upstate.edu

Lesson 9: Cascade API Objects

14-18 minutes

Learning Objectives

- Understand the nature of Cascade API objects
- Learn how to:
- Use the API documentation
- Work with metadata
- Work with different types of blocks
- Work with Velocity formats
- Work with sites

Collapse all

XML and `$_XPathTool`

- There are at least two ways to retrieve block/page data:
- Using a block and a page region
- Use the Locator Tool (`$_`)
- A block (other than a text block), when attached to a page region, returns an XML tree
- The root node of the tree is the value associated with the variable `$contentRoot`
- Important: this approach always assumes the existence of a block attached to a page region
- Without a page region attached with a block, it is meaningless to talk about the variable `$contentRoot`
- To get the page XML data from an associated data definition, an index block must be attached to the DEFAULT region
- The type of nodes in an XML tree is `org.jdom.Element`
- When dealing with `org.jdom.Element` objects, we are working with an XML tree
- When working with such a tree, we use the `$_XPathTool` object, with its two `static` methods, namely `selectNodes` and `selectSingleNode`, and XPath expressions, to traverse the tree and retrieve XML nodes
- When nodes are selected, we use `org.jdom.Element` methods like `getChild(String)`, `getChildren()`, `getChildren(String)`, `getAttribute(String)`, `getAttributes()`, `getAttributes(String)`, and

`getValue()` to work with these nodes

- Element

The Nature of Cascade API Objects

- When using the Locator Tool and its `locate` or `locate`*X* methods to retrieve data, what we get are Cascade API objects
- The Cascade API classes are proprietary classes from Hannon Hill
- In a sense, the Cascade API is meant to replace indexing and the use of page regions
- Generally speaking, Cascade API objects have nothing to do with XML trees; there are some minor exceptions
- Therefore, when working with Cascade API objects, we do not work with `$contentRoot` or XML
- A Cascade API object is an object encapsulating data, along with a set of methods, with which we can work with the data
- To retrieve data encapsulated in a Cascade API object, call a `get`*X* method or `x` property
- Both method names and property names are case-sensitive, though a property name can be in camelCase
- Although the Cascade API classes also provide `set`*X* methods, we are not supposed to use them, and they never work in the context of Velocity
- Important: only methods defined in these Cascade API classes can be used to work with encapsulated data; the `$_XPathTool` object should not be used to work with them (again, with some minor exceptions), and the `$contentRoot` variable is never defined
- Therefore, to work with these objects efficiently, we should have the Cascade API documentation at hand

- cascade.api.adapters

Understanding the Cascade API Documentation

- What we call the Cascade API is a family of Java classes related by inheritance
- The ultimate ancestor class (not considering `java.lang.Object`) of all other classes is `com.hannonhill.cascade.api.adapters.BaseAssetAPIAdapter`
- All other classes are descendents of this class
- Every class defines a set of methods
- By means of inheritance, a descendent class also inherits all methods from its ancestor classes
- Therefore, when working with Cascade API objects, we need to know what methods are available to such an object, both methods directly defined in its corresponding class and methods inherited from its ancestors
- Consider `com.hannonhill.cascade.api.adapters.PageAPIAdapter`; this is the inheritance hierarchy of the class:
  `java.lang.Object`

  `com.hannonhill.cascade.api.adapters.BaseAssetAPIAdapter`

  `com.hannonhill.cascade.api.adapters.NamedAssetAPIAdapter`

  `com.hannonhill.cascade.api.adapters.PermissionsCapableAssetAPIAdapter`

  `com.hannonhill.cascade.api.adapters.FolderContainedAssetAPIAdapter`

  `com.hannonhill.cascade.api.adapters.MetadataAwareAssetAPIAdapter`

  `com.hannonhill.cascade.api.adapters.PublishableAssetAPIAdapter`

  `com.hannonhill.cascade.api.adapters.PageAPIAdapter`
- When working with a `com.hannonhill.cascade.api.adapters.PageAPIAdapter` object, like the one associated with `$currentPage`, we need to know that there are methods like `getStructuredData()` and `getStructuredDataNodes( java.lang.String )` defined in the class
- But there are also methods like `getLink()` and `getShouldBePublished()` inherited from `com.hannonhill.cascade.api.adapters.PublishableAssetAPIAdapter`, `getMetadata()` and `getLabel()` inherited from `com.hannonhill.cascade.api.adapters.MetadataAwareAssetAPIAdapter` and so on

- That means that when we look at a documentation page, we need to look at not only methods defined in a class, but also other methods inherited from its ancestors
- When looking for a method to call, search such a documentation page for keywords like `link` and `metadata` for related methods; they may be listed near the bottom of the page
- When calling such a method, look carefully at the returned type, if it is not `void`, and we may need to consult some other documentation pages
- For example, the returned type of `$currentPage.getStructuredData()` is `com.hannonhill.cascade.api.asset.common.StructuredDataNode[]`
- `com.hannonhill.cascade.api.asset.common.StructuredDataNode` is an interface which is implemented by `com.hannonhill.cascade.api.adapters.StructuredDataNodeAPIAdapter`
- `[]` indicates that what is returned is an array (not an `ArrayList`), storing `com.hannonhill.cascade.api.asset.common.StructuredDataNode` objects

Working with Metadata

- Call `getMetadata()` or use the `Metadata` property to get the `com.hannonhill.cascade.api.adapters.MetadataAPIAdapter` object

```
#set( $block = $_.locateBlock( "_cascade/blocks
/code/test-metadata", "formats" ) )
#set( $m     = $block.Metadata )
```

- To retrieve values of wired fields, use one of the following methods: `getDisplayName()` (or `DisplayName`), `getDescription()` (or `Description`), `getTitle()` (or `Title`), `getSummary()` (or `Summary`), `getTeaser()` (or `Teaser`), `getKeywords()` (or `Keywords`), `getAuthor()` (or `Author`), `getReviewDate()` (or `ReviewDate`), `getStartDate()` (or `StartDate`), and `getEndDate()` (or `EndDate`)

```
$m.DisplayName
$m.Title
$m.Summary
$m.Description
$m.Author
$m.Keywords
$m.Teaser
$m.StartDate
$m.EndDate
$m.ReviewDate
```

- The `getLabel` method (or `Label` property) is defined in `com.hannonhill.cascade.api.adapters.MetadataAwareAssetAPIAdapter`
- There are no methods defined to work with expiration folders

- To get the dynamic fields, use `getDynamicFields()` (`DynamicFields`), which returns an array of `com.hannonhill.cascade.api.asset.common.DynamicMetadataField` (an interface, implemented by `com.hannonhill.cascade.api.adapters.DynamicMetadataFieldImpl`) objects
```
#set( $dfs = $m.DynamicFields )
```
- To get a single dynamic field, pass in its identifier to `getDynamicField(java.lang.String)`:
```
#set( $languages = $m.getDynamicField( "languages"
) )
```
- Given a `com.hannonhill.cascade.api.adapters.DynamicMetadataFieldImpl` object, we can use one of those `is`*X* methods to test its type:
```
#if( $languages.Multiselect )
#end
```
- There are four `is`*X* methods: `isRadio()`, `isDropdown()`, `isCheckbox()`, and `isMultiselect()`
- Note that there is no `isText()` method; to make sure that a dynamic field is a text field, the dynamic field must fail all four `is`*X* tests
- When a dynamic field is either a checkbox or a multiselect, call the `getValues()` method to get the selected items; `getValues()` returns an array of Strings
- Otherwise, call the `getValue()` method to get the String value
```
#foreach( $df in $m.DynamicFields )
$df.Name
#if( $df.Checkbox || $df.Multiselect )
$df.Values.size()
#foreach( $value in $df.Values )
$value
#end
#else
$df.Value
#end
#end
```

Working with Text Blocks

- When working with a text block, call `getText()` (or `Text`) to retrieve the text:
```
#set( $block = $_.locateBlock( "_cascade/blocks
/code/test-text", "formats" ) )
$block.Text
```
- Important: When working with a text block, pay attention to the context
- When `$block.Text` is used in an XML environment, the text could be wrapped inside an `<system-xml>` element

- Assuming that the text block we are working with stores the String "Not Hello!", the code snippet will output the String
- But if we add more code to output the content of an XML block as well:

```
## a text block
#set( $block = $_.locateBlock( "_cascade/blocks
/code/test-text", "formats" ) )
$block.Text
## an XML block
#set( $block = $_.locateBlock( "_cascade/blocks
/script/advanced-lesson-1-code-script", "formats"
) )
$block.XML
```

This code snippet outputs:

```
<system-xml>Not Hello!</system-xml>
<scripts>
        <code>
#import( "site://_brisk/core/library/velocity
/chanw/chanw-library-import" )
#chanwGetMacroCode( "chanw-initialization"
"core/library/velocity/chanw" "_brisk"
"chanwProcessDataDefinitionBlock" )
    </code>
</scripts>
```

Working with XML Blocks

- When working with an XML block, call `getXML()` (or `XML` to retrieve the XML markups
- The `getXMLAsXMLElement()` method (or `XMLAsXMLElement` property) of `com.hannonhill.cascade.api.adapters.XMLBlockAPIAdapter` returns an `org.jdom.Element` object; and this can cause confusion because `$_XPathTool` can be use to work with this `org.jdom.Element` object

Working with Feed Blocks

- When working with a feed block, call `getFeedURL()` (or `FeedURL`) to get the URL String
- Call `getFeed()` (or `Feed`) to get the feed XML as a String
- Call `getFeedAsXMLElement()` (or `FeedAsXMLElement`) to get the feed as an `org.jdom.Element` object

```
#set( $block = $_.locateBlock( "bluepages/bp-
feed/feed1", "Upstate-Globals" ) )
$_EscapeTool.xml( $block.FeedURL )
$block.Feed
$block.FeedAsXMLElement.Class.Name
```

- Again, `$_XPathTool` can be use to work with this `org.jdom.Element` object; this may cause some confusion

Working with XHTML Blocks

- A simple way to distinguish an XHTML block from a data definition block is that the `StructuredData` property is undefined for an XHTML block
- Call `getXHTML()` (or `XHTML`) to get the XHTML markups as a String
- Call `getXHTMLAsXMLElement()` (or `XHTMLAsXMLElement`) to get the XHTML markups as an `org.jdom.Element` object

```
#set( $block = $_.locateBlock( "_cascade/blocks
/data/xhtml", "formats" ) )
#if( $block.StructuredData.Class.Name )
A data definition block
#else
$block.XHTML
$block.XHTMLAsXMLElement.Class.Name
#end
```

Index Blocks

- As pointed out above, the use of `$_` and the Cascade API is in a way a replacement of indexing
- To further disassociate the Cascade API from indexing, we need to know that a located index block is absolutely useless because the `com.hannonhill.cascade.api.adapters.IndexBlockAPIAdapter` class does not define any method, not even one, for us to use

Distinguishing `org.jdom.Element` Objects from Cascade API Objects

- A `org.jdom.Element` object represents a node in an XML tree; the Cascade API objects generally do not deal with XML
- Each Cascade API class defines its own set of methods, and these methods generally have nothing to do with XML
- `org.jdom.Element` objects can be returned by three methods of three Cascade API classes:
- `com.hannonhill.cascade.api.adapters.FeedBlockAPIAdapter.getFeedAsXMLElement`
- `com.hannonhill.cascade.api.adapters.XHTMLDataDefinitionBlockAPIAdapter.getX`
- `com.hannonhill.cascade.api.adapters.XMLBlockAPIAdapter.getXMLAsXMLElement(`
- That is to say, unless we call one of these three methods, we will never see `org.jdom.Element` objects in the context of Cascade API
- When dealing with an `org.jdom.Element` object, `getChild` and `getChildren` return `org.jdom.Element` objects
- These two methods do NOT accept XPath expressions as

arguments; instead, they only take element names

- To get the text of an `org.jdom.Element` object, use `getValue` (or `Value`)
- When dealing with `com.hannonhill.cascade.api.adapters.StructuredDataNodeAPIAdapter` objects, we do not have XML
- It is only meaningful to call `getChild` and `getChildren` methods through a `com.hannonhill.cascade.api.adapters.StructuredDataNodeAPIAdapter` object of type group; and the methods return `com.hannonhill.cascade.api.adapters.StructuredDataNodeAPIAdapter` objects
- These methods accept XPath expressions as arguments
- Only `com.hannonhill.cascade.api.adapters.StructuredDataNodeAPIAdapter` objects of type text store text values
- To retrieve these values, use `getTextValue` (or `TextValue`) and `getTextValues` (or `TextValues`); watch out for null values
- `com.hannonhill.cascade.api.adapters.StructuredDataNodeAPIAdapter` objects of type asset are choosers
- Use `getAsset` (or `Asset`) to get the asset attached to a chooser
- Such an asset is of type `com.hannonhill.cascade.api.asset.home.FolderContainedAsset`
- If there is no asset attached to such a chooser, the method returns null
- Use `com.hannonhill.cascade.api.adapters.StructuredDataNodeAPIAdapter.AssetIdent` to find out the type of the asset attached to a chooser

Working with Velocity Formats

- At least in one way the Locator Tool is more powerful than indexing: the Locator Tool can be used to locate formats
- A located Velocity format can be evaluated
- To locate a format:

```
#set( $format = $_.locate(
    "_cascade/formats/test-blocks",
    $_FieldTool.in(
"com.hannonhill.cascade.model.dom.identifier.EntityTypes"
).TYPE_FORMAT,
    "formats" ) )
```

- Use `getScript` (or `Script`) to access the code of a located format
- Importing vs. evaluating:
- There is an important difference between importing a Velocity

format and evaluating the code of a located Velocity format

- A format can be imported only once
- If the format contains executable code (code not inside macros or `#define`), the code can be executed only once
- If we want to execute the code more than once, then we have to locate the format and evaluate the code
- The code can be evaluated again and again
- When a format is documented in a specific way, we can even extract part of the code within the format and evaluate only that part
- See Lesson 5: Documenting Reusable Code for details

Working with Sites

- A `com.hannonhill.cascade.api.adapters.SiteAPIAdapter` object represents a site
- Such an object is retrievable through a `com.hannonhill.cascade.api.adapters.PageAPIAdapter` object by using `getSite` (or the `Site` property)
- Assuming that a site should have an index page in the base folder, we can locate this index page to get to the site object
- As of Cascade 8.7, there are four methods defined in this class:
- `java.lang.String getUrl()`
- `com.hannonhill.cascade.model.dom.NamingRuleCase getNamingRuleCase()`
- `com.hannonhill.cascade.model.dom.NamingRuleSpacing getNamingRuleSpacing()`
- `[Ljava.lang.String; getNamingRuleAssets()`
- Code example:

```
#set( $site = $currentPage.Site )
$site.Class.Name
$site.Url
$site.NamingRuleCase     ## LOWER
$site.NamingRuleSpacing  ## HYPHEN
$site.NamingRuleAssets.size()   ## 2
```

- Note that the returned type of, for example, `getNamingRuleCase()` is `com.hannonhill.cascade.model.dom.NamingRuleCase`, not a String
- When the returned value is compared with a String, the `toString` method is called implicitly:

```
#set( $isLower = ( $site.NamingRuleCase == "LOWER" ) )
$isLower   ## true
```

Examples

- introductory/09_cascade_api_objects