BreadCrumb Trail Tag v2.0 Component Specification

1. Design

Today's websites have complex navigation rules. As a user navigates through a website, it is very easy to become disoriented. A breadcrumb trail provides links following the path of the user. This component provides an easily customizable tag to provide breadcrumb functionality to a website. The look and feel of the tag is set using CSS style sheets.

While simple web pages can assume the ownership of the whole screen, a *portlet* application running under a *portal* environment can only render to an assigned section of the screen. The new version of this component will be enhanced to support such clustered environments.

In addition to the above, the tag can be used to dynamically adjust the included site map based on the usage patterns:

1. Query strings are now included in both the node matching (the site map XML document includes full regex matching capabilities in identifying the nodes) and generated trails.
2. The last query string for each node in the current path will be 'remembered' and rendered back into the trail – regardless of how the site map has been setup.
3. If a page has not explicitly been specified in the site map, it will automatically be appended to the end of the current trail.

This enhancement has separated the logic in this component into 3 basic areas:

1. *The current node discovery process*. This process provides the component with different ways to discover what the current node is. Two implementations of this have been included:
    a. An inline text tag where the body of the tag can be specified by the application (in order to hardcode the url of the page).
    b. An http servlet implementation that will interrogate the http servlet for the url.

2. *The path discovery process.* The prior breadcrumb parser has been replaced with a path discovery process that will attempt to discover the path from the current node (as identified above) and the root node. Two implementations of this have been included:
    a. A dynamic path discovery that will use the XML site map as a base and then include dynamic capabilities (as described above) on top of it.
    b. A static path discovery process that will mimic the behavior of the v1.0 component (with the exception that regex patterns can be specified also).

3. *The node formatting process.* The node formatting process allows the type of node rendering for the trail to be fully pluggable for the environment needed. This component provides 3 implementations of this:
    a. A WebLogic netui anchor implementation, which utilizes the netui anchors included in WebLogic, for proper portal url generation.
    b. A WebSphere urlGeneration anchor implementation that utilizes the included urlGeneration included with WebSphere for proper portal url generation.
    c. A simple html anchor implementation to generate html anchors.

   The application can specify this as tags in the body of the bread crumb trail tag.

Please note that this is almost a complete rewrite of the V1.0 component and the class diagrams have not been marked up to reflect what has changed because everything has essentially changed. The classes that did change have, in the class doc, a listing of the items that have changed and the documentation of individual elements will have implementation notes of what has changed. Please note that the zuml file has been resync'd with the implementation, the newest standards applied and all documentation corrected.

1.1 Design Patterns

Strategy Pattern – is used extensively in this application to provide various implementations of functionality that can easily be swapped

Iterator Pattern – is used by the BreadCrumbTrailTag to iteratively evaluate the body of the tag

1.2 Industry Standards

XML, JSP specification 1.1, JSP tag libraries

1.3 Change List

Existing source files:

- Updated the tld, xml and xsd files to reflect new tags and xml requirements
- BreadCrumbParser – eliminated
- BreadCrumbParserException – eliminated
- BreadCrumbNode
    1. Eliminated the parent url (variable, constructor and setters
    2. Eliminated all setter methods (class is immutable now)
    3. Updated all NullPointerExceptions (NPEs) to IllegalArgumentExceptions (IAEs).
    4. Eliminated all synchronization
    5. Updated all documentation in zuml.

- BreadCrumbException
  1. Changed constructors to be public
  2. Added 2nd constructor (message, throwable)
  3. Updated all documentation in zuml.

- BreadCrumbException
  1. This class now inherits from BodyTagSupport
  2. Updated all documentation in zuml.
  3. Changed all NullPointerExceptions to IllegalArgumentException and changed all string setters to save a null when it's an empty string
  4. Eliminated the SITE_MAP_ATTRIBUTE_NAME
  5. Eliminated dataSource and associated getter/setter
  6. Eliminated siteDataSource and associated getter/setter
  7. Eliminated parserClassName and associated getter/setter
  8. Eliminated the private variable out
  9. Eliminated the parseSiteDataSource private method
  10. Eliminated the printCurrentRootNode private method
  11. Eliminated the printCurrentNode private method
  12. Eliminated the printRootNode private method
  13. Eliminated the printNode private method
  14. Eliminated the printN private method
  15. Added, documented and added out argument to printStyleSettings private method
  16. Added, documented and added out argument to printStyle private method
  17. Added, documented and added out argument to printSeparator method
  18. Added, documented and added out argument to getSilentErrors/setSilentErrors method
  19. Added constructor
  20. Changed doStartTag fully
  21. Added doInitBody method
  22. Added doAfterBody method
  23. Added doEndTag method
  24. Added iterator private variables
  25. Added title private variable and associated getter/setter methods
  26. Added pathOverride private variable and associated getter/setter

All other classes shown in the class diagram(s) are new classes to this design.

1.4 Tags and their parameters:

Please see bread_crumb_trail_tag_2.0.tld for more details

1.4.1 *“breadCrumbTrail” tag*

The breadcrumb trail tag can have most of its attributes defined in a configuration file for easy assigning of common attributes (see the configurable column). This tag has the following attributes assignable to the tag:

Attribute	**Description**	**Required**	**Configurable**
pathSeparator	The characters to use to separate nodes in the path	No	Yes
pathSeparatorStyle	The CSS style of the separator	No	Yes
nodeStyle	The CSS style of the non-root and non-current nodes	No	Yes
rootNodeStyle	The CSS style of the root node	No	Yes
currentNodeStyle	The CSS style of the current node	No	Yes
mouseOverStyle	The CSS style when the mouse hovers over any node	No	Yes
silentErrors	Whether exceptions should silently be ignored	No	Yes
pathDirection	The direction of the trail	No	Yes
pathOverride	A hardcoded path that overrides normal processing	No	No
title	The title of the current page	No	No

The breadcrumb trail tag also supports a body that can be used (when the node formatter is not specified in the object factory component) to specify the formatting of each node. The templateNodeFormatter or htmlAnchor tags can be used. Additionally, the following scripting variables are defined and can be used when the user specifies simple text in the body:

Variable Name	**Type**	**Description**
breadCrumbFormatNode	BreadCrumbNode	The current node being formatted
breadCrumbFormatStyle	String	The current CSS style to use

1.4.2 *“templateNodeFormatter” tag*

This tag will format an anchor based on a template file (such as websphere.jsp or weblogic.jsp). The tag will define the current node to format and the current formatting style as request attributes:

Name*	**Type**
breadCrumbFormatNode	BreadCrumbNode
breadCrumbFormatStyleClass	String

* the names are from the BreadCrumbTrailTag constants

The specified template file is then included into the output stream.

This tag has the following attributes assignable to the tag:

Attribute	Description	Required	Configurable
template	The name of the template JSP that will be used to generate the anchor.	No	Yes – via Object Factory

1.4.3 *“htmlAnchor” tag*

This tag can be used in the body of the “breadCrumbTrail” to format a simply html anchor. There are no attributes assignable to this tag:

1.4.4 *“httpServlet” tag*

This tag can be defined before the “breadCrumbTrail” to define the current node based on the requestURI/query string from the HttpServlet.

This tag has the following attributes assignable to the tag:

Attribute	Description	Required	Configurable
title	The page title to assign to this page	No	No

1.4.5 *“inlinenode” tag*

This tag can be defined before the “breadCrumbTrail” to define the current node based on the hard coded text within the body of this tag.

This tag has the following attributes assignable to the tag:

Attribute	Description	Required	Configurable
title	The page title to assign to this page	No	No

The body of this tag must be specified and will be used as the URL for the page (allowing the application programmer to override the url to a specific value).

1.5 Required Algorithms

1.5.1 *The template*

This component will render the following template:

```
<span>
     <style>
        path-separator-style { style string }
        node-style { style string }
        current-node-style { style string }
        root-node-style { style string }
```

```
    </style>
    ** The pluggable node formatting **
</span>
```

The various style(s), shown above, will only be included if the corresponding variable is non-null (i.e. if the ‘nodeStyle’ variable is null, the ‘node-style’ line above will not be included). The “pluggable node formatting” will be where the node formatting results are put. Please note that the whole “<span>” tag will not be included if there is no style information (i.e. all the style variables are null).

1.5.2 Query Strings

The old component ignored the query string portion of the Url. This version will handle query strings in the node discovery, the path discovery and in the node formatting.

The node discovery will simply create the node with a url that has the query string appended to it (i.e. “www.topcoder.com?id=1”).

The path discovery will can then use the query string portion as part of the matching (in other words, the regex used to match nodes can include patterns on the query string). The developer should note that if query strings are used in the site, the site map must include query string matching in the xml file (or a “.*” regex pattern on the end of the url to match all query strings for that url). Please also note that differing query strings (on the same url) will create different nodes on the trail if the node is a dynamic node (i.e. doesn’t match any of the regex pattern).

The node formatter will then include the query string as part of the links that are rendered.

1.5.3 Overall process

The overall process that the BreadCrumbTrailTag will follow is:

1. Determine the current node.
2. Determine the current path to the root.
3. Write out the header and styles.
4. Write out the node formatting for each node in the path.
5. Write out the ending tags.

Steps 1-3 are carried out in the doStartTag method of the class. Step 4 in the doInitBody/doAfterBody tags (i.e. letting the body process them). Step 5 will occur in the doEndTag method.

1.5.3.1 Determining the current node

The current node is determined in the following way:

1. Is a BREADCRUMB_CURRENT_NODE attribute already defined? If the user specified one of the node defining tags before the breadcrumb trail tag, it will have defined the node in the page scope variable BREADCRUMB_CURRENT_NODE.
2. If found, simply use that node and go to step 2 in the overall process.
3. Using the object factory, create the default NodeDiscovery class and call getCurrentNode.
4. Use the node returned and go to step 2 in the overall process
5. If an exception occurs or the node is null, we either throw and exception or simply return SKIP_BODY (if we are suppressing errors).

1.5.3.2 Determining the current path to the root

We determine the current path to the root node by calling a PathDiscovery implementation with the current node (defined in 1.5.2.1):

1. Using the object factory, create the default PathDiscovery class
2. Call getPath with the current node
3. If a path is returned, skip back to step 3 of the overall process
4. If the path is null, simply return SKIP_BODY (couldn't find a path)
5. If an exception is thrown, either throw the exception or return SKIP_BODY (if we are suppressing errors).

1.5.3.3 Writing out the header and styles

Writing out the header ("span") and the styles will only occur if there are styles specified (i.e. they are all non-null).

1.5.3.4 Writing out the node formatting

This is where things get interesting for the trail tag. There are two ways of processing the path – backwards and forwards. A list iterator will be created either at the 0th index position (processing root to current) or at the size-1 position (processing current to root). The processing will then either run forwards or backwards on the iterator.

The actual formatting will occur in the body of the tag. The process then does the following:

1. Jsp Engine calls doInitBody method
2. doInitBody will take (from the iterator in the correct direction) the next/previous node and style. The method will then put them into the attribute map under specific names. The names have been defined as variables in the TagExtraInfo class and will be available to the nested tags or nested text.
3. Jsp Engine calls the body processing
4. Jsp Engine calls the doAfterBody method

5. doAfterBody will then read in the formatting that occurred in the body, write it out to the JspWriter and clear the body (in preparation for the next processing).
6. If there is another node in the iterator to be processed, it returns EVAL_BODY_AGAIN
7. If there is no other node in the iterator, SKIP_BODY is returned

By doing this process, the doInitBody advances the iterator through the nodes and makes those nodes/styles available to the body. The doAfterBody is then responsible for writing out the results and determining if any nodes are left.

1.5.3.5 Writing out the ending tags

This logic appears in the doEndTag method and simply writes out the closing html tag for the span.

1.5.4 Dynamic Path Discovery

The dynamic path discovery is really the heart of this component. The class will attempt to discover the path of the current node to the root using a combination of a ‘learned’ path and a static site map (see 1.5.4 for information about the static site map).

Both of those variables are kept in session variables that will be reused every time the class is called for a path.

The site map is simply a mapping of NodeMatcher (key) object to a List (value) of NodeMatcher objects. The key is considered the parent of all the objects in the list (the children). Note: the developer is free to also create a reverse mapping (child to all parents) if it will help their implementation. The only restriction the developer has (in assigning new state variables) is to define a public static variable for that state information (like CURRENT_REVERSE_SITEMAP).

The current path is the path the class has ‘learned’ so far. Each element of the path is of NodePair type and contains the actual node (including any contextual information [like the query string]) and the NodeMatcher that the node matched (which will be used to access the site map).

The rules that we will be following for generating a path is:

1. If the current node matches any of the nodes on the current path (as defined by the node matcher), we take the path from the current root to the current node – discarding all nodes after that one.
2. If current node matches no nodes in the site map, it’s considered a ‘new’ node that is linked from the last site. In other words, we append the node to the current path (if there is one, if not – we simply return null because we don’t know the path).

    3. Otherwise, we perform a shortest path search on the site map to a root and then try to match it up with as much of the current path as possible.

The underlying NodeMatcher for this implementation will be a regex matcher (which will be fairly slow on large sites). Because of that, we perform two quick checks that cover two common situations:

1. We iterate the current path backwards and see if the node matches the NodeMatcher for any element. If the element has a null matcher, compare the node's URL directly to determine if it matches. If a match is made and it is not the last NodePair on the chain, a new NodePair is generated using the current node and the node matcher (effectively discarding the old node's contextual information) and then discard all elements past the matching node. A new path from root to that node is returned. This will cover the situations where the user clicked on the trail (which should match somewhere on the trail) or pressed the back button (which should match the last node).

2. We take the last element in the path and see if the current node matches any of the children of that node. If it does, we simply append a new NodePair to the end of the path and return the new path. This covers the situation where they went to a place that is well defined (in the site map).

3. Failing those two – we will do a shortest path search on the sitemap for any NodeMatcher that matches the node. Example:
    We may iterate the sitemap and find 3 potential candidates (i.e. the current node matched 3 NodeMatcher keys in the site map). We then iterate all the parents of those 3 nodes (this is where a reverse site map may come in handy!). If any of those parents have no parents, then we have found a shortest path to a root node. If they all have parents, we then recursively search the parents of the parents until we have found a root node (or more than one). Note: the developer is free to implement whatever shortest path algorithm they choose – as long as it correctly identifies the shortest path (or paths if there are more than one).
4. We then take the shortest path(s) and try to determine which one has the most commonality with the current path starting with the root node forward. Example:
    Let's say our current path is A->B->C->D->E
    Let's say we found to paths to our current page M:
        A->B->L->M
        A->B->C->M
    Because "A->B->C->M" had the 3 nodes in common with our current path, we choose that path. If we have more than one with the same commonality, choose the first one (where 'first' can be defined anyway you choose).
5. Calculate the new path, save it and return it.

6. If the no shortest path is found for the node and we have a current path, we assume this is a new ‘dynamic’ link and simply append the node onto the end of the current path.

7. If no shortest path was found for the node and we have no current path (i.e. the user probably went to this page directly and it’s not listed in the site map), return null indicating we have no idea what the path is.

1.5.5 XML Path Discovery

The Xml Path Discovery object has two functions, it can be used as a static map path discovery (i.e. not dynamic capable) and it can serve as a source for the static site information to other PathDiscovery implementations (like the Dynamic Path Discovery). The processing of the file is really no different than the old BreadCrumbParser did (and the developer should roughly follow that). There are only a few real differences:

1. There is now a pattern attribute in the XML file that must be read
2. The loading should create RegexNodeMatchers rather than BreadCrumbNodes
3. The parent-child relationship is encoding in the sitemap itself. The sitemap is a map of parent nodes to child nodes where ‘node’ is defined as a RegexNodeMatcher.

Every node found in the Xml document should appear as a key in the site map with a value of a List. That list will then hold references to any children that node may have (or be empty if it’s a leaf node).

Let’s take an example file:

```
    <bc_node title="MyMain" url="/" pattern="/">
        <bc_node title="Forums" url="/forums" pattern="/forums">
           <bc_node title="Help" url="/forums" pattern="/forums?id.*"/>
        </bc_node>
        <bc_node title="Topics" url="/topics" pattern="/topics"/>
    </bc_node>

    <bc_node title="AltMain" url="/main" pattern="/">
        <bc_node title="Help" url="/forums" pattern="/forums?id.*"/>
    </bc_node>
```

Here we have a “Help” node that has two parents. The resulting site map would look like (using only the title to represent each node):

Key	**Value List**
MyMain	Forums, Topics
Forums	Help
Topics	{ empty }
Help	{ empty }
AltMain	Help

1.6 Component Class Overview

BreadCrumbTrailTag:

The main tag class that application will use to render a breadcrumb trails. This tag will render the current node to the root node (or vise-versa) using a path separator and specific styles. The application can setup the separator and the various styles either by specifying default values in the configuration manager or by specifying attributes on the string. The object factory will be used to discover the node, the path to the root and the node formatter. The node discovery and node formatter is optional. If the node discovery is not specified, it is assumed a node discovery tag was used on the page to define the current node. If the node formatter was not specified, it is assumed the user specified a body to this tag that will format the node.

BreadCrumbTrailTagExtraInfo:

This is the tag extra info that describes the attribute to variable mapping that will occur within the BreadCrumbTrailTag. This tag extra info will define two variables (that can be used in the body of the BreadCrumbTrailTag):

- BreadCrumbTrailTag.BREADCRUMB_FORMAT_NODE variable that will contain a reference to the BreadCrumbNode that should be formatted
- BreadCrumbTrailTag.BREADCRUMB_FORMAT_STYLE variable that will contain a reference to the String that represents the style that will be used. Please note that this string can be an empty string or null to represent no styles to be applied

NodeDiscovery:

This interface defines the contract for classes wishing to discover the current node. Implementations of this interface will be called when the current page has not been defined (ie a page scope attribute for BreadCrumbTrailTag.BREADCRUMB_CURRENT_NODE). The implementation should discover, in it's own way, the current node when getCurrentNode() is called and return the node representation of that url.

AbstractNodeDiscovery:

This is an abstract implementation of the NodeDiscovery interface and is useful to NodeDiscovery implementations that are also a Tag. This abstract class simplifies the work required when an implementation of the NodeDiscovery will also provide services as a tag and provides an attribute to set the title. This abstract class will implement doStartTag, call getCurrentNode to get the current node and then set the result as a page scope attribute using BreadCrumbTrailTag.BREADCRUMB_CURRENT_NODE.

PathDiscovery:

This interface defines the contract for classes wishing to discover the path from current node to the root node. Implementations of this interface will be called with the current page (specified by a url string) and the current page context. The implementation will discover the complete path from the current node to the root or return null when it can't discover the current path.

BreadCrumbNode:

This node represents a structure for holding url and the page title of node.

TemplateNodeFormatterTag:

This tag will format the node and style using a template. A template is a separate file that will be included in the out stream. This class will take the current node/style and define them as request attributes when the template file is included (those attributes can then be used by the template file to format the correct representation of them)..

HttpServletNodeDiscoveryTag:

This tag and implementation of the NodeDiscovery will discover the current node from the HttpServletRequest. This implementation will return a valid Uri (with query string if specified) in the getCurrentNode() method. If this is used as a tag, this will define a page variable called BreadCrumbTrailTag.BREADCRUMB_CURRENT_NODE with the value a string representing the node uri with query string

HtmlAnchorNodeFormatterTag:

This tag will format the node and title using the html anchor pattern of <a href='node.getURL()'>title</a>

InlineTextNodeDiscoveryTag:

This class provides a tag (only) implementation of node discovery that will allow the user to hardcode the url into the body of this tag and specify the title in the attributes. The hard coded information is then combined with the title to make a BreadCrumbNode. The node is then put in a page level attribute called BreadCrumbTrailTag.BREADCRUMB_CURRENT_PAGE.

NodePair:

This class is a typical pair class that holds a paired association of a node matcher to the matching node. The nodeMatcher describes the node that matched the above node and describes the default node information. The node describes the actual node that matched the nodeMatcher and contains the contextual specific information.

NodeMatcher:

Defines the contract for a node matcher. Implementations need to provide three functions:

- A matching function that will take a node and determine if it matches.

- A default title string used for those nodes that match.
- A default url to use if no contextual information is available.

AbstractNodeMatcher:

This abstract implementation of the NodeMatcher interface provides title and url type services to subclasses. This implementation will provide a title and url holder variable and a getter method.

RegexNodeMatcher:

This implementation of a NodeMatcher will provide node-matching services given a specific regex pattern. This class will return true when the passed node's url matches the specified pattern.

SiteDiscovery:

Defines the contract for a data source for the site. Implementations of this interface should create and return a site map consisting of NodeMatcher nodes that describe the site map. The returns map should have each unique NodeMatcher listed as the key and a List implementation of NodeMatcher nodes that describe the children of that key.

DynamicPathDiscovery:

This implementation of the PathDiscovery will attempt to create a static site map and then apply dynamic information to that site map. The dynamic information can either be in matching nodes with different query strings or in new nodes that will be assumed to map to the last known node.

XmlPathDiscovery:

This implementation of the PathDiscovery and SiteDiscovery will attempt to create a static site map from either a filename or string source (both of which is an XML document). This class will parse the document to create a static map of regex matchers (where each node can provide a regex matching). If used as a PathDiscovery, this provides a static view of the site (i.e. no dynamic capabilities).

1.7 Component Exception Definitions

BreadCrumbException:

Exception thrown in all cases where the BreadCrumbTrailTag cannot correctly parse, format and draw breadcrumb trail. The only other exception thrown by this component is the IllegalArgumentException.

1.8 Thread Safety

The Jsp Engine will call the tag(s) in a thread safe manner and therefore thread safety isn’t an issue. However, many classes are immutable or have no state information and will naturally be thread-safe.

2. Environment Requirements

2.1 Environment

- At minimum, Java1.4 is required for compilation and executing test cases.
- This component must run inside a JSP 1.1 or greater servlet container.

2.2 TopCoder Software Components

- Object Factory 2.0 – this is a enhancement request (see the enhancement request document in the docs directory for more details). This enhancement request was made because a number of classes, which will be created by the object factory, are hidden from the user completely. These classes have useful, to the user, construction arguments that can only be called if the object factory is enhanced. Example: the XmlPathDiscovery will need a filename for the static site information. Without this enhancement, we'd have needed to pass the filename through the tag, through the interface to the implementation. This is information that is very implementation specific and is very static (to the whole site). By allowing the object factory to construct the class using a filename embedded in the configuration file, the class can be constructed with little impact to the public api.
- Configuration manager version 2.1.4 will provide default configuration of the breadcrumb trail tag.

NOTE: The default location for TopCoder Software component jars is../lib/tcs/COMPONENT_NAME/COMPONENT_VERSION relative to the component installation. Setting the tcs_libdir property in topcoder_global.properties will overwrite this default location.

2.3 Third Party Components

None needed

3. Installation and Configuration

3.1 Package Name

com.topcoder.web.ui.tag

3.2 Configuration Parameters]

The following configuration parameters are available to the BreadCrumbTrailTag under the com.topcoder.web.ui.tag.BreadCrumbTrailTag namespace

Parameter	**Description**	**Values**
pathSeparator	The path separator that will be used	String, optional defaults to “:”
pathSeparatorStyle	The css style used when rendering the path separator	String, optional, defaults to nothing
nodeStyle	The non-root, non-current node css style	String, optional, defaults to nothing
rootNodeStyle	The root node css style	String, optional, defaults to nothing
currentNodeStyle	The current node css style	String, optional, defaults to nothing
mouseOverStyle	The mouse over css style for all nodes	String, optional, defaults to nothing
pathDirection	The direction of the path between the root node and the current node	Integer, optional, defaults to ROOT_TO_CURRENT
silentErrors	Whether errors should be silently ignored	Boolean, optional, defaults to false

The following configuration parameters are available to the BreadCrumbTrailTag under the com.topcoder.web.ui.tag.BreadCrumbTrailTag namespace

Parameter	**Description**	**Values**
pathSeparator	The path separator that will be used	String, optional defaults to “:”
pathSeparatorStyle	The css style used when rendering the path separator	String, optional, defaults to nothing
nodeStyle	The non-root, non-current node css style	String, optional, defaults to nothing
rootNodeStyle	The root node css style	String, optional, defaults to nothing
currentNodeStyle	The current node css style	String, optional, defaults to nothing
mouseOverStyle	The mouse over css style for all nodes	String, optional, defaults to nothing
pathDirection	The direction of the path between the root node and the current node	Integer, optional, defaults to ROOT_TO_CURRENT
silentErrors	Whether errors should be silently ignored	Boolean, optional, defaults to false

The following classes can have object factory specified constructor parameters.

TemplateNodeFormatterTag

Parameter	Description	Values
template	The name of the template JSP that will be used to generate the anchor.	String - required

XmlPathDiscovery

Parameter	Description	Values
filename	The location of the file that contains the XML site map	String, required

XmlPathDiscovery

Parameter	Description	Values
source	A string containing either a filename or a direct XML document	String, required
isFile	True if the source is a file,	Boolean, required

3.3 Dependencies Configuration

None

4. Usage Notes

4.1 Required steps to test the component

- Extract the component distribution.
- Follow Dependencies Configuration.
- Execute 'ant test' within the directory that the distribution was extracted to.

4.2 Required steps to use the component

Deploy the bread_crumb_trail.tld tag library definition file to the WEB-INF folder
Deploy the bread_crumb_trail.jar to the classpath of your web server
Reference the bread_crumb_trail.tld in your JSP page
Create a site or page XML data source with titles and urls of pages composing Bread Crumb Trail.
Create a BreadBrumbTrail tag and reference the collection

4.3 Demo

There are several demonstrations of this component that can be done.

For the following demonstrations, assume the users have visited the following pages before the demonstration page:

(Main) http://www.xyz.com

(Forums) http://www.xyz.com/forums

(Help) http://www.xyz.com/forums?id=1

The static site map defines them like:

```
<?xml version="1.0" encoding="UTF-8"?>

<BreadCrumb xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
            xsi:noNamespaceSchemaLocation="BreadCrumbTrailSchema.xsd"
            version="v2">

    <bc_node title="Main" url="/" pattern="/">
        <bc_node title="Forums" url="/forums" pattern="/forums">
           <bc_node title="Help" url="/forums?id=1"
                                 pattern="/forums?id.*"/>
        </bc_node>
    </bc_node>
</BreadCrumb>
```

4.3.1 *Normal demonstration*

The following page “http://www.xyz.com/forums?thread=1” defines the following

```
<%@ taglib uri="WEB-INF/bread_crumb_trail.tld" prefix="bc" %>

<hmtl><body>
My portable hole stopped working on the floor!

<bc:breadCrumbTrail title=”portable hole”>
   <bc:htmlAnchor/>
</bc:breadCrumbTrail>

// etc etc
</body></html>
```

Would create a bread trail of:
“Main : Forums : Help : portable hole” where:

```
       “Main” would be:
           <a href=”/”>Main</a>
       “Forums” would be:
           <a href=”/forums”>Forums</a>
       “Help” would be:
           <a href=”/forums?id=1”>Help</a>
       “portable hole” would be:
           <a href=”/forums?thread=1”>portable hole</a>
```

4.3.2 *Weblogic demonstration*

We could have used a Weblogic anchors instead (see the weblogic.jsp file in the docs directory)…

```
<%@ taglib uri="WEB-INF/bread_crumb_trail.tld" prefix="bc" %>
<%@ taglib uri="http://beehive.apache.org/netui/tags-html-1.0"
prefix="netui”%>
```

```
<hmtl><body>
My portable hole stopped working on the floor!

<bc:breadCrumbTrail title=”portable hole” >
  <bc:templateNodeFormatter template=”weblogic.jsp”/>
</bc:breadCrumbTrail>

// etc etc
</body></html>
```

Would create a bread trail of (note: the netui anchor will get replaced by whatever the tags generate):
“Main : Forums : Help : portable hole” where:

```
      “Main” would be:
          <netui:anchor href=”/”>Main</netui:anchor>
      “Forums” would be:
          <netui:anchor href=”/forums”>Forums</netui:anchor>
      “Help” would be:
          <netui:anchor href=”/forums?id=1”>Help</netui:anchor>
      “portable hole” would be:
          <netui:anchor href=”/forums?thread=1”>
                        portable hole</netui:anchor>
```

4.3.3 *WebSphere demonstration*

We could have used a WebSphere anchors instead(see the websphere.jsp file in the docs directory)…

```
<%@ taglib uri="WEB-INF/bread_crumb_trail.tld" prefix="bc" %>
<%@ taglib uri="/WEB-INF/tld/engine.tld" prefix="wps" %>

<hmtl><body>
My portable hole stopped working on the floor!

<bc:breadCrumbTrail title=”portable hole” >
  <bc:templateNodeFormatter template=”websphere.jsp”/>
</bc:breadCrumbTrail>

// etc etc
</body></html>
```

Would create a bread trail of (note: the netui anchor will get replaced by whatever the tags generate):

“Main : Forums : Help : portable hole” where:

```
      “Main” would be:
         <wps:urlGeneration contentNode="/">
           <A HREF="<%wpsURL.write(out);%>”>Main</A>
         </wps:urlGeneration>
      “Forums” would be:
         <wps:urlGeneration contentNode="/forums">
           <A HREF="<%wpsURL.write(out);%>”>Forums</A>
         </wps:urlGeneration>
      “Help” would be:
         <wps:urlGeneration contentNode="/forums?id=1">
           <A HREF="<%wpsURL.write(out);%>”>Help</A>
         </wps:urlGeneration>
      "portable holde" would be:
         <wps:urlGeneration contentNode="/forums?thread=1">
           <A HREF="<%wpsURL.write(out);%>">portable hole</A>
         </wps:urlGeneration>
```

4.3.4 Specified formatting demonstration

We could have used a specialty engine instead

```
<%@ taglib uri="WEB-INF/bread_crumb_trail.tld" prefix="bc" %>
<%@ taglib uri="/WEB-INF/tld/myAnchors.tld" prefix="abb" %>

<hmtl><body>
My portable hole stopped working on the floor!

<bc:breadCrumbTrail title="portable hole" rootNodeStyle="color.green">
  <abb:anchor
     class="<%=breadCrumbFormatStyleClass%>"
     href="<%=breadCrumbFormatNode.getUrl()%>">
        <%=breadCrumbFormatNode.getTitle() %>
   </abb:anchor>
</bc:breadCrumbTrail>

// etc etc
</body></html>
```

The above (for just the "main" – but you can probably guess how the others would look) would then render:

<abb:anchor class=".rootNodeStyle" href="/">Main</abb:anchor>

4.3.5 Specified the node itself using the http servlet tag

The following would define the page via http servlet for a given title.

```
<%@ taglib uri="WEB-INF/bread_crumb_trail.tld" prefix="bc" %>

<hmtl><body>
My portable hole stopped working on the floor!

<bc:httpServlet title="portable hole"/>
<bc:breadCrumbTrail>
   <bc:htmlAnchor/>
</bc:breadCrumbTrail>

// etc etc
</body></html>
```

This demonstration is exactly the same as in section 4.3.1 but using the httpServlet tag directly

4.3.6 Overriding the page url fully

The following could be used to directly specify the url (overriding any query string or other information)

```
<%@ taglib uri="WEB-INF/bread_crumb_trail.tld" prefix="bc" %>
```

```
<hmtl><body>
My portable hole stopped working on the floor!

<bc:inlinenode title=”portable hole”>
   /forums/protablehole
</bc:inlinenode>

<bc:breadCrumbTrail>
   <bc:htmlAnchor/>
</bc:breadCrumbTrail>

// etc etc
</body></html>
```

The current node would then be rendered (using html anchor) like:
<a href=”/forums/portablehole”>portable hole</a>

4.3.7 Multiple parents

Assume the static site map is defined like:

```
<?xml version="1.0" encoding="UTF-8"?>

<BreadCrumb xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
            xsi:noNamespaceSchemaLocation="BreadCrumbTrailSchema.xsd"
            version="v2">

    <bc_node title="MyMain" url="/" pattern="/">
        <bc_node title="Forums" url="/forums" pattern="/forums">
           <bc_node title="Help" url="/forums" pattern="/forums?id.*"/>
        </bc_node>
    </bc_node>

    <bc_node title="AltMain" url="/main" pattern="/">
        <bc_node title="Help" url="/forums" pattern="/forums?id.*"/>
    </bc_node>

</BreadCrumb>
```

As you can see – the help forums has two parents: “MyMain” and “AltMain”

If the user types in “http://www.xyz.com/forums?id=1”, the component will find the shortest path back to a root and select the following trail:

“AltMain : Help”

5. Future Enhancements

Additional tags for different portal implementations (JBoss, Pluto, etc.)