MarkLogic Server

JavaScript Reference Guide

MarkLogic 8
February, 2015

Last Revised: 8.0-6, October, 2016

Copyright © 2016 MarkLogic Corporation. All rights reserved.

Table of Contents

JavaScript Reference Guide

1.0 Server-Side JavaScript in MarkLogic 4
1.1 Google V8 JavaScript Engine 4
1.2 Familiarity For the JavaScript Developer 5
1.3 Server-Side MarkLogic Power for Data Services 5
1.4 Dates in Server-Side JavaScript 5
1.5 Numeric Datatype Mappings in JavaScript 6
1.6 JavaScript in Query Console 7
1.7 Programming in Server-Side JavaScript 7
1.8 Using xdmp.invoke or xdmp.invokeFunction for Scripting 7

2.0 MarkLogic JavaScript Object API 8
2.1 Node and Document API 8
2.1.1 Node Object 8
2.1.2 Document Object 10
2.2 XML DOM APIs 10
2.2.1 Node Object for XML Nodes 11
2.2.2 Document Object for Document Nodes 13
2.2.3 NodeBuilder API 14
2.2.4 Element 16
2.2.5 Attr 18
2.2.6 CharacterData and Subtypes 18
2.2.7 TypeInfo 20
2.2.8 NamedNodeMap 20
2.2.9 NodeList 21
2.3 Value Object 21
2.4 Accessing JSON Nodes 21
2.5 ValueIterator 22
2.5.1 The ValueIterator Interface 22
2.5.2 Preparing to Switch from ValueIterator to Sequence 23
2.6 JavaScript instanceof Operator 25
2.7 JavaScript Error API 26
2.7.1 JavaScript Error Properties and Functions 27
2.7.2 JavaScript stackFrame Properties 27
2.7.3 JavaScript try/catch Example 28
2.8 JavaScript console Object 28
2.9 JavaScript Duration and Date Arithmetic and Comparison Methods 28
2.9.1 Arithmetic Methods on Durations 29
2.9.1.1 xs.yearMonthDuration Methods 29

2.9.1.2 xs.dayTimeDuration Methods 30
2.9.2 Arithmetic Methods on Duration, Dates, and Times 31
2.9.2.1 xs.dateTime Methods 32
2.9.2.2 xs.date Methods 34
2.9.2.3 xs.time Methods 35
2.9.3 Comparison Methods on Duration, Date, and Time Values 36
2.9.3.1 xs.yearMonthDuration Comparison Methods 37
2.9.3.2 xs.dayTimeDuration Comparison Methods 38
2.9.3.3 xs.dateTime Comparison Methods 39
2.9.3.4 xs.date Comparison Methods 40
2.9.3.5 xs.time Comparison Methods 41
2.9.3.6 xs.gYearMonth Comparison Methods 42
2.9.3.7 xs.gYear Comparison Methods 43
2.9.3.8 xs.gMonthDay Comparison Methods 44
2.9.3.9 xs.gMonth Comparison Methods 44
2.9.3.10 xs.gDay Comparison Methods 45
2.10 MarkLogic JavaScript Functions 46

3.0 JavaScript Functions and Constructors 47
3.1 Built-In JavaScript Functions 47
3.2 Functions That are part of the Global Object 47
3.2.1 declareUpdate Function 48
3.2.2 require Function 48
3.3 Using XQuery Functions and Variables in JavaScript 49
3.3.1 require Function 49
3.3.2 Importing XQuery Modules to JavaScript Programs 49
3.3.2.1 Mapping Between XQuery Function and Variable Names to JavaScript 50
3.3.2.2 Type Mapping Between XQuery and JavaScript 50
3.4 Importing JavaScript Modules Into JavaScript Programs 51
3.5 Other MarkLogic Objects Available in JavaScript 51
3.6 Amps and the module.amp Function 52
3.6.1 module.amp Function 52
3.6.2 Simple JavaScript Amp Example 52
3.7 JavaScript Type Constructors 53

4.0 Technical Support 57

5.0 Copyright 58
5.0 NOTICE 58

1.0 Server-Side JavaScript in MarkLogic

MarkLogic 8 integrates JavaScript as a first-class server-side programming language. You can call a JavaScript program from an App Server, and that program has server-side access to the MarkLogic built-in functions. This chapter describes the JavaScript implementation in MarkLogic and includes the following sections:

- Google V8 JavaScript Engine
- Familiarity For the JavaScript Developer
- Server-Side MarkLogic Power for Data Services
- Dates in Server-Side JavaScript
- Numeric Datatype Mappings in JavaScript
- JavaScript in Query Console
- Programming in Server-Side JavaScript
- Using xdmp.invoke or xdmp.invokeFunction for Scripting

1.1 Google V8 JavaScript Engine

MarkLogic Server integrates the Google V8 JavaScript engine (https://code.google.com/p/v8/), a high-performance open source C++ implementation of JavaScript.

The version of V8 MarkLogic embeds is 3.24, with the following flag enabled:

- `--harmony-iteration`
- `--harmony-generators`

This version of V8 offers some of the newer EcmaScript 6 features, one of which is used to implement `for ... of` loops, which are useful with APIs that return a `ValueIterator`, allowing functions that return large results sets to efficiently stream the results. For details, see “ValueIterator” on page 22.

EcmaScript 6 generators use the `function*` syntax. For a description of EcmaScript 6 generators, see documentation for implementaiton of generators such as https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/function* and http://wiki.ecmascript.org/doku.php?id=harmony:generators. For generators, MarkLogic only supports the `Generator.prototype.next()` method (which the `for ... of` loop uses), not the `Generator.prototype.return()` and `Generator.prototype.throw()` methods.

The following is a simple JavaScript generator example to run in MarkLogic:

```
function* gen(limit){
  for (var i = 0; i < limit; i++)
    yield xdmp.eval('xs.dateTime(new Date())');}
var result=[];
for (var i of gen(10)){
  result.push(i);}
result;
/* returns ten different dateTime values (because they are each run
   in a separate eval) */
```

1.2 Familiarity For the JavaScript Developer

JavaScript as a programming language has become extremely popular, and it is familiar to a huge number of developers. Over the last several years, JavaScript has expanded its footprint from the browser to other programming environments like Node.js. MarkLogic Server-Side JavaScript expands that familiarity one level deeper into the database server level. This allows you to combine the power of programming at the database level with the familiarity of JavaScript.

1.3 Server-Side MarkLogic Power for Data Services

With JavaScript running within MarkLogic, you can do processing of your data right at the server level, without having to transfer the data to a middle tier. In MarkLogic, you have always been able to do this with XQuery. In MarkLogic 8, you can do that same processing using JavaScript, for which most organizations have a lot of experienced programers.

1.4 Dates in Server-Side JavaScript

MarkLogic has many XQuery functions that return date values, using W3C standard XML dates and durations. These functions are all available in Server-Side JavaScript, and their values are returned in the XML types.

For the return value from any of these functions, you can call `toObject()` and the date values are converted into JavaScript UTC dates. This way you can use the powerful XML date and duration functions if you want to, and you can combine that with any JavaScript handling of dates that you might prefer (or that you might already have JavaScript code to handle). For reference material on JavaScript Date functions, see any JavaScript reference (for example, Mozilla). For the MarkLogic Server-Side JavaScript date functions, see http://docs.marklogic.com/js/fn/dates.

Consider the following example:

```
var results = new Array();
var cdt = fn.currentDateTime();
results.push(cdt);
var utc = cdt.toObject();
results.push(utc);
results;

=>
["2015-01-05T15:36:17.804712-08:00", "2015-01-05T23:36:17.804"]
```

In the above example, notice that the output from the `cdt` variable (the first item in the `results` array) retains the timezone information inherent in XML dateTime values. The output from the `utc` variable (the second item in the `results` array) no longer has the timezone shift, as it is now a UTC value.

Similarly, you can use any of the UTC methods on MarkLogic-based dates that are converted to objects, For example, the following returns the UTC month:

```
fn.currentDateTime().toObject()
  .getMonth();  // note that the month is 0-based, so January is 0
```

The following returns the number of milliseconds since January 1, 1970:

```
var utc = fn.currentDateTime().toObject();
Date.parse(utc);
// => 1420502304000 (will be different for different times)
```

The flexibility to use JavaScript date functions when needed and XML/XQuery date functions when needed provides flexibility in how you use dates in Server-Side JavaScript.

1.5 Numeric Datatype Mappings in JavaScript

In Server-Side JavaScript, you have full access to all of the rich datatypes in MarkLogic, including the numeric datatypes. In general, Server-Side JavaScript maps numeric datatypes in MarkLogic to a JavaScript `Number`. There are a few cases, however, where MarkLogic wraps the number in a MarkLogic numeric type instead of returning a JavaScript `Number`. Those cases are:

- If a value will overflow or might lose precision in a JavaScript `Number`, then MarkLogic wraps it in a numeric datatype (`xs.decimal`, for example).
- If the returned value contains frequency information (for example, a numeric value returned from a search), then it is wrapped in a numeric type such as `xs.integer`, `xs:double`, or `xs:float`.
- If the returned value is a JavaScript future, then MarkLogic wraps it in a numeric type.

1.6 JavaScript in Query Console

Query Console, which ships on port 8000 on default installations of MarkLogic, allows you to evaluate JavaScript using Server-Side JavaScript, making it is very easy to try out examples. For example, the following are each "hello world" examples that you can run in Query Console by entering the following (with JavaScript selected as the Query Type):

```
"hello world"

fn.concat("hello ", "world")
```

Both return the string `hello world`. For details about Query Console, see the *Query Console User Guide*.

1.7 Programming in Server-Side JavaScript

When you put a JavaScript module under an App Server root with a `sjs` file extension, you can evaluate that module via HTTP from the App Server. For example, if you have an HTTP App Server with a root `/space/appserver`, and the port set to 1234, then you can save the following file as `/space/appserver/my-js.sjs`:

```
xdmp.setResponseContentType("text/plain");
"hello"
```

Evaluating this module in a browser pointed to `http://localhost:1234/my-js.sjs` (or substituting your hostname for localhost if your browser is on a different machine) returns the string `hello`.

You cannot serve up a Server-Side JavaScript module with a `.js` file extension (`application/javascript` mimetype) directly from the App Server; directly served modules need a `.sjs` extension (`application/vnd.marklogic-javascript` mimetype). You can import JavaScript libraries with either extension, however, as described in "Importing JavaScript Modules Into JavaScript Programs" on page 51.

1.8 Using xdmp.invoke or xdmp.invokeFunction for Scripting

If you want to have a single program that does some scripting of tasks, where one task relies on updates from the previous tasks, you can create a JavaScript module that uses `xdmp.invoke` or `xdmp.invokeFunction`, where the calls to `xdmp.invoke` or `xdmp.invokeFunction` have options to make their contents evaluate in a separate transaction.

2.0 MarkLogic JavaScript Object API

This chapter describes the Object API built into Server-Side JavaScript in MarkLogic and includes the following sections:

- Node and Document API
- XML DOM APIs
- Value Object
- Accessing JSON Nodes
- ValueIterator
- JavaScript instanceof Operator
- JavaScript Error API
- JavaScript console Object
- JavaScript Duration and Date Arithmetic and Comparison Methods
- MarkLogic JavaScript Functions

2.1 Node and Document API

MarkLogic APIs often return or expect nodes and/or documents. To make it easy to use these APIs in JavaScript, MarkLogic has added the built-in `Node` and `Document` objects. These are objects but they are not part of the global object. This section describes the interface for these objects and includes the following parts:

- Node Object
- Document Object

2.1.1 Node Object

A `Node` can be any kind of node, such as an element node, a document node, a text node, and so on. If a function returns a Node in Server-Side JavaScript, you can examine the `Node` object using the following properties:

Property	Description
`baseURI`	A String representing the base URI of the node.
`valueOf()`	The atomic value of the node.

Property	Description	
nodeType	A number representing the type of the Node object. The following are meanings the possible values of `nodeType`:	
	ELEMENT_NODE	1
	ATTRIBUTE_NODE	2
	TEXT_NODE	3
	PROCESSING_INSTRUCTION_NODE	7
	COMMENT_NODE	8
	DOCUMENT_NODE	9
	BINARY_NODE	13
	NULL_NODE	14
	BOOLEAN_NODE	15
	NUMBER_NODE	16
	ARRAY_NODE	17
	OBJECT_NODE	18
`toObject()`	JavaScript object if the node is type `Array`, `Boolean`, `Number`, `Object` or `Text`; otherwise it is `Undefined`.	
`xpath(String XPathExpression, Object NamespaceBindings)`	Evaluate the XPath expression. The first argument is a string representing the XPath expression, and the second argument is an Object where each key is a namespace prefix used in the first argument, and each value is the namespace in which to bind the prefix. For the XPath expression, if you want the expression evaluated relative to the current node, start the path with a dot (.); for example, `"./my-node"`.	

For additional DOM properties available on XML nodes (document, element, attribute, processing instruction, and comment), see “Node Object for XML Nodes” on page 11.

The following is an example of using the `xpath` function on a Node object. The `cts.doc` function returns a Node, specifically a Document node, which inherits an `xpath` method from Node. The second parameter to `Node.xpath` binds the XML namespace prefix “bar” to the XML namespace URI “bar”.

```
// assume a document created as follows:
declareUpdate();
xdmp.documentInsert("/my/doc.xml", fn.head(xdmp.unquote(
  '<bar:foo xmlns:bar="bar"><bar:hello><bar:goodbye \n\
```

```
      attr="attr value">bye</bar:goodbye>\n\
  </bar:hello>\n\
</bar:foo>')));

// Use the Node.xpath method on the document:
var node = cts.doc("/my/doc.xml");
node.xpath("//bar:goodbye/@attr", {"bar":"bar"});

// Running in Query Console displays the following value (as an
// attribute node): "attr value"
```

2.1.2 Document Object

The `Document` object inherits all of the properties from the Node Object above, and has the following additional properties:

Property	Description	
`documentFormat`	A string representing the format of the document node. The following are the meanings of the possible values of `documentFormat`:	
	`BINARY`	`"BINARY"`
	`JSON`	`"JSON"`
	`TEXT`	`"TEXT"`
	`XML`	`"XML"`
`root`	The root node of the document.	

2.2 XML DOM APIs

MarkLogic implements XML DOM APIs to provide read-only access to XML nodes. This section describes these APIs and includes the following parts:

- Node Object for XML Nodes
- Document Object for Document Nodes
- NodeBuilder API
- Element
- Attr
- CharacterData and Subtypes
- TypeInfo
- NamedNodeMap

- NodeList

2.2.1 Node Object for XML Nodes

In addition to the `Node` properties described in “Node Object” on page 8, the XML node types all have a subset of the W3C DOM API of `Node`, as follows:

Properties	Description
`childNodes`	An Iterator that contains all children of this node.
`firstChild`	The first child of this node. If there is no such node, returns null. Note that it returns the first child node of any kind, not just the first element child, so if the first child is a text node with empty space in it that is what is returned.
`lastChild`	The last child of this node. If there is no such node, returns null. Note that it returns the last child node of any kind, not just the last element child, so if the last child is a text node with empty space in it that is what is returned.
`localname`	Returns the localname part of the qualified name (QName) of this node. For nodes of any type other that `ELEMENT_NODE` or `ATTRIBUTE_NODE`, this always returns null.
`namespaceURI`	The namespace URI of this node, or null if it is unspecified. For nodes of any type other that `ELEMENT_NODE` or `ATTRIBUTE_NODE`, this always returns null.
`nextSibling`	The node immediately following this node. If there is no such node, returns null.
`nodeName`	The name of this node, depending on its type.
`nodeValue`	The value of this node, depending on its type.
`ownerDocument`	The document the node belongs to.
`parentNode`	Node that is the parent of the node.
`prefixSibling`	Node representing the previous node in the tree, or null if no such node exists.
`hasChildNodes()`	Boolean indicating if the node has child nodes.
`hasAttributes()`	Boolean indicating if the node has any attributes.

Properties	Description
`attributes`	`NamedNodeMap` of all the attributes, if any. For nodes of any type other than `ELEMENT_NODE` this map will be empty.
`baseURI`	The base URI of this node, if any.
`textContent`	Like `fn.string` on the node except that document nodes are `null`.
`isSameNode( Node other)`	Returns true if the two nodes are the same (similar to the XQuery operator `=` on nodes).
`isEqualNode( Node other)`	Returns true if the two nodes are the equal (similar to the XQuery `fn:deep-equals`, but treating everything as untyped).
`insertBefore( Node newChild, Node refChild)`	Raises NO_MODIFICATION_ALLOWED error.
`replaceChild( Node newChild, Node oldChild)`	Raises NO_MODIFICATION_ALLOWED error.
`removeChild( Node oldChild)`	Raises NO_MODIFICATION_ALLOWED error.
`appendChildNodes( Node newChild)`	Raises NO_MODIFICATION_ALLOWED error.
`normalize()`	Does nothing (MarkLogic documents are already normalized).

The DOM APIs provide read-only access to the XML nodes; any DOM APIs that attempt to modify the nodes will raise the DOM error `NO_MODIFICATION_ALLOWED_ERR`.

2.2.2 Document Object for Document Nodes

The `Document` object inherits all of the properties from the Node Object for XML Nodes above (in addition to the properties from the Node Object), and has the following additional properties:

Properties	Description
`documentElement`	Element that is the direct child of the document.
`documentURI`	The URI of the document.
`getElementsByTagName(` `  String tagname)`	NodeList of elements in the document with the given tag name, in document order. The tag name is a string. If it includes a colon, it will match as a string match with the exact prefix. The `getElementsByTagNameNS` function is preferred for namespaced elements.

`getElementsByTagNameNS(`
`  String namespaceURI,`
`  localname)`
	NodeList of elements in document with the given namespace URI and localname, in document order. A null value for the namespace URI signifies no namespace.

`getElementById(` `  String elementId)`	Element that has the given ID, if any.

`importNode(`
`  Node importedNode,`
`  Boolean deep)`
	Raises NO_MODIFICATION_ALLOWED error.

`normalizeDocument()`	Does nothing (MarkLogic documents are already normalized).

2.2.3 NodeBuilder API

The NodeBuilder API makes it easy to construct nodes in JavaScript, including XML nodes and JSON nodes, and has the following functions and properties:

Functions/Properties	Description
`addAttribute(String name, String value, [String URI])`	Add a new attribute to the current element being created. You cannot create duplicate attributes; if an attribute with that name already is present in the element, XDMP-DUPATTR is thrown.
`addComment(String text)`	Add a comment node to the current parent node being created.
`addDocument(String text, [String URI]))`	Add a document with the given URI and the specified text content. This results in a document of format text (that is, document node with a text node root).

`addDocument( Function content, [String URI]))`	Add a document with the given URI. The function will be given the builder as its argument and evaluated to produce the content. For example:
	`var x = new NodeBuilder();`
	`var b = x.addDocument(`
	`  function(builder){`
	`  builder.addElement("foo",`
	`  "some stuff")});`
	`b.toNode().root;`
	`=>`
	`<foo>some stuff</foo>`

`addElement(String name, String text, [String URI])`	Add an element to the current parent node with the specified name, text content, and namespace URI. The function will be given the builder as its argument and evaluated to produce the content. The element creation is completed after calling `addElement`, and consequently subsequent calls to `addAttribute` would not apply to this element.

Functions/Properties	Description

`addElement(String name, Function content, [String URI])`	Add an element to the current parent node with the specified name and namespace URI. The element creation is completed after calling `addElement`, and consequently subsequent calls to `addAttribute` would not apply to this element. The function in the second argument will be given the builder as its argument and evaluated to produce the content. For example:
	`var x = new NodeBuilder();`
	`var b = x.addElement("foo",`
	`  function(builder){`
	`  builder.addText("some stuff")});`
	`b.toNode();`
	`=>`
	`<foo>some stuff</foo>`

`addNode(Node node)`	Add a copy of the specified node to the current parent node.
`addProcessingInstruction( String target, String text)`	Add a processing instruction node to the current parent node with the specified target and text.
`addText(String value)`	Add a text node to the current parent node being created.
`addBinary(String hex)`	Add a binary node to the current parent node being created. The argument is a hex encoded string.
`addNumber(Number val)`	Add a number node to the current parent node being created.
`addBoolean(Boolean val)`	Add a boolean node to the current parent node being created.
`addNull()`	Add a null node to the current parent node being created.
`endDocument()`	Complete creation of the document.
`endElement()`	Complete creation of the element.
`startDocument([String URI])`	Start creating a document with the specified URI.
`startElement(String name, [String URI])`	Start creating an element as a child of the current document or element with the specified name and (optionally) namespace URI.
`toNode()`	Returns the constructed node.

In order to use anything created with NodeBuilder as a node, you must first call `toNode()`.

The following is an example of creating an XML document node:

```
var x = new NodeBuilder();
x.startDocument();
x.startElement("foo", "bar");
x.addText("text in bar");
x.endElement();
x.endDocument();
var newNode = x.toNode();
newNode;
// returns a document node with the following serialization:
// <?xml version="1.0" encoding="UTF-8"?>
// <foo xmlns="bar">text in bar</foo>
```

2.2.4 Element

The following properties and functions are available on element nodes:

Properties/Functions	Description
`tagName`	Qualified name of the element.
`getAttribute(String name)`	Returns an attribute value by name.
`getAttributeNode(String name)`	Returns an attribute node (`Attr`) by name.

`getAttributeNS(`
`  String namespace,`
`  String name)`
	Returns the value of the attribute with the specified namespace and name, from the current node.

`getAttributeNode(String name)`	Return the named attribute of this element, if any, as a node.

`getAttributeNodeNS(`
`  String namespaceURI,`
`  String localname)`
	Return the attribute of this element with a matching namespace URI and localname.

`getElementsByTagName(` `  String tagname)`	`NodeList` of element descendants of this element with the given tag name, in document order. The tag name is a string. If it includes a colon, it will match as a string match with that exact prefix. `getElementsByTagNameNS` is preferred for namespaced elements.

`getElementsByTagNameNS(`
`  String namespaceURI,`
`  String localname)`
	`NodeList` of element descendants with the given namespace URI and localname, in document order. A null value for the namespace URI signifies no namespace.

`hasAttribute(String name)`	Returns true if the element has the named attribute.

Properties/Functions	Description
`hasAttributeNS( String namespaceURI, String localname)`	Returns true if the element has an attribute with the given namespace URI and localname.
`schemaTypeInfo`	`TypeInfo` of the element.
`setAttribute(String name, String value)`	Raises NO_MODIFICATION_ALLOWED error.
`removeAttribute(String name)`	Raises NO_MODIFICATION_ALLOWED error.
`setAttributeNode(Attr newAttr)`	Raises NO_MODIFICATION_ALLOWED error.
`removeAttributeNode( Attr newAttr)`	Raises NO_MODIFICATION_ALLOWED error.
`setAttributeNS( String namespaceURI, String localname)`	Raises NO_MODIFICATION_ALLOWED error.
`removeAttributeNS( String namespaceURI, String localname)`	Raises NO_MODIFICATION_ALLOWED error.
`setIdAttribute( String name, Boolean isId)`	Raises NO_MODIFICATION_ALLOWED error.
`setIdAttributeNS( String namespaceURI, String localname, Boolean isId)`	Raises NO_MODIFICATION_ALLOWED error.
`setIdAttributeNode( Attr idAttr, Boolean isId)`	Raises NO_MODIFICATION_ALLOWED error.

2.2.5 Attr

The following properties are available on attribute (`Attr`) nodes, in addition to the XMLNode properties which it inherits:

Properties	Description
`name`	Qualified name of this attribute.
`specified`	Boolean indicating whether the attribute is explicit (true) or defaulted from the schema (false).
`value`	Value of this attribute, as a string.
`ownerElement`	Element that has the attribute.
`isId`	Boolean indicating whether this is an ID attribute. (It has the type `xs:ID`).
`schemaTypeInfo`	`TypeInfo` of the element.

The `Attr` object is being deprecated in DOM4.

2.2.6 CharacterData and Subtypes

The CharacterData inherits all of the APIs from an XMLNode plus the following additional properties and methods. It has subtypes that inherit from Text node, Comment nodes, and Processing Instruction nodes, which are also included in the table.

Functions/Properties	Description
`data`	The textual content of the node (same as `fn:data`).
`length`	The number of characters in the textual content of the node.
`substringData(Number offset, Number count)`	Substring of the textual content, starting at the given character offset and continuing for the given number of characters.
`isElementContentWhitespace`	True if the Text node is ignorable whitespace. In a MarkLogic context this is almost always false as MarkLogic strips ignorable whitespace on ingest. It can be true of data were ingested before a schema for it was loaded. (Text node only).

Functions/Properties	Description
`wholeText`	Returns the value of this node concatenated with logically adjacent text nodes. For MarkLogic, because it already combines logically adjacent text nodes, this is just the value of the node itself. (Text node only)
`target`	The target of the processing instruction. For example, given the PI `<?example something?>`, `example` is the target and `something` is the data.
`appendData(String arg)`	Raises NO_MODIFICATION_ALLOWED error.
`insertData(Number offset, Number count)`	Raises NO_MODIFICATION_ALLOWED error.
`deleteData(Number offset, Number count)`	Raises NO_MODIFICATION_ALLOWED error.
`replaceData(Number offset, Number count, String arg)`	Raises NO_MODIFICATION_ALLOWED error.
`replaceWholeText( String content)`	Raises NO_MODIFICATION_ALLOWED error. (Text node only)
`splitText(Number offset)`	Raises NO_MODIFICATION_ALLOWED error. (Text node only)

2.2.7 TypeInfo

The following are the functions and properties of TypeInfo. Additionally, it has the schema component methods bound to it.

Functions/Properties	Description
`typeName`	The local name of the type.
`typeNamespace`	The namespace URI of the type.

`isDerivedFrom(`
`  String typeNamespace,`
`  String typeName,`
`  unsigned long`
`  derivationMethod)`
	Returns true if this type is derived from the type named by the arguments. The derivation method argument is a flag indicating acceptable derivation methods (0 means all methods are acceptable). The flag values that may be combined are:
	DERIVATION_RESTRICTION (0x1)
	DERIVATION_EXTENSION (0x2)
	DERIVATION_UNION (0x4)
	DERIVATION_LIST (0x8)

2.2.8 NamedNodeMap

The following are the functions and properties of NamedNodeMap.

Functions/Properties	Description
`length`	Number of nodes in the map.
`getNamedItem(name)`	Returns the node in the map with the given name, if any. `getNamedItemNS` is preferred for namespaced nodes.

`getNamedItemNS(`
`  String namespaceURI,`
`  String localName)`
	Returns the node in the map with the given namespace URI and localname.

`item(Number index)`	Get the node at the index place (first, second, and so on).

Functions/Properties	Description
`setNamedItem(Node arg)`	Raises NO_MODIFICATION_ALLOWED error.
`removeNamedItem(` `  String name)`	Raises NO_MODIFICATION_ALLOWED error.
`setNamedItemNS(Node arg)`	Raises NO_MODIFICATION_ALLOWED error.

`removeNamedItem(`
`  String namespaceURI,`
`  String localname)`
	Raises NO_MODIFICATION_ALLOWED error.

2.2.9 NodeList

The `NodeList` is an iterator that has the following additional properties.

Properties	Description
`length`	Number of nodes in the list.
`item(Number index)`	Get the item at the index place (first, second, and so on).

2.3 Value Object

The `Value` object is a wrapper object that wraps MarkLogic XQuery types, allowing you to pass these objects into functions that are designed for those types. The DateTime functions are good example. For example, consider the following:

```
fn.currentDate() instanceof Value
```

This returns `true`, because it is like the XML type `xs:date` (which prints out as `xs.date` in JavaScript). Many of the MarkLogic built-in JavaScript functions return as this Value object. For the methods available on this object, see *Value Object*. For more details about dates, see Dates in Server-Side JavaScript.

The `Value` object allows you to use objects and take advantage of the types used in XQuery functions from JavaScript. Additionally, you can also call `toObject()` on a `Value` object to turn it into its closest JavaScript native type.

2.4 Accessing JSON Nodes

When you store JSON in a MarkLogic database, it is stored as a document node with a JSON node child. You can access JSON documents stored in the database with `fn.doc`, or with any other function that returns a document. You have direct read-only access to the JSON nodes through the native JavaScript properties, including get a named property, querying for the existence of a named property, and enumerate all available named properties.

If you want to convert a JavaScript object to a JSON node, you can call `xdmp.toJson` on the JavaScript object and it will return a JSON node.

For more details about JSON nodes and documents, see Working With JSON in the *Application Developer's Guide*.

2.5 ValueIterator

Built-in and library module functions that return result sets to be iterated over are returned as a `ValueIterator` in Server-Side JavaScript. `ValueIterator` will be deprecated and replaced with a new `Sequence` interface in future versions of MarkLogic, but `ValueIterator` will remain for the life of MarkLogic 8.

This topic describes how to use `ValueIterator` today, and how to make your code flexible enough to work with Sequence in the future.

- The ValueIterator Interface
- Preparing to Switch from ValueIterator to Sequence

2.5.1 The ValueIterator Interface

A `ValueIterator` is an extension of a JavaScript `Iterator`, having all of the `Iterator` properties plus some additional ones.

> **Note:** The `ValueIterator` object will be deprecated in future versions of MarkLogic, but will remain for the life of MarkLogic 8. For details, see "Preparing to Switch from ValueIterator to Sequence" on page 23.

The following table shows the properties and functions you can apply to objects that are typed as `ValueIterator`:

Properties/Functions	Description
`count`	Returns the number of remaining values in the iterator.
`clone()`	Returns a clone of the iterator. This is useful if you need to make a copy of a node so that it does not hold onto a reference to a document in the database.
`next()`	Returns the next item in the iterator. This function has two properties: `value` and `done`.
`toArray()`	Returns an array of value items from the remaining values in the iterator. If there are no more values, it returns an empty array.

Iterate over the items in a ValueIterator using a `for...of` loop. The following example, returns the first 2 documents in a search, along with their URIs. The `results` array is just a convenience for displaying the final resutls if you run the example in Query Console.

```
// gather up results into an array
var results = new Array();
// first two search results
// empty cts.andQuery matches every document
var res = fn.subsequence(cts.search(cts.andQuery([])), 1, 2);
// loop thru ValueIterator (using for ... of loop) and push
// result and URI into results array
for (var x of res) {
  results.push(x);
  results.push(x.nodeUri);
};
// print out the results
results;
```

If you want to construct a `ValueIterator` in your own code, either run a function that returns a `ValueIterator` or use the `xdmp.arrayValues` function.

2.5.2 Preparing to Switch from ValueIterator to Sequence

The next major release of MarkLogic will deprecate `ValueIterator` in favor of a new interface named `Sequence`, which is a JavaScript Iterable. All MarkLogic interfaces that use a `ValueIterator` in MarkLogic 8 will use a `Sequence` in MarkLogic 9. It will no longer be possible to create a `ValueIterator` or receive one as a function result.

Code that manipulates `ValueIterator` results as described in the following table will continue to work in the future when function return types change to `Sequence`.

Guideline	Do	Do Not

Only use a `for...of` loop to iterate over the values in a `ValueIterator` returned by MarkLogic. Do not program directly to the underlying Iterator interface	`var uris = cts.uris('/');`
	`for (var u of uris) {`
	`  // do something with u`
	`}`
		`var uris = cts.uris('/');`
		`uris.next.value();`
		`uris.next.value();`
		`...`

Use `fn.head` to access the first item in a `ValueIterator`, rather than using the pattern *`results`*`.next().value`.	`fn.head(`
	`  cts.uris('/')`
	`);`
		`cts.uris('/').next().value`

Use `fn.count` to count the number of items in a `ValueIterator` object, rather than the `count` property.	`fn.count(`
	`  cts.uris('/')`
	`);`
		`cts.uris('/').count`

Code that depends on the `ValueIterator` properties and methods `next`, `count`, and `clone` will not work in the future. (You never need to clone a `Sequence` because you can iterate over the same `Sequence` object repeatedly.)

You can use the `instanceof` operator to create code that behaves differently, depending on whether you are working with a `ValueIterator` or a `Sequence`. No MarkLogic 8 functions will return a Sequence, so you can be certain that code will not execute in MarkLogic 8.

For example, the following code uses `ValueIterator.clone` to preserve a set of values across iteration in MarkLogic 8, but skips the unnecessary cloning when the result type becomes Sequence.

```
var results = cts.uris('/', ['limit=10']);
var clone = {};
if (uris instanceof ValueIterator) {
  // iterator destructive, so clone to preserve orig
  clone = results.clone();
} else if (results instanceof Sequence) {
  // iteration is not destructive, no clone needed
  clone = results;
}
for (var val of clone) {
```

```
  // do something with val
}
```

2.6 JavaScript instanceof Operator

The JavaScript `instanceof` operator is available to test MarkLogic types (in addition to JavaScript types). For example, the following returns true:

```
var a = xdmp.arrayValues(["saab", "alfa romeo", "tesla"]);
a instanceof ValueIterator;
// returns true
```

Similarly, the following are some other examples of using `instaceof` with MarkLogic and JavaScript object types.

An `xs.date` object type:

```
var a = fn.currentDate();
a instanceof xs.date;
// returns true
```

Not an `xs.date` object type:

```
var a = fn.currentDate().toObject();
a instanceof xs.date;
// returns false
```

A JavaScript `Date` object type:

```
var a = fn.currentDate().toObject();
a instanceof Date;
// returns true
```

You can test for any of the following MarkLogic object types using `instanceof`:

- `Value` (all MarkLogic Object types are subtypes of `Value`)
- `xs.anyAtomicType`
- `cts.query` (and all of its subtypes—the subtypes are also instance of `cts.query`)
- `ArrayNode`
- `BinaryNode`
- `BooleanNode`
- `ObjectNode`
- `XMLNode`
- `Document`
- `Node`
- `NodeBuilder`
- `Attr`

- `CharacterData`
- `Comment`
- `Text`
- `Element`
- `ProcessingInstruction`
- `XMLDocument`
- `ValueIterator`

The following is an example using an XML document:

```
// assume "/one.xml" has content <name>value</name>
var a = fn.head(fn.doc("/one.xml")).root;
var b = fn.head(a.xpath("./text()"));
b instanceof Text;
// returns true
```

The following is an example using a JSON document:

```
// Assume "/car.json" has the content:
// {"car":"The fast electric car drove down the highway."}
var res = new Array();
var a = fn.head(fn.doc("/car.json"));
res.push(a instanceof Document);
var b = a.root;
res.push(b instanceof ObjectNode);
res;
// returns [true, true]
```

Similarly, you can test for any XML type. Note that the XML types in JavaScript have a dot (`.`) instead of a colon (`:`) between the namespace and the type name. For example, `xs.integer`, `xs.string`, and so on.

2.7 JavaScript Error API

When errors and exceptions are thrown in a Server-Side JavaScript program, a stack is thrown and can be caught using a standard JavaScript `try/catch` block. For details about each individual error message, see the *Messages and Codes Reference Guide*. This section includes the following parts:

- JavaScript Error Properties and Functions
- JavaScript stackFrame Properties
- JavaScript try/catch Example

2.7.1 JavaScript Error Properties and Functions

The following is the API available to JavaScript exceptions.

Properties/Functions	Description
`code`	A string representing the code number.Only available for DOM errors, where the number is the DOM error code.
`data`	An array of strings containing the data thrown with the error.
`message`	The Error message string.
`name`	The error code string.
`retryable`	A boolean indicating if the error is retryable.
`stack`	The JavaScript stack. If the error is thrown from XQuery, the stack contains the concatenated stack from both XQuery and JavaScript.
`stackFrame`	An array of stack frames. See the stackFrame table below for details. For details, see “JavaScript stackFrame Properties” on page 27.
`toString()`	A formatted error message populated with data.

2.7.2 JavaScript stackFrame Properties

The following is the API available to each `stackFrame`:

Properties	Description
`line`	The line number of the current frame.
`column`	The column number starting the current frame.
`operation`	The function name or operation of the current frame.
`uri`	The name of the resource that contains the script for the function/operation of this frame, or if the script name is undefined and its source ends with `//# sourceURL=…` string or deprecated `//@ sourceURL=…` string..
`language`	The query language of the current frame.

Properties	Description
`isEval`	Was the associated function compiled from a call to `eval`. (JavaScript only)
`variables`	An array of (name, value) objects containing the variable bindings in a frame. Undefined if no variable bindings are available. (XQuery only)
`contextItem`	Context item in the frame. Undefined if no context item is available. (XQuery only)
`contextPosition`	Context position in the frame. Undefined if no context item is available. (XQuery only)

2.7.3 JavaScript try/catch Example

The following is a simple JavaScript `try/catch` example:

```
try{ xdmp.documentInsert("/foo.json", {"foo": "bar"} ); }
catch (err) { err.toString(); }

=> catches the following error
( because it is missing the declareUpdate() call )
XDMP-UPDATEFUNCTIONFROMQUERY: xdmp:eval("// query&#10;
  try{ xdmp.documentInsert(&quot;/foo.json&quot;, {&q...", ())
  -- Cannot apply an update function from a query
```

2.8 JavaScript console Object

MarkLogic implements a `console` object, which contains functions to do things that log output to ErrorLog.txt in the MarkLogic data directory. The following are the `console` functions:

- `console.assert`
- `console.debug`
- `console.dir`
- `console.error`
- `console.log`
- `console.trace`
- `console.warn`

2.9 JavaScript Duration and Date Arithmetic and Comparison Methods

XQuery has operators that allow you to perform date math on duration typed data to do things like subtract durations to return dateTime values. In Server-Side JavaScript, you can get data returned in the various dateTime duration types and use the duration methods to add, subtract, multiply, , divide and compare those durations. This section describes these duration arithmetic and comparison methods and includes the following parts:

- Arithmetic Methods on Durations
- Arithmetic Methods on Duration, Dates, and Times
- Comparison Methods on Duration, Date, and Time Values

2.9.1 Arithmetic Methods on Durations

Arithmetic methods are available on the following duration objects:

- xs.yearMonthDuration Methods
- xs.dayTimeDuration Methods

2.9.1.1 xs.yearMonthDuration Methods

The JavaScript object that is an instance of `xs.yearMonthDuration` is analogous to and has the same lexical representation to the XQuery `xs:yearMonthDuration` type, as described http://www.w3.org/TR/xpath-functions/#dt-yearMonthDuration. The following methods are available on `xs.yearMonthDuration` objects:

Method	Description
`add(xs.yearMonthDuration)`	Adds two `xs.yearMonthDuration` values. Returns an `xs.yearMonthDuration`.
`subtract(xs.yearMonthDuration)`	Subtracts one `xs.yearMonthDuration` value from another. Returns an `xs.yearMonthDuration`.
`multiply(Number)`	Multiplies one `xs.yearMonthDuration` value by a `Number`. Returns an `xs.yearMonthDuration`.
`divide(Number)`	Divides an `xs.yearMonthDuration` by a `Number`. Returns an `xs.yearMonthDuration`.
`divide(xs.yearMonthDuration)`	Divides an `xs.yearMonthDuration` by an `xs.yearMonthDuration`. Returns a `Number`.

The following are some simple examples using these methods:

```
var v1 = xs.yearMonthDuration("P3Y7M");
var v2 = xs.yearMonthDuration("P1Y4M");

var r = {
"v1 + v2" : v1.add(v2),
"v1 - v2" : v1.subtract(v2),
"v1 * 2" : v1.multiply(2),
"v1 / 2" : v1.divide(2),
"v1 / v2" : v1.divide(v2)
};
r;

/*
returns:
{"v1 + v2":"P4Y11M",
 "v1 - v2":"P2Y3M",
 "v1 * 2":"P7Y2M",
 "v1 / 2":"P1Y10M",
 "v1 / v2":2.6875}
*/
```

2.9.1.2 xs.dayTimeDuration Methods

The JavaScript object that is an instance of `xs.dayTimeDuration` is analogous to and has the same lexical representation to the XQuery `xs:dayTimeDuration` type, as described http://www.w3.org/TR/xpath-functions/#dt-dayTimeDuration. The following methods are available on `xs.dayTimeDuration` objects:

Method	Description
`add(xs.dayTimeDuration)`	Adds two `xs.dayTimeDuration` values. Returns an `xs.dayTimeDuration`.
`subtract(xs.dayTimeDuration)`	Subtracts one `xs.dayTimeDuration` value from another. Returns an `xs.dayTimeDuration`.
`multiply(Number)`	Multiplies one `xs.dayTimeDuration` value by a `Number`. Returns an `xs.dayTimeDuration`.
`divide(Number)`	Divides an `xs.dayTimeDuration` by a `Number`. Returns an `xs.dayTimeDuration`.
`divide(xs.dayTimeDuration)`	Divides an `xs.dayTimeDuration` by an `xs.dayTimeDuration`. Returns a `Number`.

The following are some simple examples using these methods:

```
var v1 = xs.dayTimeDuration("P5DT4H");
var v2 = xs.dayTimeDuration("P1DT1H");

var r = {
"v1 + v2" : v1.add(v2),
"v1 - v2" : v1.subtract(v2),
"v1 * 2" : v1.multiply(2),
"v1 / 2" : v1.divide(2),
"v1 / v2" : v1.divide(v2)
};
r;

/*
returns:
{"v1 + v2":"P6DT5H",
 "v1 - v2":"P4DT3H",
 "v1 * 2":"P10DT8H",
 "v1 / 2":"P2DT14H",
 "v1 / v2":4.96}
*/
```

2.9.2 Arithmetic Methods on Duration, Dates, and Times

Methods are available on the following duration, date, and dateTime objects:

- xs.dateTime Methods
- xs.date Methods
- xs.time Methods

2.9.2.1 xs.dateTime Methods

The following methods are available on `xs.dateTime` objects:

Method	Description
`add(xs.dayTimeDuration)`	Returns the `xs.dateTime` value representing the end of the time period by adding an `xs.dayTimeDuration` to the `xs.dateTime` value that starts the period. Returns an `xs.dateTime`.
`add(xs.yearMonthDuration)`	Returns the `xs.dateTime` value representing the end of the time period by adding an `xs.yearMonthDuration` to the `xs.dateTime` value that starts the period. Returns an `xs.dateTime`.
`subtract(xs.dateTime)`	Returns the difference between two `xs.dateTime` values as an `xs.dayTimeDuration`.
`subtract(xs.dayTimeDuration)`	Returns the `xs.dateTime` value representing the beginning of the time period by subtracting an `xs.yearMonthDuration` from the `xs.dateTime` value that ends the period. Returns an `xs.dateTime`.
`subtract(xs.dayTimeDuration)`	Returns the `xs.dateTime` value representing the beginning of the time period by subtracting an `xs.dayTimeDuration` from the `xs.dateTime` value that ends the period. Returns an `xs.dateTime`.

The following are some simple examples using these methods:

```
var v1 = xs.dateTime(xs.date('2013-08-15'), xs.time('12:30:45-05:00'))
var v2 = xs.dateTime(xs.date('2012-04-01'), xs.time('01:10:25-02:00'))
var v3 = xs.yearMonthDuration("P3Y3M")
var v4 = xs.dayTimeDuration("PT1H")

var r = {
"v1 + v3" : v1.add(v3),
"v1 + v4" : v1.add(v4),
"v1 - v2" : v1.subtract(v2),
"v1 - v3" : v1.subtract(v3),
"v1 - v4" : v1.subtract(v4)
};
r;

/*
returns:
{"v1 + v3":"2016-11-15T12:30:45-05:00",
 "v1 + v4":"2013-08-15T13:30:45-05:00",
 "v1 - v2":"P501DT14H20M20S",
 "v1 - v3":"2010-05-15T12:30:45-05:00",
 "v1 - v4":"2013-08-15T11:30:45-05:00"}
*/
```

2.9.2.2 xs.date Methods

The following methods are available on `xs.date` objects:

Method	Description
`add(xs.dayTimeDuration)`	Returns the `xs.date` value representing the end of the time period by adding an `xs.dayTimeDuration` to the `xs.date` value that starts the period. Returns an `xs.date`.
`add(xs.yearMonthDuration)`	Returns the `xs.date` value representing the end of the time period by adding an `xs.yearMonthDuration` to the `xs.date` value that starts the period. Returns an `xs.date`.
`subtract(xs.date)`	Returns the difference between two `xs.date` values as an `xs.dayTimeDuration`.
`subtract(xs.dayTimeDuration)`	Returns the `xs.date` value representing the beginning of the time period by subtracting an `xs.yearMonthDuration` from the `xs.date` value that ends the period. Returns an `xs.date`.
`subtract(xs.dayTimeDuration)`	Returns the `xs.date` value representing the beginning of the time period by subtracting an `xs.dayTimeDuration` from the `xs.date` value that ends the period. Returns an `xs.date`.

The following are some simple examples using these methods:

```
var v1 = xs.date('2013-08-15')
var v2 = xs.date('2012-04-01')
var v3 = xs.yearMonthDuration("P3Y3M")
var v4 = xs.dayTimeDuration("P1DT3H")

var r = {
"v1 + v3" : v1.add(v3),
"v1 + v4" : v1.add(v4),
"v1 - v2" : v1.subtract(v2),
"v1 - v3" : v1.subtract(v3),
"v1 - v4" : v1.subtract(v4)
};
r;

/*
returns:
{"v1 + v3":"2016-11-15",
 "v1 + v4":"2013-08-16",
 "v1 - v2":"P501D",
 "v1 - v3":"2010-05-15",
 "v1 - v4":"2013-08-13"}
*/
```

2.9.2.3 xs.time Methods

The following methods are available on `xs.time` objects:

Method	Description
`add(xs.dayTimeDuration)`	Adds the value of the hours, minutes, and seconds components of an `xs.dayTimeDuration` to an `xs.time` value. Returns an `xs.time`.
`subtract(xs.time)`	Returns the difference between two `xs.time` values as an `xs.dayTimeDuration`.
`subtract(xs.dayTimeDuration)`	Subtracts the value of the hours, minutes, and seconds components of an `xs.dayTimeDuration` from an `xs.time` value. Returns an `xs.time`.

The following are some simple examples using these methods:

```
var v1 = xs.time('12:30:45-05:00')
var v2 = xs.time('01:10:25-02:00')
var v3 = xs.dayTimeDuration("PT1H")

var r = {
"v1 + v3" : v1.add(v3),
"v1 - v2" : v1.subtract(v2),
"v1 - v3" : v1.subtract(v3)
};
r;

/*
returns:
{"v1 + v3":"13:30:45-05:00",
 "v1 - v2":"PT14H20M20S",
 "v1 - v3":"11:30:45-05:00"}
*/
```

2.9.3 Comparison Methods on Duration, Date, and Time Values

Comparison methods are available to compare values (less than, greater than, and so on) on the following duration, date, and dateTime objects:

- xs.yearMonthDuration Comparison Methods
- xs.dayTimeDuration Comparison Methods
- xs.dateTime Comparison Methods
- xs.date Comparison Methods
- xs.time Comparison Methods
- xs.gYearMonth Comparison Methods
- xs.gYear Comparison Methods
- xs.gMonthDay Comparison Methods
- xs.gMonth Comparison Methods
- xs.gDay Comparison Methods

2.9.3.1 xs.yearMonthDuration Comparison Methods

The following comparison methods are available on `xs.yearMonthDuration` objects:

Method	Description
`lt(xs.yearMonthDuration)`	Less than comparison on `xs.yearMonthDuration` values. Returns a `Boolean`.
`le(xs.yearMonthDuration)`	Less than or equal to comparison on `xs.yearMonthDuration` values. Returns a `Boolean`.
`gt(xs.yearMonthDuration)`	Greater than comparison on `xs.yearMonthDuration` values. Returns a `Boolean`.
`ge(xs.yearMonthDuration)`	Greater than or equal to comparison on `xs.yearMonthDuration` values. Returns a `Boolean`.
`eq(xs.yearMonthDuration)`	Equality comparison on `xs.yearMonthDuration` values. Returns a `Boolean`.
`ne(xs.yearMonthDuration)`	Not equal to comparison on `xs.yearMonthDuration` values. Returns a `Boolean`.

The following are some simple examples using these methods:

```
var v1 = xs.yearMonthDuration("P3Y7M");
var v2 = xs.yearMonthDuration("P1Y4M");

var r = {
"v1 lt v2" : v1.lt(v2),
"v1 le v2" : v1.le(v2),
"v1 gt v2" : v1.gt(v2),
"v1 ge v2" : v1.ge(v2),
"v1 eq v2" : v1.eq(v2),
"v1 ne v2" : v1.ne(v2)
};
r;

/*
returns:
{"v1 lt v2":false,
 "v1 le v2":false,
 "v1 gt v2":true,
 "v1 ge v2":true,
 "v1 eq v2":false,
 "v1 ne v2":true}
*/
```

2.9.3.2 xs.dayTimeDuration Comparison Methods

The following comparison methods are available on `xs.dayTimeDuration` objects:

Method	Description
`lt(xs.dayTimeDuration)`	Less than comparison on `xs.dayTimeDuration` values. Returns a `Boolean`.
`le(xs.dayTimeDuration)`	Less than or equal to comparison on `xs.dayTimeDuration` values. Returns a `Boolean`.
`gt(xs.dayTimeDuration)`	Greater than comparison on `xs.dayTimeDuration` values. Returns a `Boolean`.
`ge(xs.dayTimeDuration)`	Greater than or equal to comparison on `xs.dayTimeDuration` values. Returns a `Boolean`.
`eq(xs.dayTimeDuration)`	Equality comparison on `xs.dayTimeDuration` values. Returns a `Boolean`.
`ne(xs.dayTimeDuration)`	Not equal to comparison on `xs.dayTimeDuration` values. Returns a `Boolean`.

The following are some simple examples using these methods:

```
var v1 = xs.dayTimeDuration("P5DT4H");
var v2 = xs.dayTimeDuration("P1DT1H");

var r = {
"v1 lt v2" : v1.lt(v2),
"v1 le v2" : v1.le(v2),
"v1 gt v2" : v1.gt(v2),
"v1 ge v2" : v1.ge(v2),
"v1 eq v2" : v1.eq(v2),
"v1 ne v2" : v1.ne(v2)
};
r;

/*
returns:
{"v1 lt v2":false,
 "v1 le v2":false,
 "v1 gt v2":true,
 "v1 ge v2":true,
 "v1 eq v2":false,
 "v1 ne v2":true}
*/
```

2.9.3.3 xs.dateTime Comparison Methods

The following comparison methods are available on `xs.dateTime` objects:

Method	Description
`lt(xs.dateTime)`	Less than comparison on `xs.dateTime` values. Returns a `Boolean`.
`le(xs.dateTime)`	Less than or equal to comparison on `xs.dateTime` values. Returns a `Boolean`.
`gt(xs.dateTime)`	Greater than comparison on `xs.dateTime` values. Returns a `Boolean`.
`ge(xs.dateTime)`	Greater than or equal to comparison on `xs.dateTime` values. Returns a `Boolean`.
`eq(xs.dateTime)`	Equality comparison on `xs.dateTime` values. Returns a `Boolean`.
`ne(xs.dateTime)`	Not equal to comparison on `xs.dateTime` values. Returns a `Boolean`.

The following are some simple examples using these methods:

```
var v1 = xs.dateTime(xs.date('2013-08-15'), xs.time('12:30:45-05:00'))
var v2 = xs.dateTime(xs.date('2012-04-01'), xs.time('01:10:25-02:00'))

var r = {
"v1 lt v2" : v1.lt(v2),
"v1 le v2" : v1.le(v2),
"v1 gt v2" : v1.gt(v2),
"v1 ge v2" : v1.ge(v2),
"v1 eq v2" : v1.eq(v2),
"v1 ne v2" : v1.ne(v2)
};
r;

/*
returns:
{"v1 lt v2":false,
 "v1 le v2":false,
 "v1 gt v2":true,
 "v1 ge v2":true,
 "v1 eq v2":false,
 "v1 ne v2":true}
*/
```

2.9.3.4 xs.date Comparison Methods

The following comparison methods are available on `xs.date` objects:

Method	Description
`lt(xs.date)`	Less than comparison on `xs.date` values. Returns a `Boolean`.
`le(xs.date)`	Less than or equal to comparison on `xs.date` values. Returns a `Boolean`.
`gt(xs.date)`	Greater than comparison on `xs.date` values. Returns a `Boolean`.
`ge(xs.date)`	Greater than or equal to comparison on `xs.date` values. Returns a `Boolean`.
`eq(xs.date)`	Equality comparison on `xs.date` values. Returns a `Boolean`.
`ne(xs.date)`	Not equal to comparison on `xs.date` values. Returns a `Boolean`.

The following are some simple examples using these methods:

```
var v1 = xs.date('2013-08-15');
var v2 = xs.date('2012-04-01');

var r = {
"v1 lt v2" : v1.lt(v2),
"v1 le v2" : v1.le(v2),
"v1 gt v2" : v1.gt(v2),
"v1 ge v2" : v1.ge(v2),
"v1 eq v2" : v1.eq(v2),
"v1 ne v2" : v1.ne(v2)
};
r;

/*
returns:
{"v1 lt v2":false,
 "v1 le v2":false,
 "v1 gt v2":true,
 "v1 ge v2":true,
 "v1 eq v2":false,
 "v1 ne v2":true}
*/
```

2.9.3.5 xs.time Comparison Methods

The following comparison methods are available on `xs.time` objects:

Method	Description
`lt(xs.time)`	Less than comparison on `xs.time` values. Returns a `Boolean`.
`le(xs.time)`	Less than or equal to comparison on `xs.time` values. Returns a `Boolean`.
`gt(xs.time)`	Greater than comparison on `xs.time` values. Returns a `Boolean`.
`ge(xs.time)`	Greater than or equal to comparison on `xs.time` values. Returns a `Boolean`.
`eq(xs.time)`	Equality comparison on `xs.time` values. Returns a `Boolean`.
`ne(xs.time)`	Not equal to comparison on `xs.time` values. Returns a `Boolean`.

The following are some simple examples using these methods:

```
var v1 = xs.time('12:30:45-05:00');
var v2 = xs.time('01:10:25-02:00');

var r = {
"v1 lt v2" : v1.lt(v2),
"v1 le v2" : v1.le(v2),
"v1 gt v2" : v1.gt(v2),
"v1 ge v2" : v1.ge(v2),
"v1 eq v2" : v1.eq(v2),
"v1 ne v2" : v1.ne(v2)
};
r;

/*
returns:
{"v1 lt v2":false,
 "v1 le v2":false,
 "v1 gt v2":true,
 "v1 ge v2":true,
 "v1 eq v2":false,
 "v1 ne v2":true}
*/
```

2.9.3.6 xs.gYearMonth Comparison Methods

The following comparison methods are available on `xs.gYearMonth` objects:

Method	Description
`eq(xs.gYearMonth)`	Equality comparison on `xs.gYearMonth` values. Returns a `Boolean`.
`ne(xs.gYearMonth)`	Not equal to comparison on `xs.gYearMonth` values. Returns a `Boolean`.

The following are some simple examples using these methods:

```
var v1 = xs.gYearMonth('2013-08');
var v2 = xs.gYearMonth('2012-04');

var r = {
"v1 eq v2" : v1.eq(v2),
"v1 ne v2" : v1.ne(v2)
};
r;

/*
returns:
{"v1 eq v2":false,
 "v1 ne v2":true}
*/
```

2.9.3.7 xs.gYear Comparison Methods

The following comparison methods are available on `xs.gYear` objects:

Method	Description
`eq(xs.gYear)`	Equality comparison on `xs.gYear` values. Returns a `Boolean`.
`ne(xs.gYear)`	Not equal to comparison on `xs.gYear` values. Returns a `Boolean`.

The following are some simple examples using these methods:

```
var v1 = xs.gYear('2013');
var v2 = xs.gYear('2012');

var r = {
"v1 eq v2" : v1.eq(v2),
"v1 ne v2" : v1.ne(v2)
};
r;

/*
returns:
{"v1 eq v2":false,
 "v1 ne v2":true}
*/
```

2.9.3.8 xs.gMonthDay Comparison Methods

The following comparison methods are available on `xs.gMonthDay` objects:

Method	Description
`eq(xs.xs.gMonthDay)`	Equality comparison on `xs.xs.gMonthDay` values. Returns a `Boolean`.
`ne(xs.xs.gMonthDay)`	Not equal to comparison on `xs.xs.gMonthDay` values. Returns a `Boolean`.

The following are some simple examples using these methods:

```
var v1 = xs.gMonthDay('--08-20');
var v2 = xs.gMonthDay('--04-14');

var r = {
"v1 eq v2" : v1.eq(v2),
"v1 ne v2" : v1.ne(v2)
};
r;

/*
returns:
{"v1 eq v2":false,
 "v1 ne v2":true}
*/
```

2.9.3.9 xs.gMonth Comparison Methods

The following comparison methods are available on `xs.gMonth` objects:

Method	Description
`eq(xs.gMonth)`	Equality comparison on `xs.gMonth` values. Returns a `Boolean`.
`ne(xs.gMonth)`	Not equal to comparison on `xs.gMonth` values. Returns a `Boolean`.

The following are some simple examples using these methods:

```
var v1 = xs.gMonth('--08');
var v2 = xs.gMonth('--04');

var r = {
"v1 eq v2" : v1.eq(v2),
"v1 ne v2" : v1.ne(v2)
};
r;

/*
returns:
{"v1 eq v2":false,
 "v1 ne v2":true}
*/
```

2.9.3.10 xs.gDay Comparison Methods

The following comparison methods are available on `xs.gDay` objects:

Method	Description
`eq(xs.gDay)`	Equality comparison on `xs.gDay` values. Returns a `Boolean`.
`ne(xs.gDay)`	Not equal to comparison on `xs.gDay` values. Returns a `Boolean`.

The following are some simple examples using these methods:

```
var v1 = xs.gDay('---08');
var v2 = xs.gDay('---04');

var r = {
"v1 eq v2" : v1.eq(v2),
"v1 ne v2" : v1.ne(v2)
};
r;

/*
returns:
{"v1 eq v2":false,
 "v1 ne v2":true}
*/
```

2.10 MarkLogic JavaScript Functions

There are a large number of MarkLogic built-in functions available in JavaScript. In general, most functions available in XQuery have siblings that are available in JavaScript. For details on the MarkLogic functions available in JavaScript, see “JavaScript Functions and Constructors” on page 47.

3.0 JavaScript Functions and Constructors

This chapter describes how to use the MarkLogic built-in functions, and describes how to import and use XQuery libraries in your JavaScript program. It includes the following sections:

- Built-In JavaScript Functions
- Functions That are part of the Global Object
- Using XQuery Functions and Variables in JavaScript
- Importing JavaScript Modules Into JavaScript Programs
- Other MarkLogic Objects Available in JavaScript
- Amps and the module.amp Function
- JavaScript Type Constructors

3.1 Built-In JavaScript Functions

MarkLogic contains many built-in functions that offer fast and convenient programmatic access to MarkLogic functionality. The built-in functions are available as JavaScript functions without the need to import or require any libraries (that is why they are called “built-in”). You can find the functions in the Server-Side JavaScript API Documentation.

The functions are available via the following global objects:

- `cts.`
- `fn.`
- `math.`
- `rdf.`
- `sc.`
- `sem.`
- `spell.`
- `sql.`
- `xdmp.`

for example, to get the current time, you can call the following:

```
fn.currentDateTime();
```

3.2 Functions That are part of the Global Object

There are MarkLogic-specific functions that are part of the global JavaScript object (without a namespace prefix). This section calls out the following global functions:

- declareUpdate Function
- require Function

3.2.1 declareUpdate Function

In order to perform an update to a document, you must declare the transaction as an update; if `declareUpdate` is not called at the beginning of a statement, the statement is run as a query. The following is the syntax of the `declareUpdate` function (see `Global-Object.declareUpdate`):

```
declareUpdate(Object options)
```

where `options` is an optional argument as follows:

```
{explicitCommit: true/false}
```

If the `options` argument is omitted or `explicitCommit` property is set to `false`, the transaction is explicitly committed. If the `explicitCommit` property is set to `true`, then it starts a multi-statement transaction and requires an explicit `xdmp.commit` or `xdmp.rollback` to complete the transaction.

For details on transactions, see Understanding Transactions in MarkLogic Server in the *Application Developer's Guide*.

The following is an example of an update transaction in JavaScript:

```
declareUpdate();
var myDoc = {"foo":"bar"};
xdmp.documentInsert("/myDoc.json", myDoc);
// creates the /myDoc.json document
```

The following runs as a multi-statement transaction (although this transaction only has a single statement):

```
declareUpdate({explicitCommit: true});
var myDoc = {"foo":"bar"};
xdmp.documentInsert("/myDoc.json", myDoc);
xdmp.commit();
// creates the /myDoc.json document
```

3.2.2 require Function

The `require` function (see `Global-Object.require`) is available in the global object, and it allows you to import a library into your JavaScript program. For details, see "require Function" on page 49.

3.3 Using XQuery Functions and Variables in JavaScript

You can import XQuery library modules into a Server-Side JavaScript program and then call those functions and/or variables from JavaScript. Importing XQuery libraries is useful if you have existing XQuery code that you want to use in your JavaScript programs, and it is also useful if you want to perform a task that is well-suited to XQuery from your JavaScript Program. This section describes how to use XQuery modules with your JavaScript programs and includes the following parts:

- require Function
- Importing XQuery Modules to JavaScript Programs

3.3.1 require Function

You can import an XQuery or JavaScript library by using the following JavaScript function:

```
require(String location)
```

where `location` is the path to the JavaScript or XQuery file. The extension of the path can be omitted for simplicity. The path obeys the same rules for XQuery defined in Rules for Resolving Import, Invoke, and Spawn Paths in the *Application Developer's Guide*.

Typically, the `require` function is the first line of the JavaScript program, and a program can have 0 or more `require` functions. When importing an XQuery library, a common practice is to name your JavaScript variable as you would name your namespace prefix. For example, if you are importing the Search API library, your require statement might look as follows:

```
var search = require("/MarkLogic/appservices/search/search.xqy");
search.search("hello");
// returns a search response for documents matching "hello"
```

3.3.2 Importing XQuery Modules to JavaScript Programs

MarkLogic has a rich set of XQuery library modules to make it easy to write programs to do a variety of things, such as building a search application, building an alerting application, adding spelling correction to your application, and so on. You might have created your own rich sets of XQuery libraries. There might be something (such as an XPath statement) that is convenient to write in XQuery but might be less convenient to write in JavaScript.

You can make use of these XQuery libraries in MarkLogic Server-Side JavaScript programs by using the `require` function. This section describes the mapping of names and types from an XQuery environment to a JavaScript environment and includes the following parts:

- Mapping Between XQuery Function and Variable Names to JavaScript
- Type Mapping Between XQuery and JavaScript

3.3.2.1 Mapping Between XQuery Function and Variable Names to JavaScript

In XQuery, it is common to create function and variable names with hyphens (-) in them; in JavaScript, a hyphen (-) is a subtraction operator, so the names are not compatible. In JavaScript, camelCase is a common way to name functions and variables. To deal with these differences between the languages, any XQuery function or variable imported to a JavaScript program with the `require` function is accessible according to the following rules:

- Namespace prefixes, which in XQuery are followed by a colon (:) and then the function localname, are denoted by the namespace prefix followed by a period (.), like any object notation.
- Function or variable names that have hyphen characters (-) are converted to camelCase names. For example, a function in XQuery named `my-function` is available to JavaScript with the name `myFunction`.
- For cases where the above rules might cause some ambiguity (these cases are rare), you can also access a function by its bracket notation, using the literal names from the XQuery function or variable. For example, an XQuery function names `hello:my-world` (that is, a function bound to the `hello` prefix with the localname `my-world`) can be accessed with the following JavaScript notation: `hello["my-world"]()`.

You can use these rules to access any public XQuery function or variable from your JavaScript program.

3.3.2.2 Type Mapping Between XQuery and JavaScript

JavaScript has looser typing rules that XQuery, and also has fewer types than XQuery. MarkLogic automatically maps types from XQuery to JavaScript. The following table shows how XQuery types are mapped to JavaScript types.

XQuery Type	JavaScript Type	Notes
`xs:boolean`	`Boolean`	
`xs:integer`	`Integer`	
`xs:double`	`Number`	
`xs:float`	`Number`	
`xs:decimal`	`Number`	If the value us greater than 9007199254740992 or the scale is less than 0, then the value is a String.
`json:array`	`Array`	
`json:object`	`Object`	

XQuery Type	JavaScript Type	Notes
map:map	Object	
xs:date	Date	Any extra precision is preserved.
xs:dateTime	Date	Any extra precision is preserved.
xs:time	String	
empty-sequence()	null	
item()	String	
xs:anyURI	String	
node()	Node	
node()*	ValueIterator	

3.4 Importing JavaScript Modules Into JavaScript Programs

You can use the `require` function to import a Server-Side JavaScript library into a Server-Side JavaScript program. When you import a JavaScript library using the `require` function, all of the functions and global variables in the JavaScript library are available via the `exports` object, which is returned by the `require` function. For example:

```
var circle = require("circle.js");
circle.area(4);
// evaluates the area function from circle.js,
// passing 4 as its parameter
```

You can import JavaScript libraries with either the `.js` or `.sjs` extension (with corresponding mimetypes `application/javascript` and `application/vnd.marklogic-javascript`). You cannot, however, serve up directly from the App Server a Server-Side JavaScript module with a `.js` file extension; directly served modules need a `.sjs` extension. For more details about the require function, see “require Function” on page 49.

3.5 Other MarkLogic Objects Available in JavaScript

There are a number of MarkLogic objects available in JavaScript to make it easier to work with nodes and documents in JavaScript. For details on these objects, see “MarkLogic JavaScript Object API” on page 8.

3.6 Amps and the module.amp Function

You can create amped functions in JavaScript. An amped function is a function that evaluates with amplified privileges based on the role to which an amp is configured. Amps require a function that is in the Modules database or under the *<marklogic-dir>/Modules* directory, as well as a piece of configuration (the amp) in the security database. For details on amps, see Temporarily Increasing Privileges with Amps in the *Understanding and Using Security Guide*. This section describes JavaScript amps and includes the following parts:

- module.amp Function
- Simple JavaScript Amp Example

3.6.1 module.amp Function

The `module.amp` function has the following signature:

```
module.amp(Function namedFunction)
```

It must be used in an exports statement that is in a JavaScript module that is in the Modules database or is under the *<marklogic-dir>/Modules* directory. A sample exports statement is as follows:

```
exports.ampedFunctionName = module.amp(ampedFunctionName);
```

where `ampedFunctionName` is the name of the function in your library to be amped.

3.6.2 Simple JavaScript Amp Example

The following creates a JavaScript module for an amp, creates an amp to reference the module, and then calls the function from another module. The result is that the function can be run by an unprivileged user, even though the function requires privileges.

1. Create the amp module as a file in your Modules database or under the *<marklogic-dir>/Modules* directory. For example, on a UNIX system, create the following file as `/opt/MarkLogic/test-amp.sjs` (you will need to make sure the file is readable by MarkLogic):

```
// This is a simple amp module
// It requires creating an amp to the URI of this sjs file with the
// function name.

function ampedInsert() {
xdmp.documentInsert("/amped.json", {prop:"this was produced by an \n\
  amped function"}, [xdmp.permission("qconsole-user", "read"),
                     xdmp.permission("qconsole-user", "update")]);
};

exports.ampedInsert = module.amp(ampedInsert);
```

2. Create the amp that points to this function. For example, in the Admin Interface, go to Security > Amps and select the Create tab. Then enter the name of the amp function under localname (`ampedInsert`), leave the namespace blank, enter the path to the JavaScript module (for example, `/test-amp.sjs`), select filesystem for the database, and finally assign it a role to which the function amps. For this example, sect the `admin` role.

3. Now, from an App Server root, create a JavaScript module with the following contents:

```
declareUpdate();
var mod = require("/test-amp.sjs");
mod.ampedInsert();
```

4. As an unprivileged user, run the program created above. For example, if the program was saved as `/space/appserver/test.sjs`, and your App Server root on port 8005 is `/space/appserver`, then access `http://localhost:8005/test.sjs`.

   You can create an unprivileged user in the Admin Interface by creating a user without giving it any roles.

You can then go to Query Console and see that it created the document called `/amped.json`.

This example is simplified from a real-world example in two ways. First, it places the amped module under the Modules directory. The best practice is to use a modules database to store your amped function. Note that when using a modules database, you need to insert the module into that database with the needed permissions on the document. Second, the example amps to the `admin` role. In a real-world example, it is best practice to create a role that has the minimal privileges needed to perform the functions that users with the role require.

3.7 JavaScript Type Constructors

There are MarkLogic-specific constructors added to the JavaScript environment to allow you to construct XQuery types in JavaScript. The constructors have the same names as their XQuery counterparts, but with a dot (.) instead of a colon(:) to separate the namespace from the constructor name. For constructors that have a minus sign (-) in them, you will have to put square brackets around the localname to call it (for example, `cts['complex-polygon']`).

To use each constructor, pass a constructible object into the constructor. The following example shows how to use the `xs.QName` constructor:

```
fn.namespaceUriFromQName(xs.QName("xdmp:foo"))
=> http://marklogic.com/xdmp
   (because the xdmp namespace is always in scope)
```

The following is a list of MarkLogic constructors that you can call from JavaScript.

```
xs.simpleDerivationSet
xs.gYear
xs.public
```

```
xs.language
xs.short
xs.decimal
xs.reducedDerivationControl
xs.gYearMonth
xs.date
xs.double
xs.nonPositiveInteger
xs.positiveInteger
xs.blockSet
xs.normalizedString
xs.namespaceList
xs.gMonth
xs.integer
xs.int
xs.anyAtomicType
xs.gMonthDay
xs.NCName
xs.unsignedShort
xs.derivationControl
xs.IDREFS
xs.derivationSet
xs.token
xs.ID
xs.nonNegativeInteger
xs.anyURI
xs.NMTOKEN
xs.allNNI
xs.QName
xs.base64Binary
xs.boolean
xs.long
xs.Name
xs.yearMonthDuration
xs.duration
xs.NMTOKENS
xs.dayTimeDuration
xs.negativeInteger
xs.NOTATION
xs.unsignedInt
xs.unsignedLong
xs.untypedAtomic
xs.formChoice
xs.dateTime
xs.float
xs.ENTITY
xs.byte
xs.time
xs.unsignedByte
xs.ENTITIES
xs.string
xs.IDREF
xs.hexBinary
xs.gDay
```

```
cts.andNotQuery
cts.andQuery
cts.boostQuery
cts.box
cts.circle
cts.collectionQuery
cts.collectionReference
cts.complexPolygon
cts.confidenceOrder
cts.directoryQuery
cts.documentFragmentQuery
cts.documentOrder
cts.documentQuery
cts.elementAttributePairGeospatialQuery
cts.elementAttributeRangeQuery
cts.elementAttributeReference
cts.elementAttributeValueQuery
cts.elementAttributeWordQuery
cts.elementChildGeospatialQuery
cts.elementGeospatialQuery
cts.elementPairGeospatialQuery
cts.elementQuery
cts.elementRangeQuery
cts.elementReference
cts.elementValueQuery
cts.elementWordQuery
cts.falseQuery
cts.fieldRangeQuery
cts.fieldReference
cts.fieldValueQuery
cts.fieldWordQuery
cts.fitnessOrder
cts.geospatialElementAttributePairReference
cts.geospatialElementChildReference
cts.geospatialElementPairReference
cts.geospatialElementReference
cts.geospatialJsonPropertyChildReference
cts.geospatialJsonPropertyPairReference
cts.geospatialJsonPropertyReference
cts.geospatialPathReference
cts.indexOrder
cts.jsonPropertyChildGeospatialQuery
cts.jsonPropertyGeospatialQuery
cts.jsonPropertyPairGeospatialQuery
cts.jsonPropertyRangeQuery
cts.jsonPropertyReference
cts.jsonPropertyScopeQuery
cts.jsonPropertyValueQuery
cts.jsonPropertyWordQuery
cts.linestring
cts.locksFragmentQuery
cts.longLatPoint
cts.lsqtQuery
cts.nearQuery
```

```
cts.notInQuery
cts.notQuery
cts.order
cts.orQuery
cts.pathGeospatialQuery
cts.pathRangeQuery
cts.pathReference
cts.period
cts.periodCompareQuery
cts.periodRangeQuery
cts.point
cts.polygon
cts.propertiesFragmentQuery
cts.punctuation
cts.qualityOrder
cts.query
cts.reference
cts.region
cts.registeredQuery
cts.reverseQuery
cts.scoreOrder
cts.searchOption
cts.similarQuery
cts.space
cts.special
cts.termQuery
cts.token
cts.tripleRangeQuery
cts.trueQuery
cts.unordered
cts.uriReference
cts.word
cts.wordQuery
dir.type
math.coefficients
sem.iri
sem.variable
sem.blank
```

4.0 Technical Support

MarkLogic provides technical support according to the terms detailed in your Software License Agreement or End User License Agreement.

We invite you to visit our support website at http://help.marklogic.com to access information on known and fixed issues, knowledge base articles, and more. For licensed customers with an active maintenance contract, see the Support Handbook for instructions on registering support contacts and on working with the MarkLogic Technical Support team.

Complete product documentation, the latest product release downloads, and other useful information is available for all developers at http://developer.marklogic.com. For general questions, join the general discussion mailing list, open to all MarkLogic developers.

5.0 Copyright

MarkLogic Server 8.0 and supporting products.

NOTICE

Copyright © 2018 MarkLogic Corporation.

This technology is protected by one or more U.S. Patents 7,127,469, 7,171,404, 7,756,858, 7,962,474, 8,935,267, 8,892,599 and 9,092,507.

All MarkLogic software products are protected by United States and international copyright, patent and other intellectual property laws, and incorporate certain third party libraries and components which are subject to the attributions, terms, conditions and disclaimers found at http://docs.marklogic.com/guide/copyright/legal.

MarkLogic and the MarkLogic logo are trademarks or registered trademarks of MarkLogic Corporation in the United States and other countries. All other trademarks are property of their respective owners.

For all copyright notices, including third-party copyright notices, see the Combined Product Notices for your version of MarkLogic.