/search/config/SiteDimensionService`, and `/atg/commerce/search/config/SegmentDimensionService`, are also present on the search client environment. When a site visitor submits a query entered in a search form, the form handler invokes the

`/atg/commerce/search/config/SearchConfigNameService` component, which uses these services to traverse the decision tree based on the visitor's language (locale) , the user segments he or she is a member of, and the current site. It proceeds through the tree until it finds the first search configuration that matches these values. It then adds a reference to this search configuration to the query. The search configuration rules are applied to the results returned. If there is no matching search configuration, then no search configuration rules are applied.

For example, suppose the dimension tree created in ATG Merchandising looks like this:



In this example, English is the only language defined, and there are only two segments available, Big Spenders and Californians, and site is not used as a dimension.

To determine the search configuration to use, the software would proceed like this:

- Determine the visitor's language, as described in Determining the Language.
- If the language is English:
    - Determine if the visitor is a member of the Big Spenders segment. If so, use the Segment: Big Spenders search configuration.
    - If the visitor is not a member of the Big Spenders segment, determine if the visitor is a member of the Californians segment. If so, use the Segment: Californians search configuration.
    - If the visitor is not a member of the Californians segment, use the Segment: All Others search configuration.
- If the language is not English:
    - Determine if the visitor is a member of the Big Spenders segment. If so, use the Segment: Big Spenders 2 search configuration.
    - If the visitor is not a member of the Big Spenders segment, do not use any search configuration.

Some things to note in this example:

- The ordering of the items in the tree is important. If the visitor's language is English, and he or she is a member of both the Big Spenders and the Californians segments, the Segment: Big Spenders search configuration is selected, because its position in the tree is above the Segment: Californians search configuration.
- In a group of dimension folders or search configurations at the same position in the hierarchy, the folder or search configuration whose dimension value is All Others is

always the last item in that group of folders or search configurations. In the example above, the Language: All Others folder comes after Language: English, and the Segment: All Others search configuration comes after Segment: Big Spenders and Segment: Californians. ATG Merchandising enforces this ordering to ensure that a folder or search configuration whose dimension value is All Others is used only if the visitor's value for that dimension doesn't match any other folder or configuration.

Determining the Language

When a merchandiser defines a search configuration in the ATG Merchandising UI, he or she selects a language from a preconfigured list. Though the term "language" is used throughout the Merchandising UI, the values selected actually represent Java locales. So, for example, the list of available languages might include both British English (representing the locale `en_GB`) and US English (representing the locale `en_US`).

When a site visitor enters a search query, the software determines the visitor's locale, and uses this value in the process of selecting a search configuration. Different sites may use different logic for determining a visitor's locale. The following steps describe one common approach:

1. If the URL in the request includes a query parameter that specifies a locale, use that locale.
2. If the locale is not specified in the URL, examine the current profile's `locale` property. If this property is set (typically the case only if the user is logged in), use the value of that property.
3. If the `locale` property of the profile is not set, examine the HTTP headers of the request for a locale setting. This is set by the browser based either on a preference setting in the browser itself, or on a value derived from an operating-system setting.

Handling Redirects

A merchandiser can create a rule in ATG Merchandising that specifies a URL to redirect to if the search query contains certain terms. For example, if a certain product is highly anticipated but has not yet been released, the merchandiser may want to create a redirection rule specifying that if a site visitor searches for this product, the site should display a page where the visitor can sign up to be notified when the product is available. If this redirection rule is executed, the results returned by ATG Search are not displayed.

The search form handlers have an `redirectEnabled` property that specifies whether the form handler should check the search results for a redirect URL. If this property is set to `true`, the form handler examines the search results for a redirect URL, and if it finds one, displays the page specified by that URL. If this property is set to `false`, the form handler ignores any redirect URL and displays the results returned by ATG Search. The `redirectEnabled` property is set to `false` by default in the ATG Merchandising Search Testing environment, but is set to `true` by default otherwise.

12 Recording Events for Reporting

If your ATG installation includes ATG Customer Intelligence (ACI), you can use it to generate Commerce Search reports. In particular, you can generate reports that associate search terms with items that are viewed or purchased.

To do this, your site must record “click-through” events. These occur when a customer clicks on a product or SKU returned by a search, to view it or purchase it. The recording of these events works like this:

- For each search result, the `GetClickThroughId` servlet bean generates a click-through ID, which you append to the URL for that result using a query parameter. The servlet bean also adds the result document to a cache.
- When a customer clicks a link to view a search result, the `SearchClickThroughServlet` examines the request URL, finds the click-through ID, and uses it to look up the document in the cache. If it finds the document, the servlet fires a JMS event containing the search request and response objects and the selected document. This event is logged to be used for reporting.

This chapter describes how to code your pages and configure your site to log the events used for Commerce Search reporting. See the *ATG Commerce Programming Guide* for information about loading this data into ACI. For more information about reporting in general, see the ACI documentation.

Using the GetClickThroughId Servlet Bean

The `/atg/search/droplet/GetClickThroughId` servlet bean is typically used in a loop that renders a list of search results. For each result, it adds the item to a cache, and generates a click-through ID to be included in the URL for viewing that item. The click-through ID consists of a query identifier and a document identifier, separated by a delimiter.

You use this servlet bean in pages that render a listing of search results. For example, the following JSP code creates a hyperlink to a product page and appends the `searchClickId` query parameter to the URL:

```
<dsp:droplet name="/atg/search/droplet/GetClickThroughId">
  <dsp:param name="result" value="${searchResult}" />
   <dsp:oparam name="output">
     <dsp:a href="/myapp/en/product.jsp">
       <dsp:param name="searchClickId" param="${searchClickId}"
```

```
      </dsp:a>
   </dsp:oparam>
</dsp:droplet>
```

The resulting URL looks something like this:

```
http://www.mycompany.com/myapp/en/product.jsp?searchClickId=0000020003,25
```

For more information, see GetClickThroughId.

Configuring the Cache

`GetClickThroughId` uses a session-based cache component, `/atg/search/cache/SearchQueryCache`, to store results returned by ATG Search. This cache uses a Least Recently Used (LRU) storage algorithm. This means that if the cache is full, whenever a new item is added to the cache, the oldest item is discarded.

Discarding items can make reporting less accurate, but prevents the cache from becoming too large and consuming too much memory. To optimize the tradeoff between accuracy and resource use, you can set the following properties of the `SearchQueryCache` component:

- `queryCount` -- Specifies the maximum number of search request/response objects to store. Default is 10. To specify no maximum, set this value to -1.
- `documentCount` -- Specifies the maximum number of documents to store per search request/response. A document is stored in the cache if the page of results it is on is displayed. So, for example, if 10 results are displayed per page, and the user views the first 3 pages of results, 30 documents are stored in the cache for that response. Default is 1000. To specify no maximum, set this value to -1.

Note that the LRU caching algorithm is applied independently to the number of search queries and the number of documents per search query. So, for example, if `queryCount` is set to 5, and a user enters 6 search queries, the results from the first query are discarded from the cache, regardless of how many documents are returned for the other stored queries. Similarly, if `documentCount` is set to 100, and the user pages through the first 12 pages of results for a query (with 10 results displayed per page), then the cache will store documents 21 through 120, and the first 20 documents will be discarded; but none of the other stored queries will be affected.

Configuring the SearchClickThroughServlet

The `/atg/search/servlet/pipeline/SearchClickThroughServlet` is inserted into the DAF servlet pipeline after the `ProfileRequestServlet`. When a user clicks a link to view a search result, `SearchClickThroughServlet` reads the click-through ID from the request URL and looks up the document in the `SearchQueryCache`. If it finds the document, it triggers a `SearchClickThroughMessage` JMS event, which is logged for loading into the ACI Data Warehouse.

To configure this servlet, set the following properties:

enabled
If `true`, the servlet processes the request. Default is `false`.

searchClickIdQueryArgs
An array of the query arguments to read to find the click-through ID for a viewed item. One of the values in this array must match the name of the output parameter set by `GetClickThroughId`. Default is `searchClickId`.

Limiting the Pages to Examine

By default, this servlet examines all URLs to look for click-through IDs. This process can be inefficient, because only product detail pages will typically have these IDs. Therefore `SearchClickThroughServlet` has a `clickThroughPages` property that you can use to limit the pages to examine. This property is an array of URLs; these URLs can include asterisk (`*`) characters as wildcards. If `clickThroughPages` is not null, `SearchClickThroughServlet` will examine only the URLs that match one of the `clickThroughPages` entries. For example, you could set `clickThroughPages` to:

```
/myapp/*/product*.jsp,\
/myapp/*/sku*.jsp
```

Appendix A: Commerce Search Servlet Beans

This appendix provides reference entries for the following Commerce Search servlet beans:

- **CommerceFacetTrailDroplet**
- **GetClickThroughId**
- **RefinementValueDroplet**

CommerceFacetTrailDroplet

Class Name	`atg.commerce.search.refinement.CommerceFacetTrailDroplet`
Component	`/atg/commerce/search/refinement/CommerceFacetTrailDroplet`

This servlet bean takes as input a String representing a facet trail, plus additional input parameters specifying modifications to the facet trail. It outputs the modified facet trail as a `FacetTrail` object, which can then be rendered on the page or used to construct a subsequent search request.

The input parameters can be set explicitly or they can be set by the page's URL query parameters. For example, when a customer clicks a link for a selection value, the query parameter corresponding to the servlet bean's `addFacet` parameter can be set to this selection value. When the new page is displayed, the chosen value will appear at the end of the facet trail. Similarly, another link could be used to remove a selection value or range from the facet trail.

Properties

The following table describes the properties of the `CommerceFacetTrailDroplet` component and their default settings. Note that each property whose name ends with "ParameterName" specifies the name of the query parameter that supplies the value to use for the corresponding input parameter if the input parameter is not supplied.

Property	Description
`facetManager`	Specifies the component used to retrieve items from the refinement repository. Default: `/atg/commerce/search/refinement/CommerceFacetManager`
`facetTrailSeparator`	The character used as a separator between the facets and the faceting property values in the facet trail. Default: the colon character (`:`)
`lastRangeValueIndicator`	A String appended to a selection range in the facet trail if the range is the last one for a particular facet. Default: LAST
`valueIndicatorSeparator`	Separator placed between a selection range and the `lastRangeValueIndicator` String if the range is the last one for a particular facet. Default: the vertical bar character (`|`)
`categoryRefineConfigPropertyName`	The name of the property of the category repository item that contains a reference to the refinement configuration for the category. Default: `refineConfig`
`trailParameterName`	The name of the query parameter that specifies the value to use for the `trail` input parameter, if the input parameter is not supplied. Default: `trail`
`addFacetParameterName`	The name of the query parameter that specifies the value to use for the `addFacet` input parameter, if the input parameter is not supplied. Default: `addFacet`
`removeFacetParameterName`	The name of the query parameter that specifies the value to use for the `removeFacet` input parameter, if the input parameter is not supplied. Default: `removeFacet`
`removeAllFacetsParameterName`	The name of the query parameter that specifies the value to use for the `removeAllFacets` input parameter, if the input parameter is not supplied. Default: `removeAllFacets`
`removeFacetTypeParameterName`	The name of the query parameter that specifies the value to use for the `removeFacetType` input parameter, if the input parameter is not supplied. Default: `removeFacetType`

Input Parameters

trail
String that represents the current facet trail. This parameter's value is typically specified through a query parameter in the URL for the page. The name of the query parameter that sets the value of this input parameter is configured through the `trailParameterName` property.

refineConfig
The `refineConfig` repository item to use for querying ATG Search. If this value is not specified, the refinement configuration will be chosen automatically.

addFacet
String that represents an entry (consisting of a facet and an associated selection value or range) to add to the facet trail. This parameter's value is typically specified through a query parameter in the URL for the page. The name of the query parameter that sets the value of this input parameter is configured through the `addFacetParameterName` property.

removeFacet
String that represents an entry to remove from the trail. This parameter's value is typically specified through a query parameter in the URL for the page. The name of the query parameter that sets the value of this input parameter is configured through the `removeFacetParameterName` property.

removeAllFacets
If this parameter is set to `true`, the facet trail is cleared. This parameter's value is typically specified through a query parameter in the URL for the page. The name of the query parameter that sets the value of this input parameter is configured through the `removeAllFacetsParameterName` property.

removeFacetType
The item ID of a refinement element repository item (i.e., a facet); specifies that all facet values or ranges for this facet should be removed from the facet trail. This parameter's value is typically specified through a query parameter in the URL for the page. The name of the query parameter that sets the value of this input parameter is configured through the `removeFacetTypeParameterName` property.

Output Parameters

facetTrail
The `FacetTrail` object generated from the input or query parameters.

errorMessage
The message generated if an error occurs when creating the `FacetTrail` object.

Open Parameters

output
This open parameter is rendered if no errors occur when creating the `FacetTrail` object.

error
This open parameter is rendered if any errors occur when creating the `FacetTrail` object.

Examples

For examples of using the `CommerceFacetTrailDroplet`, see the Faceted Search chapter.

GetClickThroughId

Class Name	`atg.search.cache.droplet.GetClickThroughId`
Component	`/atg/search/droplet/GetClickThroughId`

This servlet bean takes as input a single search result, stores it in a document cache, and returns a click-through ID associated with the result. This ID can subsequently be used by the `SearchClickThroughServlet` to retrieve the result from the cache.

`GetClickThroughId` is typically used in a loop that renders a list of search results as hyperlinks to pages displaying those items. For each result, the click-through ID is set as the value of a query parameter that is appended to the URL.

Properties

The following table describes the properties of the `GetClickThroughId` component and their default settings.

Property	Description
`searchResultParameter`	Name of the input parameter that specifies the search result. Default: `result`
`searchClickIdParameter`	Name of the output parameter that holds the click-through ID. Default: `searchClickId`
`searchQueryCachePath`	String specifying the Nucleus pathname of the cache component. Default: `/atg/search/cache/SearchQueryCache`
`searchClickIdDelimiter`	Delimiter separating the query identifier and the document identifier in the click-through ID. Default: the comma character (`,`)

Input Parameter

result
The current search result.

Output Parameter

searchClickId
The click-though ID for retrieving the current search result from the cache.

Open Parameter

output
The open parameter for rendering the click-through ID.

Example

```
<dsp:droplet name="/atg/search/droplet/GetClickThroughId">
  <dsp:param name="result" value="${searchResult}" />
   <dsp:oparam name="output">
     <dsp:a href="/myapp/en/product.jsp">
       <dsp:param name="searchClickId" param="${searchClickId}"
     </dsp:a>
   </dsp:oparam>
</dsp:droplet>
```

RefinementValueDroplet

Class Name	`atg.repository.search.refinement.RefinementValueDroplet`
Component	`/atg/commerce/search/refinement/RefinementValueDroplet`

This servlet bean takes as input a facet selection value and the repository ID of the `refinementElement` that represents the facet, and outputs the value in a more human-readable form. The value is formatted by the `/atg/commerce/search/refinement/DefaultMetaPropertyValueFormatter` component, which is specified by the `defaultValueFormatter` property of the servlet bean.

The `DefaultMetaPropertyValueFormatter` formats the following data types:

- For a repository ID property (e.g., `childSKUs.$repositoryId`), it returns the item display name.
- For a date property (e.g., `creationDate`), it returns a locale-specific date string.
- For an enumerated property (e.g., `stockAvailabilityStatus`), it returns the display name of the enumerated value.
- For a Boolean property (e.g., `onSale`) it looks in a resource bundle for a key of the format *`property-name`*`_0` (for `false`) or *`property-name`*`_1` (for `true`), and returns the string associated with that key. For example, for an `onSale` property, the keys would be `onSale_0` and `onSale_1`. If the key is not found, the value is returned unchanged. The resource bundle is specified by the `resourceBundle` property of the `DefaultMetaPropertyValueFormatter` component.

You can also write custom formatters for specific properties. For more information, see the Faceted Search chapter.

Input Parameters

refinementId
The repository ID of the `refinementElement` that represents the facet in the refinement repository.

refinementValue
The facet value returned by the search engine.

Output Parameter

displayValue
The formatted value.

Open Parameters

output
This open parameter is rendered if no errors occur when formatting the value.

error
This open parameter is rendered if any errors occur when formatting the value.

Example

```
<dsp:droplet name="/atg/commerce/search/refinement/RefinementValueDroplet">
  <dsp:param name="refinementId" value="${facetHolder.facet.id}"/>
  <dsp:param name="refinementValue" value="${facetValueNode.facetValue.value}"/>
  <dsp:oparam name="output">
    <dsp:valueof param="displayValue"/>
  </dsp:oparam>
</dsp:droplet>
```

Appendix B: Search XML Reference

This appendix provides information on the XML used by ATG Search queries and responses. This information can be useful for troubleshooting.

answer

The `answer` element is the root response element for a `query` request. Based on the `query` string entered by the end-user and the search environment's configuration, ATG Search returns a response containing the results retrieved. The results can then be displayed in the user interface or subjected to further processing. The search results include:

- The matching statement
- The index item that contains that statement
- Secondary information about the results, such as where they fall in the category taxonomy
- Feedback about the query, such as spelling suggestions or refinements

Each document object contains the list of document sets and categories of which it is a member. This information includes the size of the document set, and for categories, the relevance of the document to that category.

Attribute	**Description**
`contentID`	Content ID of the index item.
`QUID`	Value of the query's `QUID` attribute.
`sorting`	Value of the query's `sorting` attribute.
`sortProp`	For grouping by property, the type, name, and default value.
`docSetSort`	Value of the query's `docSetSort` attribute.
`mode`	Value of the query's `mode` attribute.
`strategy`	Value of the query's `strategy` attribute.
`debug`	Value of the query's `debug` attribute.

`refineDebug`	Value of the query's `refineDebug` attribute.
`refineMax`	Value of the query's `refineMax` attribute.
`refineMin`	Value of the query's `refineMin` attribute.
`refineTop`	Value of the query's `refineTop` attribute.
`autospell`	Value of the query's `autospell` attribute.
`highlight`	Value of the query's `highlight` attribute.
`minScore`	Value of the query's `minScore` attribute.
`pageNum`	Value of the query's `pageNum` attribute.
`pageSize`	Value of the query's `pageSize` attribute.
`docSort`	Value of the query's `docSort` attribute.
`docSortPred`	Value of the query's `docSortPred` attribute.
`docSortProp`	Value of the query's `docSortProps` attribute.
`docSortPropDefault`	Value of the query's `docSortPropDefault` attribute.
`docSortOrder`	Value of the query's `docSortOrder` attribute.
`docSortCase`	Value of the query's `docSortCase` attribute.
`relQuestSettings`	Value of the query's `relQuestSettings` attribute.
`responseNumberSettings`	Value of the query's `responseNumberSettings` attribute.
`topicSettings`	Reserved for future use.
`responseCount`	Total number of results returned.
`groupCount`	Total number of groups returned.
`docCand`	Number of document candidates considered during retrieval.
`docMax`	Estimate of how many documents are retrievable using the current query.
`docMin`	Minimum number of documents that could be returned, if all query terms were in the same documents.
`ansCand`	Number of sentence candidates considered during retrieval.
`ansMax`	Estimate of how many sentences are retrievable with the current query.
`ansMin`	Minimum number of sentences that could be considered, if all query terms were in the same sentence.
`ansPool`	Number of sentence results before grouping and paging.

pageOffsetInfo	Value of the query's pageOffsetInfo attribute.
maxRelatedSets	Value of the query's maxRelatedSets attribute.
responseTimeMs	Amount of time taken to process the query.

The answer element contains the following child elements:

- categories
- debug
- documentSets
- parserOptions
- priorInput
- queryAction
- queryRule
- queryTerms
- question
- response
- responseTree
- refinements
- spelling
- startCategory
- userquestion
- weightedProps

categories

This element is a child of the answer element, and contains category child elements. For this element, a category is equivalent to a document set.

category

The category provides document count information, and contains query refinements, documents, and additional categories. For this element, a category is equivalent to a document set.

Attributes	Description
`_path`	Path name of document set.
`label`	Optional display label for document set.
`_id`	Topic ID, if the document set represents a topic.
`nUniqueDocuments`	Total number of unique documents under this document set and all descendents.
`nTotalDocuments`	Total number of documents under this document set and all descendents.
`nDisplayedDocuments`	Total documents returned under this document set, with respect to constraints and settings.
`nDocuments`	Total documents immediately under this document set.
`nChildren`	Number of child sets immediately under this document set.
`nTotalChildren`	Total number of descendent sets under this document set and its child sets.
`nDisplayedChildren`	Number of returned descendents with respect to constraints and settings.

Child elements of a `category` are collections of simple single elements:

- `refinements`—Has `refinement` elements as children.
- `documents`—Has `document` elements as children.

A `category` can also contain additional `category` elements.

context

The `context` element is a child of the `parserOptions` element. It has no child elements.

ATG Search contains a large general-purpose dictionary which represents all of the knowledge about a language that it processes. The dictionaries for each language can be loaded separately, or in combinations. For each language, the dictionary contains index terms (also called stems), part-of-speech data, syntactic and semantic features, morphological rules, compound and phrase data, term normalization data, term weights, thesaurus entries, text patterns, and various other pieces of data.

The adaptor components are extensions to the general purpose dictionary. Adaptors typically reflect domains, such as financial, computer, and manufacturing. Each domain requires specialized information in the dictionary, which may or may not be applicable to other domains. Administrators determine which adaptors are loaded (see the *Term Dictionaries* chapter of the *ATG Search Administration Guide*). Adaptors include the following:

- index terms

- compound terms
- term normalizations
- additional thesaurus entries
- modifications to thesaurus entries in the core dictionary

ATG Search offers the following adaptors for English:

- aerospace
- airlines
- apparel
- appliances
- automotive
- business
- computer
- cooking
- crafts
- ecommerce
- financial
- food
- healthcare
- hotels
- housewares
- HR
- insurance
- jewelry
- legal
- manufacturing
- media
- personal_care
- pets
- sports_outdoors
- telecommunications
- tools
- toys
- yard_garden

During indexing, the adaptors are loaded based on the languages selected in Search Administration (see the *ATG Search Administration Guide*).

To use a dictionary adapter, the adapter must be loaded when the content is indexed, and it must be included as a context in `parserOptions`. For example, to use the healthcare adaptor, it must be selected as a pre-indexing customization in the search project via Search Administration. Additionally, each query must include the `context` in its `parserOptions`:

```
<context>healthcare</context>
```

Multiple instances of this element are allowed. If an adapter is specified at index time but not as a query-time context, the loaded adapter is not used for the query.

Adaptors can affect which strings are considered tokens for indexing purposes, and this effect is independent of the context setting at query time. Therefore, you should usually include all adapters as contexts in the query. The exception would be if a site has mixed content indexed in a single partition; then, different adaptors may be enabled at query time, depending on the context of the query. This allows different thesaurus entries to be used depending on the query context; for example, in one context for healthcare, and in another context for pets.

debug

This element contains debugging information. It is a child of the `answer` element and has no child elements.

document

For `query` requests, the `document` element represents a single document within the `response` element.

Attributes	Description
`contextID`	View context information in the form: `hdoc:`*`start-final`*, *`start-final`*… Start and final refer to the text offset positions of a matching statement.
`hdoc`	Document identifier.
`goto`	Jump to off-set of this result in the document text.
`size`	Size of source of document.
`docset`	Physical document set of the document.
`type`	Major format of document.

`lang`	Language of document.
`filename`	File name (excluding path).
`relevance`	Relevance score of document.

The document element has the following child elements:

- `documentSets`—See the `documentSets` element.
- `title`—Title of document. No child elements.
- `summary`—Summary of the document. No child elements.
- `_url`—URL of the document. No child elements.
- `timestamp`—Last modified date and time of the document. No child elements.
- `question`—See the `question` element.
- `answer`—See the `answer` element.
- `properties`—List of metadata properties for this document, contained by `meta` child elements.

documentSets

This element consists of a constraint expression. It is a child of the `query` element. See Document Set Constraints in the Constraining Queries chapter for information.

envName

The `envName` element is a child of the `query` element. It has no child elements.

This element contains the name of the search environment against which the query is directed, if one has been specified. See the *ATG Search Administration Guide* for information on environments.

expandedStemming

The `expandedStemming` element is a child of the `parserOptions` element. It has no child elements.

ATG Search performs morphological analysis on both indexed content and input queries. For most word forms, a single index term is derived; however, for some forms, multiple index terms are possible. For example, the form *spoke* is both a noun root and a past tense form of the verb root *speak*.

During indexing, if multiple index terms are possible, ATG Search chooses the most common term (as defined in the dictionary). At query time, ATG Search uses all root terms for each query term. Part-of-speech tagging can help determine if the terms should be limited, such as choosing the noun *spoke* for a phrase like *the spoke*, but is not always able to correctly interpret queries. This tag determines what sort of stem expansion is used:

```
<expandedStemming>val</expandedStemming>
```

If *val* is `false`, expansion is performed only on a single index term. If *val* is `all`, all index terms are used during expansion. A value of `untagged` means that query terms that could not be part-of-speech tagged use all index terms for expansion.

language

The `language` element is a child of the `parserOptions` element. It has no child elements.

The query language determines which dictionary to use for processing. Only languages that have been loaded when the content was indexed are valid. The format is:

```
<language>lang</language>
```

The *lang* value is the name of any valid language, and defaults to English.

parserOptions

The `parserOptions` element is a child of the `query` element. It contains the following child elements:

- `language`
- `targetLanguage`
- `spellchecker`
- `expandedStemming`
- `wildcardMax`
- `securityRole`
- `context`
- `topicMaximum`
- `spellSplitWords`

ATG Search uses its natural language components to process the query during search. The natural language components provide options that affect this processing. This section describes the major options, which are passed in as XML elements in the query as part of the `parserOptions` element.

It is important to keep in mind that any parser options you set in the query XML must agree with the text processing option set selections you have made in Search Administration. For example, if you have indexed content in French, and the `language` and `targetLanguage` settings specify English, the search will return no results.

See the *ATG Search Administration Guide* for information on creating text processing option sets through Search Administration.

priorinput

This element contains prior or secondary user input. It is a child of the `query` element and has no child elements. See the `requestMode` attribute of the `query` element for information.

query

The `query` element is the standard means of communicating end-user questions to the search engine. The `query` element has no parent element, and contains the following child elements:

- `envName`
- `question`
- `startCategory`
- `priorInput`
- `parserOptions`
- `documentSets`
- `refineConstraint`
- `weightedProps`
- `reportData`

The response to a `query` request is an `answer` element. The `query` element attributes are described in the sections that follow.

andFeedback

This request attribute allows you to include feedback in the search response on searches that are performed using the and mode (see the `mode` attribute). If the full "and" of the search terms the user entered does not provide enough results, you can use the `andFeedback` results to arrive at a subset of terms that does provide results, and use that information in your results page to suggest alternate searches.

```
<query andFeedback="N">
```

N can have the following values:

- N=0
  No feedback is returned.
- N=1
  Only searches with 0 results generate feedback.
- N>1
  Searches with fewer than the specified number of results generate feedback.

The feedback consists of alternate queries that provide all possible combinations of the terms from the original query that satisfy the Boolean and mode. An example of the response feedback follows:

```
<andFeedback>
<altQuery size="1" results="1808" text="women">
  <term exclude="false">women</term>
  <term exclude="true">of</term>
  <term exclude="true">apparel</term>
</altQuery>
<altQuery size="1" results="1329" text="apparel">
  <term exclude="true">women</term>
  <term exclude="true">of</term>
  <term exclude="false">apparel</term>
</altQuery>
</andFeedback>
```

The `altquery` child element includes the following attributes:

- `size`—Number of non-excluded terms included in that subset of the query.
- `results`—Approximate number of search results returned by that subset of the query.
- `text`—Text of the non-excluded terms that comprise that subset of the query.

The `altquery` element includes one `term` child element for each of the original query terms. This makes it easy for you to display the original query and the excluded terms in a different format, if desired; to display only the included terms, refer to the `text` attribute.

The `exclude` attribute of the `term` element indicates whether the term has been excluded from that version of the query. In the above example, notice that "of" is automatically excluded in all and searches, due to its low value as a search term.

Note that if you are grouping by property and the `refineCount=group` attribute is in use (see Setting Grouping Options), the result counts returned reflect groups, not items.

autocat

ATG Search applies rules to determine what categories are relevant to a user queries. One use of this functionality is to automatically add the most relevant categories as constraints on the query itself, thus narrowing the search to the more appropriate content. These automatic constraints are controlled by the following attribute:

```
<query autocat="max"
```

```
<query autocat="maxp"
```

The `max` value is the maximum number of categories to add as constraints. Multiple categories are added as a Boolean OR of document set constraints, joined (that is, ANDed) to the pre-existing constraints. If the `max` value is appended with a `p`, then the optional taxonomy pruning post-processing algorithm is used during categorization.

autocatPrune

When used with the `autocat` attribute, `autocatPrune="true"` indicates that taxonomy pruning should be used during categorization.

```
<query autocat="max" autocatPrune="prune"
```

This process eliminates any category assignments in which content is assigned to a child category where it should also be assigned to the parent. For example, a taxonomy has a category for Product X and subcategories containing topics, such as Installation, Service, Support, Help, etc. Without pruning, the taxonomy rules would be forced to require X in the rules throughout the sub-tree, such as “support for X” and “install X”. This might be possible, but often X won’t be in the same sentence as the other terms required for the sub-categories. With pruning, the taxonomy rules could simply define rules for X under the root product X category, then define generic rules for the sub-tree, like “support” and “install”. Content that matched these generic rules and was assigned to the categories would be pruned if it was not also assigned to the product X category. Taxonomy pruning works globally across all categories, effectively pruning content down the tree that has not been assigned above it.

autospell

ATG Search always returns spelling feedback in the response. It can optionally automatically correct spelling before issuing the query, using the following attribute:

```
<query autospell="true"
```

The `bool` value must be either `true` or `false`, and defaults to `true`.

debug

If this attribute is set to `true`, the returned answer includes the query XML for debugging.

docSetSort

ATG Search can return categorization feedback about the returned results in the form of a tree. This functionality is controlled by the following attribute:

```
<query docSetSort="mode"
```

The `mode` value can be:

- `none`—No categorization feedback tree is constructed.
- `fulltree`—A full categorization tree is returned, with all intervening levels, even if they have no direct connection to the results.
- `sparsetree`—A categorization tree is returned, but intervening levels that have no direct connection to the results are omitted.

The default value is none.

docFlags

This attribute allows full control over how much document information to return, with the potential to greatly affect the size of the returned response. The default is "`url,docsets,properties`" plus "`contextid`" if `optimize` is set to a value greater than 1.

```
<query docFlags="flag1,flag2,flag3…"
```

The possible flags that can be included are:

- `summary`—Document summary, normally just for browse*
- `docsets`—Item set information
- `title`—Index item title*
- `properties`—Metadata properties, limited by `docProps`
- `timestamp`—Index item timestamp*
- `date`—Index item timestamp*
- `contextid`—Item view request highlight information
- `size`—Size of index item source*
- `type`—Major format type of index item (HTML, PDF, etc.)
- `language`—Language of index item
- `url`—Index item URL
- `document`—Same as `url`
- `all`—All flags are set

The `url` (or `document`) flag must be included to get any index item information at all. The default value is "`url,properties,docsets`".

Items marked with an asterisk must retrieve information from the disk; those have the most impact on query speed, and should be eliminated if the information is not needed.

docProps

ATG Search returns the metadata properties associated with the index item of each statement result. These returned properties can be used for user interface functionality, such as customized result pages. By default, ATG Search returns all stored metadata properties, but the list of returned properties can be controlled by this attribute:

```
<query docProps="all"

<query docProps="prop,prop,..."
```

The first example is the default, and indicates that all properties are returned. The second form lists the property names to return in a comma-delimited list. If the attribute is empty (`docProps= ""`) no properties are returned.

docSort

ATG Search returns a list of result groups in its query response. Normally, the result groups are sorted in relevance order, but you may want to allow users to sort the final results by some secondary criteria, such as date. This secondary sort does not affect what results are in the result groups, just the order of the returned groups. Secondary sorting is performed before paging, and is controlled by the following attributes:

```
<query docSort="mode" docSortOrder="order" docSortProp="prop"
dcSortPropDefault="def" docSortPropVal="val" docSortPred="predicate"
docSortCase="bool"
```

The `mode` value specifies how the index items will be sorted, and can be one of the following:

- `relevance`—The default value, return items in relevance order, assuming the item set is a category
- `alpha`—Sort index items by filename (such as `index.htm`)
- `address`—Sort index items by the beginning of the full URL (for example, `http://www.oracle.com`)
- `url`—Sort index items by full URL
- `date` —Sort index items by last modified date
- `strprop` —Sort index item by a metadata string property, requires `docSortProp` attribute
- `numprop` —Sort index items by a metadata number property, requires `docSortProp` attribute
- `title` —Sort index items by title
- `type` —Sort index items by the type, such as HTML or PDF
- `docset`—Sort index items by physical document set
- `index`—Leave index items unsorted, in index order

- `predicate` – Sort groups by combination of the modes specified in the `docSortPred` attribute (see below).

The `order` value determines whether the sort is ascending or descending, either alphabetically or numerically, depending on the sort mode. The `order` value can be either `ascending` or `descending`.

The `prop` value specifies the property name to use for the `strprop` or `numprop` modes. The property name must be a valid property of the given type; for example, for `strprop`, either `string` or `enum`, and for `numprop`, either `integer`, `float`, `boolean` or `date`. Index items that don't have this property will be excluded from the sort. To prevent that, the `def` value can specify the default property value to use for these exceptional cases. The `def` value should agree with the type of the property.

When grouping by property, it is common to have multiple values for a property within an item as well as across a group. The `val` value controls which value of the result group's properties to use. The `val` can be one of the following values:

- `first`—The first value of the first item in the group is used.
- `last`—The last value of the last item in the group is used.
- `high`—The highest (greatest) value of the property from any of the items in the group.
- `low`—The lowest (smallest) value of the property from any of the items in the group.

The default is `first`. As an example, if the items should be sorted by the lowest price and there are multiple price values per item or the items are grouped by some property, the low value for `docSortPropVal` should be used.

The `predicate` value specifies a sequence of sorting modes and orders to apply when `mode="predicate"`, forming a complex sort criterion. The value has the following form:

docSort="predicate" docSortPred="*mode*:*order*:*prop*:*def*:*bool*|…"

The five colon-delimited fields correspond to the five `docSort` attribute values. Note that for modes other than `strprop` and `numprop`, the `prop` and `def` fields are irrelevant and can be omitted. The `order` value should specify the logical precedence of the `mode`, that is, how the results would be placed in order by that individual mode. The `bool` represents the `docSortCase` value.

The overall sort order is controlled using the `docSortOrder` attribute. For example:

```
docSortPred="numprop:descending:popularity:0|numprop:ascending:cost"
docSortOrder="ascending"
```

In this example, search results are first sorted descending by popularity, then ascending by cost (for results where popularity is the same).

The `docSortCase` attribute determines whether any string secondary sorting is case-sensitive (`true`) or not (`false`). This attribute affects the `docSort` mode values of `strprop`, `alpha`, `url`, `address`, and `title`. The sort predicate specifies its case-sensitivity within its fielded format.

docSortOrder

See the `docSort` attribute.

docSortCase

See the `docSort` attribute.

docSortProp

See the `docSort` attribute.

docSortPropVal

See the `docSort` attribute.

docSortPred

See the `docSort` attribute.

docSortPropDefault

See the `docSort` attribute.

feedback

ATG Search returns feedback about related terms and phrases for the query. This functionality is enabled by the following attribute:

```
<query feedback="bool"
```

The `bool` value must be either `true` or `false`, and defaults to `false`.

maxRelatedSets

ATG Search returns information associated with the index item of each statement result. This information includes the related document sets of the index item. By default, ATG Search returns all related document sets, but the number and type of the returned item sets can be controlled by this attribute:

```
<query maxRelatedSets="max" relatedSets="path,path,..."
```

The *max* value is the maximum number of related sets to return. A value of 0 means no related item set information is returned in the response. The default is 0.

Note that even if the value is 0, the physical document set is always returned:

```
<document docset="">
```

The *path* values are item set paths (for example, `/Topics/Product`) which act as constraints on what type of related sets to return. Only related sets that are descendents of one of the *path* values are returned. The default value is an empty string, which means that the related sets are unconstrained.

mergeSettings

Extremely large indexes require more than one physical partition. If more than one partition exists, each partition is queried individually and the results merged for presentation to the end-user. The `mergeSettings` attribute works together with `responseNumberSettings` to control the number of results returned from a merged result set.

The `responseNumberSettings` attribute determines how many results are returned from each partition. The `mergeSettings` attribute determines how many of those total results are returned to the end user. If you use `mergeSettings`, be sure to set it to a number higher than the individual partition results set in `responseNumberSettings`. For example, if you return 50 results from each of four partitions, you may use `mergeSettings` to trim the combined result list to the top 100 results, but not the top 20 results. It is more efficient to trim the `responseNumberSettings` to begin with than to do so after merging.

The `mergeSettings` attribute takes the same options as `responseNumberSettings`. If `mergeSettings` is not set, `responseNumberSettings` is used instead. The syntax for `mergeSettings` is:

```
mergeSettings=doc50,prop50,…
```

Grouping Type	Description
Group-by-document	Groups the raw search results by document, returning up to some maximum number of groups of a certain size, as defined by these parameters, with the default values shown: `doc10,perDoc3,perSol1,` - `doc`–Maximum number of document result groups to return - `perDoc`–Maximum size of a group from an unstructured index item - `perSol`–Maximum size of a group from a structured index item. **Note:** An additional mode, `docrank`, is the same as `document`, but it also uses the relevancy of the document instead of the relevancy of the statement to rank results.

Grouping Type	Description
Group-by-property	Groups the raw search results by a metadata property value, returning up to some maximum number of groups of a certain size, as defined by these parameters, with the default values shown: `prop10,perProp3,` The `prop` parameter is the maximum number of property result groups to return, and the `perProp` parameter establishes the maximum size of a group. To group by property, the *mode* value requires a `sortProp` attribute with the *type*, *name*, and *default* value for the grouping property. See the `sortProp` section in this guide.

Result Type Weights	Normally, all statement results receive the same treatment in the relevancy calculation. However, you may want certain statement types to be weighted higher or lower in the search results. For example, two identical statements from two similar documents usually receive near identical relevancy, with minor differences in the context and document weight factors. However, if the statements are from two different text fields (such as `role:goal` and `role:fact`), and these fields were weighted differently, then their relevancy could vary greatly. ATG Search supports these weighting factors with the following parameters:
	
	`f*1.0,o*1.0,s*1.0,ROLE:ID*2.0`
	
	`f*`—The weight (or multiplier) of preferred answer statement relevance.
	
	`o*`—The weight of structured statement results.
	
	`s*`—The weight of unstructured statement results.
	
	A weight of 1.0 means the original (pure) relevancy is used.
	
	Individual structured types (or fields) can be defined separately, as shown:
	
	`role:goal*1.2,role:symptom*1.1,role:fact*0.5`

Whole Field Result Text	Normally, the result text is the matching statement text plus some additional context for small sentences. However, for structured content, which contains potentially multi-sentence fields of text, you might want to return the entire text of the field as the result. This behavior is controlled by the following parameter: `wholefield0` The `wholefield` parameter holds a Boolean value which, if non-zero, means that the entire enveloping field text for the result's matching statement is returned as the text.

Grouping Type	Description
Suppress Similar Statements	When using ATG Search with ATG Knowledge, you may want to retrieve documents that have unique matching text. ATG Search allows you to suppress documents from the current page if they share a matching text statement. This behavior is controlled by the following parameter: `suppress0` The `suppress` parameter holds an integer value that determines the number of matching statements to compare for suppression. For example, if the value is 2, then the top-2 matching statements for each document result are compared. If two documents share a matching statement, the first one on the page is returned, the other is suppressed. Note that the comparison is case-sensitive and ignores whitespace and punctuation. Note that the total number of results that is reported from the engine does not take into account this suppression.

minScore

ATG Search uses a *relevancy score* to rank results. The relevancy score is calculated based on how well the statement matches the query, plus how related the retrieved index item of that statement is to the query.

During the collection of the final results, before grouping and secondary sorting, ATG Search applies a minimum threshold on the relevancy score, using the following attribute:

```
<query minScore="min"
```

The *`min`* value must range from 0 to 1000, and defaults to 0. Results that do not meet the minimum threshold are discarded.

mode

ATG Search handles natural language and Boolean queries. Simple Boolean syntax is handled automatically as part of the natural language processing, but complex Boolean expressions require a special mode of processing. Furthermore, ATG Search can support simple keyword search behavior in several additional modes. These modes are controlled by this attribute:

```
<query mode="mode"
```

The `mode` value can be any one of the following:

- `nlp`—Natural-language and simple Boolean queries. This is the default value, but should **not** be used for ATG Commerce installations.
- `boolean`—Deprecated.
- `booleanDoc`—This mode performs a Boolean match at the document level. See below for details on Boolean expressions.

- `booleanStmt`—Complex Boolean expressions; see below for details. This mode operates at the statement level, and should therefore not be used for ATG Commerce integrations; for example, "red" and "shoes" would be indexed as separate statements, and a search that includes both would return no results. See below for details on Boolean expressions.
- `keyword`—Handles natural language queries in a simplistic keyword search model. ATG Search parses the query as normal, but each query term is double-quoted and required to appear in the index items of the results. For example, a query of *install procedures* in `keyword` mode would be interpreted as *++"install" ++"procedures"*.
- `and`—Handles natural language queries in an expanded keyword search model. ATG Search parses the query as normal, but each query term is required to appear in the index items of the results. This is similar to the `keyword` mode, but without the double-quotes, which means the query terms could match morphological variants and use term expansions. For example, a query of *install procedures* in and mode would be interpreted as *++install ++procedures*.

  This mode is strongly recommended for use with ATG Commerce.
- `matchall`—Natural language queries as a Boolean AND of terms, as opposed to the default Boolean OR. ATG Search parses the query as normal, but each query term is required to appear in the result statements. For example, a query of *install procedures* in `keyword` mode would be interpreted as *+install +procedures*.

As described in the User-Entered Operators chapter, the required term and excluded term operators represent simple approximations of true Boolean operators, and can be entered as part of the query, with no special user interface. In addition to these simple operators, ATG Search supports a special query syntax for Boolean expressions which are parsed in a special mode of query handling. The Boolean syntax is shown here in Backus Naur Form:

```
expr := <expr> AND <expr>
expr := <expr> OR <expr>
expr := NOT <expr>
expr := ( <expr> )
expr := ['']["]term["]['']
expr := ['']wildcard['']
expr := i..j
```

The first three statements show the syntax for the three Boolean operator expressions, whose operands can themselves be other expressions. The precedence for these operators is: *NOT*, *AND*, *OR*. The fourth statement shows that parentheses can be used to delimit an expression in order to override operator precedence. For example, *x AND y OR z* is interpreted by default as *(x AND y) OR z*, but using explicit parentheses, it could be interpreted as *x AND (y OR z)*.

The last three statements show the three types of simple term expressions: normal term, with optional quote operators; wildcard pattern, with optional single quote operator; and a number range pattern. Thus, full Boolean expressions can utilize all the simple query operators described in this section except for the simple Boolean operators (+, !, ++, !!, +|).

ATG Search handles full Boolean expressions specially, but it shares much of the natural language query handling. ATG Search parses the Boolean expression, building up an operator-operand tree. During this parsing, it processes the terms in the same way as a natural language query, including tokenization, morphology and term expansion. In addition, it also has to process the special query operators that may modify the terms. At the end of this process, ATG Search has a vector of query items that can execute normally, plus a Boolean expression tree that can filter the retrieved sentence results.

optimize

ATG Search can optimize its algorithms for faster speed with little effect on the search results. ATG Search accomplishes this by skipping the statement level relevancy evaluation for each of the candidate index items, resulting in faster query speed. The relevancy is based on the document weighting and rank configuration formula.

You may want to use this option if you are unsatisfied with the query speed on your installation. To enable optimization, use the following attribute:

```
<query optimize="level"
```

A level of 0 is the default, and means no optimization. Level 1 performs part of the statement level retrieval, but just in order to better rank the index item candidates. Level 2 avoids all statement level retrieval and relevancy computations, and achieves the fastest query speed.

Note that with optimization, the search results contain no matching statement text, only the index item (document) information. This means that no statement highlighting is possible for the index item view request.

Note: For even faster query speeds, set the `indexScheme` Text Processing Option to “uncompressed” in Search Administration. This creates larger indexes, but faster queries, and could be useful if you have a small index. Changing the `indexScheme` setting requires reindexing your content. A combination of `indexScheme=uncompressed` and `optimize=2` results in the fastest query speed.

pageNum

ATG Search supports result paging, controlled by the following attributes:

```
<query pageNum="num" pageSize="size"
```

The *`size`* value specifies how many results are returned, in number of response groups as described in the sorting attribute information. If `pageSize` is empty, no paging is performing. If paging is used, and the results do not fit on a single page, resubmit the query with `pageNum="1", pageNum="2"`, etc., to access the additional results (the first result page is page 0).

ATG strongly recommends that you do not use a `pageSize` value greater than 100, as this can lead to performance problems or even search engine failure.

pageSize

See `pageNum`.

QUID

Automatically generated request ID.

rankConfig

As described in the *ATG Merchandising Guide for Business Users*, ATG Search can perform customized rankings and other adjustments to the search using rank configuration. This process can also be performed for the item results. At query time, ATG Search can control which configuration to use with the following attribute:

```
<query rankConfig="name"
```

The *name* value must be the name of a ranking configuration loaded into the index. If no value is given, no ranking adjustments are made.

recurseDocuments

The query algorithm begins at the starting item set, and recursively descends to its children and their children. At each item set, the algorithm collects index items according to other parameters and returns some number of those. The collection of index items is controlled by the following attribute:

```
<query recurseDocuments="mode"
```

A *mode* value of on or `true` means that index items may be collected from child item sets. A *mode* value of `off` or `false` means that index items may only be collected from the immediate item set, excluding any from child sets. A *mode* value of `empty` means the same as `off`, unless the immediate item set has no documents. The default value is `off`.

refineConfig

Facet sets allow users to refine a query by searching within an existing result set. For example, an end-user conducts a search for luggage on a commerce site, then refines their search by color, price, or material. ATG Search returns refinement results based on settings in a `refineConfig.xml` file. This file defines which properties of the indexed products to return as possible facets. Before returning results, ATG Search retrieves the possible values for the properties configured in `refineConfig.xml`. Thus, the existing query can be resubmitted with an additional constraint that limits the results to one of the enumerated property values.

At query time, ATG Search can control which configuration to use and global parameters for the calculation, using the following attributes:

```
<query refineConfig="name" refineConfigDefault="default"
refineConfigMapKey="key" refineConfigMapProp="prop" refineMax="max"
refineTop="top" refineMin="min"
```

The `name` value must be a valid name of a facet set loaded into the index. If no value is given, no calculation is made.

refineConfigDefault

As described in the `refineConfig` section, ATG Search can calculate refinements based on the query results, in order to offer the end-user a quick way of narrowing the search. The *`default`* value specifies the optional name of a configuration to use in case no configuration can be determined based on the search results.

```
<query refineConfig="name" refineConfigDefault="default"
```

refineConfigMapKey

As described in the `refineConfig` section, ATG Search can calculate refinements based on the query results, in order to offer the end-user a quick way of narrowing the search.

The *key* value is a value of the `refineConfigMapProp` property to try to use first to pre-empt the determination of a value from the search results. For example, if the property is `ancestorCategories.repositoryId`, the key value should be a valid value of this property, such as `cat1009`.

```
<query refineConfig="$map" refineConfigDefault="default"
refineConfigMapProp="ancestorCategories.repositoryId"
refineConfigMapKey="cat1009"
```

refineConfigMapProp

As described in the `refineConfig` section, ATG Search can calculate refinements based on the query results, in order to offer the end-user a quick way of narrowing the search.

The *prop* value is the name of a property to use for automatically selecting a refinement configuration based on the search results. This value must be accompanied by a `refineConfig` value of `$map`.

```
<query refineConfig="$map" refineConfigDefault="default"
refineConfigMapProp="ancestorCategories.repositoryId"
refineConfigMapKey="cat1009"
```

refineMax

As described in the `refineConfig` section, ATG Search can calculate refinements based on the query results, in order to offer the end-user a quick way of narrowing the search.

The *max* value specifies the maximum number of facet properties to return, even if the facet set could generate more. The default value is 0, which means no calculation is made.

```
<query refineConfig="name" refineConfigDefault="default"
refineConfigMapKey="key" refineConfigMapProp="prop" refineMax="max"
refineTop="top" refineMin="min"
```

refineTop

As described in the `refineConfig` section, ATG Search can calculate refinements based on the query results, in order to offer the end-user a quick way of narrowing the search.

The top value specifies the maximum number of facet property values (per property). The values are selected in sort order, which usually is in terms of the number of index items that has each value. The default value is 5.

refineMin

As described in the `refineConfig` section, ATG Search can calculate refinements based on the query results, in order to offer the end-user a quick way of narrowing the search.

The min value specifies the minimum size of a facet property value, in terms of the number of index items with that value. The default value is 0.

relatedSets

See `maxRelatedSets`.

relQuestSettings

The `relQuestSettings` attribute represents low-level numeric variables that control the search and relevancy processing. These settings can be declared in the XML using the format:

```
relQuestSettings="/param=value;/param=value;..."
```

They can also be changed in the `<RelQuestSettings>` tag in the global ATG Search configuration file `<ATG10dir>\Search10.0.1\SearchEngine\`*`platform`*`\bin\AEConfig.xml`.

```
<RelQuestSettings>/param=value;/param=value;...</RelQuestSettings>
```

Query XML attributes override settings in the `AEConfig.xml` file. The *param* string is the name of the parameter, and the *value* is an appropriate value for that parameter. Some parameters take a list of values, separated by commas. The remainder of this section describes the parameters. See also the `strategy` attribute, which allows you to set a number of parameters simultaneously.

Matching Statement Parameters

ATG Search constructs a candidate list of matching statements, sorted by an estimated relevancy metric. From the candidate list, the top candidates are matched in detail and have their final relevancy computed. The parameters described in this section apply to these candidate statements.

Note: The defaults are optimized to balance processing speed with result quality. Be cautious in making changes.

Parameter Name	Syntax and Default	Parameter Description
Statement matching maximum	`/retMax=5000;`	Limits the number of top candidates.
Statement matching cut-off	`/retLimit=3000;`	Detailed matching will end before the maximum of top candidates is reached if the number of relevant statements reaches this parameter value. In this context, a relevant statement is one whose relevancy exceeds the Statement Minimum Relevance (see next row). The value of this parameter should be less than or equal to the `retMax` value.

Statement minimum relevance	`/relevMinFAQ=10;`
	
	`/relevMinSent=10;`
		Candidates that have a relevancy score less than this parameter are eliminated from the results. The `relevMinFAQ` parameter is used for preferred answer statement matches, `relevMinSent` for all other statement matches.

Statement relevance cut-off	`/relevCutoff=0;`	Candidates that have a relevancy score less than a percentage of the most relevant statement are eliminated from the results; the percentage is controlled by this parameter. For example, if the highest relevancy score is 80 and the relevancy cut-off percentage is 70%, then all candidates that have score less than 56 are eliminated. The default is 0, which disables this mechanism.
Estimation minimum relevance	`/estimateMin=10;`	Only candidates that have an estimated relevancy that is greater than or equal to this parameter are matched in more detail. Normally, this value is less than or equal to the Statement Minimum Relevance threshold.

Parameter Name	Syntax and Default	Parameter Description
Statement candidate maximum	`/estimateMax=50000;`	The number of total candidates is limited by this parameter. Since this parameter takes effect before the statements are sorted, the candidate statements are collected on a first come, first served basis. However, the query terms are processed in inverse frequency order, guaranteeing the most highly weighted terms will fill in the statement candidates first.

Matching Document Parameters

ATG Search constructs a candidate list of retrieved documents, sorted by a term frequency (TF-IDF) metric. From this list, the top candidates are inspected for matching statements.

Parameter Name	Syntax and Default	Parameter Description
Document retrieval maximum	`/maxDocuments=1000;`	The number of top candidates is limited by the parameter.
Document candidate maximum	`/estimateDocMax=10000;`	The number of total candidates is limited by the parameter. Since this parameter takes effect before the documents are sorted, the candidate documents are collected on a first come, first served basis. However, the query terms are processed in inverse frequency order, guaranteeing the most highly weighted terms will fill in the document candidates first.

Filtering by Thesaurus Link Strength

ATG Search expands query terms using a thesaurus. Thesaurus entries are characterized by link strength, ranging from *equality* to *weak*. By default, ATG Search uses all link types during retrieval, but this behavior is controlled by the following parameter:

```
/link=none;/link=equality;/link=strong;/link=medium;/link=weak;
```

The value of none clears out any previous values, which would result in no term expansions being used during search. Any subsequent values are appended to the list of link types to use. Normally, only the following four setting combinations should be used:

```
/link=none;/link=equality;/link=strong;/link=medium;

/link=none;/link=equality;/link=strong;

/link=none;/link=equality;

/link=none;
```

The first example excludes weak links, the second excludes weak and medium links, the third excludes all but equal links, and the fourth disables all term expansion.

Extending Statement Result Text

By default, ATG Search retrieves a sentence term vector and constructs a statement result with the text of the sentence as the result string. However, some statements can be very small fragments, such as a section header, and lack enough context to be useful as a search result. ATG Search can extend the statement text with subsequent statement text that is also retrieved by the query. This functionality is controlled by three parameters.

Parameter Name	Syntax and Default	Description
Minimum Answer Length	`/minAnswerLength=75;`	The maximum size of a statement text that can be extended, in number of characters. Any statement text that is greater than or equal this value will not be extended.
Maximum Answer Length	`/maxAnswerLength=250;`	The maximum size of an extended statement, in number of characters. If the extended statement size would exceed this value, the statement is not extended. The statement text is extended with successive statements until this limit is reached.
Maximum Intervening Characters	`/maxIntervening=5;`	The maximum intervening characters that can appear between the statement and its extension. Normally, only white space appears between statements, and large white space tends to indicate a separation of content which should not be joined together.

Statement Relevance Parameters

The parameters described in this section all act according to the computed weight of a statement. ATG Search relevancy computation uses a weighted sum of factors for a main score and a tie-breaker score, together forming the final relevancy value or weight.

Parameter Name	Syntax and Default	Parameter Description

Literal weight relevancy factor	`/literalWgt=25;`
	
	`/literalMain=1;`
		Quantifies how closely the surface query terms match the statement terms, discounting indirect matches through term expansions.
		
		`literalWgt` must be a non-negative integer from 0 to 100.
		
		`literalMain` is a Boolean variable. If `true`, this factor is main; if not, this is a tie-breaker factor.

Exact weight relevancy factor	`/exactWgt=0;`
	
	`/exactMain=1;`
		Quantifies if the query text matches exactly within the statement text, without regard to case and white space. The weight of this factor and whether it is a main factor are controlled by the parameters.
		
		`exactWgt` must be a non-negative integer from 0 to 100.
		
		`exactMain` is a Boolean variable. If `true`, this factor is main; it not, this is a tie-breaker factor.
		
		This factor is disabled by default, but can be enabled as part of a search strategy (see the `strategy` attribute).

Proximity weight relevancy factor	`/proxWgt=8;`
	
	`/proxMain=1;`
		Quantifies how close in proximity do the query terms match the statement terms. The weight of this factor and whether it is a main factor are controlled by the following parameters.
		
		`proxWgt` must be a non-negative integer from 0 to 100.
		
		`proxMain` is a Boolean variable. If `true`, this factor is main; it not, this is a tie-breaker factor.

Parameter Name	Syntax and Default	Parameter Description

Document weight relevancy factor	`/docWgt=8;`
	
	`/docMain=0;`
		Quantifies how well the document pertains to the query, using the term frequency calculation. The weight of this factor and whether it is a main factor are controlled by the parameters.
		
		`docWgt` must be a non-negative integer from 0 to 100.
		
		`docMain` is a Boolean variable. If true, this factor is main; it not, this is a tie-breaker factor.

Context relevancy factor	`/contextWgt=17;`
	
	`/contextMain=0;`
	
	`/contextSize=2;`
		Quantifies how well the surrounding statements also match the query. The weight of this factor and whether it is a main factor are controlled by the parameters.
		
		`contextWgt` must be a non-negative integer from 0 to 100.
		
		`contextMain` is a Boolean variable. If `true`, this factor is main; it not, this is a tie-breaker factor.
		
		`contextSize` controls the size of the context, in number of statements, and must be a positive integer.

Metadata relevancy factor	`/metaWgt=25;`
	
	`/metaMain=1;`
	
	`/metaWgtMax=100;`
		Quantifies how well the metadata of the statement's index item match the weighted properties passed in with the query. The weight of this factor and whether it is a main factor are controlled by the parameters.
		
		`metaWgt` must be a non-negative integer from 0 to 100.
		
		`metaMain` is a Boolean variable. If `true`, this factor is main; it not, this is a tie-breaker factor.
		
		`metaWgtMax` specifies the maximum weighted property weight.

Parameter Name	Syntax and Default	Parameter Description
Match denominator	`/matchDenom=0;`	The recall factor is the percentage of statement term weight that the query matched. This calculation is biased towards small statements, which have small total term weights. Use this parameter to force all statements to have the same total weight in terms of this recall calculation. A value of 0 means the normal recall calculation is performed. A positive integer value means that that value is used as the recall denominator, in place of the statement's total term weight.
Duplicate term factor	`/dupTermFactor=2;`	The recall factor is the percentage of statement term weight that the query matched. This calculation is biased towards statements with repeated terms, since each instance of a term is counted separately. Use this parameter to limit the number of occurrences that are significant in the recall calculation. A value of 0 means the normal recall calculation is performed. A value of 1 means that only 1 occurrence of each term is used. An integer value greater than 1 means that up to that number of occurrences are used in the calculation.
Exclude unknown terms	`/exUnk=1;`	A query term that does not exist in the dictionary and has not occurred in the index items provides no information to the system and it cannot retrieve anything. A value of 1 means that the unknown terms are excluded from the query processing and do not effect the relevancy. A value of 0 means that unknown terms are included in the query processing and will hurt the relevancy of the results (since they cannot retrieve anything).

Parameter Name	Syntax and Default	Parameter Description
Special treatment for all-caps terms	`/autoAllCapsMode=0;`	In a mixed case query, often terms in all capital letters refer to the most important information. A 0 value means that no special treatment is given to these terms. A 1 value means that these terms are required to appear in the statement results, the equivalent of the single + query operator. A value greater than 1 means that these terms are required to appear in the document results, the equivalent of the double ++ query operator.

Document Relevance Parameters

ATG Search constructs a candidate list of retrieved documents, sorted by a term frequency (TF-IDF) metric. The parameters described in this section all act according to the computed weight of a document. ATG Search relevancy computation uses a weighted sum of factors for a main score and a tie-breaker score, together forming the final relevancy value or weight.

All terms are used in the statement matching algorithm, giving them some effect on the final results.

Parameter Name	Syntax and Default	Parameter Description
Document Weight Term Threshold	`/docWgtTermThresh=20;`	Terms whose weight is less than this parameter are excluded from retrieval.
Document Weight Link Threshold	`/docWgtExpansion=equal;`	ATG Search uses term expansions for candidate retrieval, but excludes terms expansions whose link strength is less than this parameter. The default value of `equal` restricts the retrieval to only equally-linked terms, retrieving results that are most similar to the original query terms. The other valid values are: `strong`, `medium`, and `weak`.

Search Fields

ATG Search indexes structured content and records the fields from which each sentence term vector was created. Queries can then be constrained to a limited set of those fields, also called a *fielded search*. The following parameter establishes which fields are included in a search of structured content such as ATG Knowledge solutions or an ATG Commerce catalog:

```
/activeSolutionZones=role:id,role:goal,role:symptom,role:question;
```

This parameter can also take a special value to denote all fields should be searched:

```
/activeSolutionZones=*;
```

This is the default value.

ATG Search also indexes *un*structured content and records the fields from which each sentence term vector was created. However, in this case, all sentences from the body of the unstructured content reside in a single field, called doc. The title of the content is stored in a `role:title` field and the URL is stored in a `role:url` field. The following parameter establishes which fields are included in search of unstructured content; all other fields are excluded from the search:

```
/activeSentenceZones=doc;
```

This parameter can also take a special value to denote all fields should be searched:

```
/activeSentenceZones=*;
```

To include the title and URL fields in the search, use the following:

```
/activeSentenceZones=doc,role:title,role:url;
```

Conditional Keyword Interpretation

If the query's `mode` is `nlp`, ATG Search can treat user queries differently depending on the content of the query.

If the user query consists of N terms or fewer **and** the query is a simple list of content terms, then the engine will treat the query as a boolean AND on the documents. If the AND of the terms fails to return any results, the normal `nlp` mode is used instead. If the AND of terms succeeds, only those documents with all of the terms are returned.

The interpretation depends on the form of the user query. It must be a simple list of content terms, such as "book garden summer", rather than a statement, such as "a book about gardening in the summer". ATG Search treats the simple list as an AND, but not the more complex statements or questions.

```
/implicitAndSize=N
```

The default value is 4. To disable this feature, set the value to 0.

requestMode

Normally, ATG Search treats each user query as a separate isolated request, with no pre-existing state or *context*. However, many user searches are interrelated, and you may want to provide context. ATG Search captures several types of search context in its query request XML, and it is controlled by the following special element and attribute:

```
<query requestMode="mode"

<priorInput>context</priorInput>
```

The mode value specifies how the context string in the `priorInput` element is interpreted, and can be one of the following values:

- `normal` – The default value, no search context processing.
- `subtractDoc` – Using the *context* as a preliminary query, eliminate from the current search results any that are from index items also returned by the *context* query.
- `subtractAns` – Using the *context* as a preliminary query, eliminate from the current search results any that are statements also returned by the *context* query.
- `penalizeDoc` – Using the *context* as a preliminary query, penalize any current search results that are from index items also returned by the *context* query. If the penalty exceeds the relevancy, the result is eliminated.
- `penalizeAns` – Using the *context* as a preliminary query, penalize any current search results that are statements also returned by the *context* query. If the penalty exceeds the relevancy, the result is eliminated.
- `withinDoc` – Using the *context* as a preliminary query, restrict the current search results to index items also returned by the *context* query.
- `withinAns` – Using the context as a preliminary query, restrict the current search results to statements also returned by the *context* query.

The `subtract` modes represent the search scenario known as *not like this*, where the end-user does a search that returns relevant but poor results, and then directs the system to find results **not** like the poor results.

The `penalize` modes represent the search scenario known as *less like this*, where the end-user does a search that returns relevant but poor results, and then directs the system to find results **less** like the poor results, but not necessarily eliminating them.

The `within` modes represent the search scenario known as *search within*, where the end-user does a search that returns generally relevant results, and then directs the system to find results of a new query within those initial results.

responseNumberSettings

If grouping is in use (see the `sorting` attribute), these settings controls control how grouping is performed. The values can be set in two ways:

- Through the `AEConfig.xml` global ATG Search configuration file:

```
<ResponseNumberSettings>paramValue,paramValue,...</ResponseNumberSettings>
```

- Per-query as an attribute to the browse XML:

  ```
  responseNumberSettings="paramValue,paramValue,..."
  ```

The XML attributes override settings in the `AEConfig.xml` file. The `param` is the name of the parameter, which is prepended to the numeric value `Value`, as described in the following table:

Grouping Type	**Description**
Group-by-document	Groups the raw search results by document, returning up to some maximum number of groups of a certain size, as defined by these parameters, with the default values shown: `doc10,perDoc3,perSol1,` - `doc`–Maximum number of document result groups to return. - `perDoc`–Maximum size of a group from an unstructured index item. - `perSol`–Maximum size of a group from a structured index item. **Note:** An additional mode, `docrank`, is the same as `document`, but it also uses the relevancy of the document instead of the relevancy of the statement to rank results.
Group-by-property	Groups the raw search results by a metadata property value, returning up to some maximum number of groups of a certain size, as defined by these parameters, with the default values shown: `prop10,perProp3,` The `prop` parameter is the maximum number of property result groups to return, and the `perProp` parameter establishes the maximum size of a group. To group by property, the *mode* value requires a `sortProp` attribute with the *type*, *name*, and *default* value for the grouping property. See the `sortProp` section in this guide.

Grouping Type	Description

Result Type Weights	Normally, all statement results receive the same treatment in the relevancy calculation. However, you may want certain statement types to be weighted higher or lower in the search results. For example, two identical statements from two similar documents usually receive nearly identical relevancy, with minor differences in the context and document weight factors. However, if the statements are from two different text fields (such as `role:goal` and `role:fact`), and these fields were weighted differently, then their relevancy could vary greatly. ATG Search supports these weighting factors with the following parameters:
	
	`f*1.0,o*1.0,s*1.0,ROLE:ID*2.0`
	
	`f*`—The weight (or multiplier) of preferred answer statement relevance.
	
	`o*`—The weight of structured statement results.
	
	`s*`—The weight of unstructured statement results.
	
	A weight of 1.0 means the original (pure) relevancy is used.
	
	Individual structured types (or fields) can be defined separately, as shown:
	
	`role:goal*1.2,role:symptom*1.1,role:fact*0.5`

Whole Field Result Text	Normally, the result text is the matching statement text plus some additional context for small sentences. However, for structured content, which contains potentially multi-sentence fields of text, you might want to return the entire text of the field as the result. This behavior is controlled by the following parameter: `wholefield0` The `wholefield` parameter holds a Boolean value which, if non-zero, means that the entire enveloping field text for the result's matching statement is returned as the text.
Suppress Similar Statements	When using ATG Search with ATG Knowledge, you may want to retrieve documents that have unique matching text. ATG Search allows you to suppress documents from the current page if they share a matching text statement. This behavior is controlled by the following parameter: `suppress0` The `suppress` parameter holds an integer value that determines the number of matching statements to compare for suppression. For example, if the value is 2, then the top-2 matching statements for each document result are compared. If two documents share a matching statement, the first one on the page is returned, the other is suppressed. Note that the comparison is case-sensitive and ignores whitespace and punctuation. Note that the total number of results that is reported from the engine does not take into account this suppression.

ruleMode

ATG Search includes a query module that analyzes user queries and executes special actions which can modify the search behavior (see the *Query Rules* chapter in the *ATG Search Administration Guide*. This functionality is controlled by the following parameter:

```
<query ruleMode="mode"
```

The `mode` value must be one of the following:

- `ignore` – No query analysis will be performed.
- `display` – Perform the query analysis, but simply return feedback about the results in the response.
- `exec` – Perform the query analysis and execute the actions, returning feedback about what was executed.

The default value is `display`, although only an index that contains query rules will return results.

sorting

Search results contain a ranked list of matching statements, sorted by relevancy. These raw results could contain duplicate or similar statements or come from a limited set of documents, offering little variety of information for the end-user. To prevent these problems, ATG Search offers two grouping levels.

In grouping by document, ATG Search reviews the list of matching statements and collapses those that come from the same index item, forming groups of document results. Parameters control the size of result groups and the number of the result groups to return. Individual results that belong to a group that exceeds the size limit are eliminated from the response.

For example, if the query requests the first 20 result groups by document, but restricts the number of the groups to 1, then ATG Search returns the 20 most relevant statements from 20 unique documents, eliminating less relevant statements from the same documents. If the query restricts the number of groups to 3, then ATG Search returns the 60 most relevant results in groups of 3, one group for each unique document. The result objects of the response denote which group they belong to.

Grouping by document allows the user to make a quick survey of the kinds of answers available in the returned documents, but less relevant statements can end up higher on the results list than more relevant statements.

In grouping by property, ATG Search groups matching statements according to a metadata property of their documents. Thus, the grouped results may not share the same document or the same text. The typical scenario for this is a commerce application where you index SKUs as searchable items, but want to return results at the product level. The results can be grouped by a product ID metadata property, so that all SKUs with the same product ID value are in the same group.

The `sorting` attribute determines which grouping mode is used for browse results:

```
<browse sorting="mode"
```

The `mode` value can be `document` or `property`.

sortProp

If the mode identified in the sorting attribute of the query element is property (see `sorting`), additional information is required by the query.

```
<query sorting="property" sortProp="type:name:default"
```

The `sortProp` includes:

- *type*—One of the six valid property types: `enum`, `string`, `integer`, `float`, `boolean`, and `date`.
- *Name*—A valid property name of the given type.
- *Default*—A valid value for the property of the given type; this value is used for results that do not contain the given property.

strategy

ATG Search has a large number of parameters that control searching. To simplify the adjustment of these settings, ATG Search provides five search strategies that implement sets of parameter values. The search strategy is selected by the following attribute:

```
<query strategy="strat"
```

The `strat` can be one of the following five values:

- `everything` – Try unlimited search, without any parameter values that can restrict the search algorithm; also try all term expansions during document candidate retrieval.
- `expand` – Try an expanded search, increasing the default parameter values that can restrict the search algorithm.
- `normal` – Default system settings, optimized for fast search with good search quality; this is the default value.
- `restrict` – Try a restricted search, decreasing the default parameter values which will further restrict the search algorithm; disable non-equal term expansions; adjust relevancy calculation to prefer literal matches.
- `exact` – Try an exact search, which is the same as `restrict` plus a heavy increase in the `/exactWgt` setting. This forces results to contain the literal query string.

If there is a conflict, the `strategy` selection overrides any other preset values.

suggestCat

ATG Search applies rules to determine what categories of a taxonomy are relevant to the user queries. This allows the end-user to manually refine the search. This categorization feedback is controlled by the following attribute:

```
<query suggestcat="max" suggestcatPrune="prune"

<query suggestcat="maxp"
```

The `max` value is the maximum number of categories to return in the feedback. If the `max` value is appended with a `p` or the `suggestCatPrune` value is `true`, then the optional taxonomy pruning post-processing algorithm is used during categorization.

suggestCatPrune

See `suggestCat`.

textProps

ATG Search returns metadata properties associated with the index item of each result. Since ATG Commerce views text fields such as `displayName` as properties too, a way is needed to specify which text fields should be returned as if they were metadata. The `textProps` attribute specifies these properties as shown:

```
<query textProps="prop,prop,…"
```

Each *prop* value is the name of a text field. For structured content such as catalogs and solution knowledge bases, these names begin with `role:`, such as `role:product.displayName`. These names will appear alongside any metadata properties for use in the presentation of results.

queryAction

This element contains action arguments and a `rule` child element. See query rules in the *ATG Search Administration Guide* for information on rules. It is a child of the `answer` element.

queryRule

This element contains query analysis rule results, and an `operation` child element. See query rules in the *ATG Search Administration Guide*. It is a child of the `answer` element.

queryTerms

This element contains advanced query term vector data, and `term` child elements. It is a child of the `answer` element.

question

This element contains the text of the user query after auto-correction has been applied. It is a child of the `query` and `answer` elements. See also the `userquestion` element.

refineConstraint

This element consists of a constraint expression, and is a child of the `query` element.

refinements

This element contains query refinement data in the form of child elements representing different data types for refinement. It is a child of the `answer` element.

reportData

This element contains a subset of information used for search reports (see the *ATG Search Administration Guide* for information on reports), and is a child of the `query` element.

response

This element contains a single search result.

Attributes	Description
`score`	Relevancy score of the result.
`id`	Result index in this response.
`answerGroup`	Result group identifier.
`type`	Type of result.

Attributes	Description
`viewable`	Whether or not the document or preferred answer is viewable.
`field`	The text field name.
`sortprop`	Value of the `docSort` property.

The response element has the following child elements:

- `text`
- `document`
- `debug`

responseTree

This element provides a categorization of the query results. It contains a `responseGroup` child element, which is the document set of returned results, and is a child of the `answer` element.

securityRole

The `securityRole` element is a child of the `parserOptions` element. It has no child elements.

For statement-level security, ATG Search requires a processing option to specify which roles are accessible for the current query:

```
<securityRole>role</securityRole>
```

The *role* is the name of a security region, as expressed in the XHTML. Multiple elements of this type represent a logical OR of accessible regions. These role values are converted into statement features and act as a filter on the candidate statements.

spellchecker

The `spellchecker` element is a child of the `parserOptions` element. It has no child elements.

ATG Search includes two spelling checkers, which are used with natural language processing. The first is an internal module, which uses the indexed content to analyze spelling errors. The second is a third-party library called Wintertree, which uses a dictionary of common terms to guide its analysis.

The internal module does not correct terms that exist in the content, including proper names and other special terms, and it only suggests corrections that exist in the content. Conversely, the third-party module does not correct terms that appear in its common term list, whether they appear in the content or not, and does suggest corrections from its common term list, even if they do not appear in the content.

By default, ATG Search uses both spelling modules to achieve spelling suggestions that reflect the content but are not hampered if the content is limited. The following option controls how these modules interact:

```
<spellChecker>value</spellChecker>
```

The value can be any one of the following:

- `internal`—Use only the internal module.
- `wintertree`—Use only the third-party spelling checker.
- `internal-wintertree`—Use both modules (the default); prefer the internal module's suggestions.
- `wintertree-internal`—Use both modules, prefer the Wintertree module's suggestions.
- none—Perform no spelling correction.

Additional options allow you to control how spelling suggestions are returned:

- `spellMaxSuggestions`—Controls how many suggestions are made for misspelled words.

  ```
  <spellMaxSuggestions>4</spellMaxSuggestions>
  ```

- `spellSuggestionCutoff`—Controls when to stop suggestion corrections.

  ```
  <spellSuggestionCutoff>60</spellSuggestionCutoff>
  ```

- `spellSuggestionFactor`—Controls spelling suggestions for query terms that appear in the index and therefore are not considered misspelled. Normally, no spelling suggestions are returned for such terms. If `spellSuggestionFactor` is set, suggestable terms are returned if their frequency is greater than the original query term's frequency multiplied by the `spellSuggestionFactor`. Set to 0 to disable.

  ```
  <spellSuggestionFactor>10</spellSuggestionFactor>
  ```

spelling

This element contains spelling suggestions and feedback in the form of `term` child elements. It is a child of the `answer` element.

spellSplitWords

The `spellSplitWords` element is a child of the `parserOptions` element. It has no child elements.

This option determines whether the search engine attempts to split unknown words longer than five letters into two known words, and if it does so, whether it does this before or after applying spelling correction. This feature can compensate for simple user errors such as a missing space between two search terms.

The possible values are:

- `false`—Unknown terms are not split.
- `before`—Unknown terms are subject to splitting before applying spelling correction. This is the default behavior.
- `after`—Unknown words are not subject to splitting until after spelling correction has been applied, and only if the term is still unknown.

For example:

```
<spellSplitWords>after</spellSplitWords>
```

startCategory

This element contains the category used as a starting point for category navigation, if applicable. It is a child of the `query` element, and contains `question`, `userquestion`, and `priorInput` elements.

targetLanguage

The `targetLanguage` element is a child of the `parserOptions` element. It has no child elements.

ATG Search supports multiple languages within the same index, to support separate language-specific searches on different document collections. However, ATG Search also supports cross-language searches, which involve searching in one language and retrieving results in one or more different languages.

A use case for this functionality is shopping sites, where customers might not speak or write the site language but can read prices and sizes. In this scenario, users could query in Spanish and get English results and still achieve satisfactory results. In order to perform cross-language searches, both query and target languages must be loaded into the index, as well as special cross-language bridge data (see the *ATG Search Administration Guide*). At query time, the target language is specified with the following option:

```
<targetLanguage>lang</targetLanguage>
```

The *lang* value is the name of any valid language, or one of two special values:

- All means that all languages with indexed content are target languages.
- Same means that the same language as the query is the target, and is the default. Multiple instances of this option are allowed, which will establish target languages.

text

The text element is a child of the response element, and contains the matching statement. A response can include one or more of these elements in the form:

```
<text score="number" name="field">string</text>
```

If the textSort attribute on the query is true, text statements are sorted in document order. You can see the starting position of the <text> statements in the goto="N" attribute.

Highlighting surrounds the matching terms with {beginHighlight} and {endHighlight}, as shown:

```
"{beginHighlight}filter{endHighlight}, oil lubrication
car maintenance {beginHighlight}Air{endHighlight} and Fuel."
```

You can use this information in the Format droplet, as shown in this example:

```
<dsp:droplet name="/atg/dynamo/droplet/Format">
  <!-- this parameter is the text with the curly brace highlight info -->
  <dsp:param name="format" value="${result.text}"/>
  <!-- the following two dsp:param tags supply the replacement text for
  the {Xhighlight} encodings -->
  <dsp:param name="beginHighlight" value="<div class='highlight'>"/>
  <dsp:param name="endHighlight" value="</div>"/>
  <dsp:oparam name="output">
    <!-- converted text is output here -->
    <dsp:valueof param="message"/>
  </dsp:oparam>
</dsp:droplet>
```

topicMaximum

The topixMaximum element is a child of the parserOptions element. It has no child elements.

ATG Search can perform categorization on the query text for automatic category constraints or category feedback. The number of categories used by these processes is controlled by the following options:

```
<topicMaximum>max</topicMaximum>
<topicConfidence>conf</topicConfidence>
<topicRelevance>relev</topicRelevance>
```

The `max` value specifies the maximum categories to return per request. A value of 0 means there is no limit. The default value is 10.

The `conf` value is the confidence threshold for any assigned categories, ranging from 0 to 100. The default is 0.

The `relev` value is the relevance threshold for any assigned categories, ranging from 0 to 100. The default is 1.

userquestion

The `userquestion` element contains the raw user search string, unmodified. It is a child of `startCategory`, and has no child elements.

weightedProps

The weightedProps element is a child of the `query` element. See Weighted Metadata Preference Expressions.

wildcardMax

The `wildcardMax` element is a child of the `parserOptions` element. It has no child elements.

ATG Search can handle wildcard and regular expressions in the user's query, as described in the User-Entered Operators chapter. These patterns are expanded to a set of index terms which are then used during retrieval. ATG Search limits the number of expansions to prevent an explosion of terms that could greatly degrade performance, such as `s*`. The maximum expansions per pattern are specified by the following option:

```
<wildcardMax>max</wildcardMax>
```

The value defaults to 20. Setting this option to 0 disables the interpretation of wild card and regular expressions in user queries.

Index