MarkLogic Server

JavaScript Reference Guide

MarkLogic 10
May, 2019

Last Revised: 10.0, May, 2019

Table of Contents

JavaScript Reference Guide

1.0 Server-Side JavaScript in MarkLogic

MarkLogic 10 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:

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.

MarkLogic embeds version 6.7 of the Google V8 JavaScript engine.

This version of V8 offers some of the newer EcmaScript 2015 (formerly known as EcmaScript 6) features. Some EcmaScript 15 features are:

- Arrow Function
- Spread Operator and rest Parameters
- Maps and Sets
- Classes
- Constants and Block-Scoped Variables
- Template Strings
- Symbols

EcmaScript 2015 generators use the `function*` syntax. For a description of EcmaScript 6 generators, see documentation for implementation 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 (let i = 0; i < limit; i++)
    yield xdmp.eval('xs.dateTime(new Date())');}
const result=[];
for (const 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 programmers.

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:

```
const results = new Array();
const cdt = fn.currentDateTime();
results.push(cdt);
const 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:

```
const 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 54.

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.

The module being invoked using `xdmp.invoke` may either be JavaScript or XQuery. The module is considered to be JavaScript if the module path ends with a file extension configured for the MIME type `application/vnd.marklogic-javascript` or `application/vnd.marklogic-js-module` in MarkLogic's Mimetypes configuration. Otherwise, it is assumed to be XQuery.

Invoking a function is programming-language specific. The XQuery version of `xdmp:invoke-function` can only be used to invoke XQuery functions. The Server-Side JavaScript version of this function (`xdmp.invokeFunction`) can only be used to invoke JavaScript functions.

The next example invokes a module using external variables, and executes in a separate transaction.

Assume you have a module in the modules database with a URI "http://example.com/application/log.sjs" containing the following code:

```
xdmp.log(myvar)
```

Then you can call this module using xdmp.invoke as follows:

```
xdmp.invoke("log.sjs",
{myvar: "log this"},
        {
          modules : xdmp.modulesDatabase(),
          root : "http://example.com/application/",
          isolation : "different-transaction"
});


  => Invokes a JavaScript module from the modules database
     with the URI http://example.com/application/log.sjs.
     The invoked module will then be executed, logging the
     message sent in the external variable to the log.
```

1.9 Each App Server Thread Runs a V8 Engine Instance

Each App Server thread runs its own isolate of the V8 JavaScript engine. Objects from one isolate cannot be used in another. This means that V8 data structures such as function objects cannot be shared across App Server threads.

For example, if you cache a function object in a server field in one thread, and then try to access it from another thread (such as from code executed under `xdmp.spawn`), the function will not be valid. A Server-Side JavaScript function is only valid in the thread in which it is created.

1.10 Exception Handling

If you are accustomed to working with XQuery or you are developing in both XQuery and Server-Side JavaScript, you should be aware that the semantics of exception handling are not the same in the two languages.

MarkLogic implements the standard exception handling semantics for JavaScript: JavaScript statements in a try block are not rolled back if they complete before a caught exception is raised. In XQuery, all expressions evaluated in a try block are rolled back, even if the exception is caught.

For example, in the following code, the call to `xdmp.documentSetMetadata` data throws an `XDMP-CONFLICTINGUPDATES` exception because it tries to update the document metadata twice in the same transaction. The exception is trapped by the try-catch. The initial document insert succeeds because it was evaluated before the exception occurs.

```
'use strict';
declareUpdate();

try{
  xdmp.documentInsert("doc.json",
                      {content :"value"},
                      {metadata:{a:1, b:2}})
  xdmp.documentSetMetadata("doc.json", {c:3})
} catch(err) {
  err.toString();
}
```

The equivalent XQuery code would not insert "doc.json". For more details, see try/catch Expression in the *XQuery and XSLT Reference Guide*.

1.11 Interaction with XQuery

You can call into Server-Side JavaScript code from XQuery, and vice versa.

For example, you can use a library module such as the XQuery triggers library (`trgr`) from Server-Side JavaScript, whether or not the documentation explicitly calls it out. For details, see “Using XQuery Functions and Variables in JavaScript” on page 52.

You can also eval or invoke code blocks in either language. Use `xdmp.xqueryEval` to evaluate a block of XQuery from Server-Side JavaScript. Use `xdmp.invoke` to invoke either XQuery or Server-Side JavaScript from Server-Side JavaScript.

Similarly, you can use `xdmp:javascript-eval` to evaluate Server-Side JavaScript from XQuery, and `xdmp:invoke` to invoke either XQuery or Server-Side JavaScript from XQuery.

2.0 MarkLogic JavaScript Object API

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

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:

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.	
`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"`. Note that `xpath` returns a Sequence if the expression matches more than one node.	

For additional DOM properties available on XML nodes (document, element, attribute, processing instruction, and comment), see “Node Object for XML Nodes” on page 14.

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 node:
const 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:

2.2.1 Node Object for XML Nodes

In addition to the `Node` properties described in "Node Object" on page 12, 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 local name 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.

Properties	Description
`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.
`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 local name, 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:
	`const x = new NodeBuilder();`
	`const 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:
	`const x = new NodeBuilder();`
	`const 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:

```
const x = new NodeBuilder();
x.startDocument();
x.startElement("foo", "bar");
x.addText("text in bar");
x.endElement();
x.endDocument();
const 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 local name.

`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 local name, 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 local name.
`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 local name.

`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

`Value` is an object class that wraps MarkLogic XQuery types, enabling you to pass these objects into functions that are designed for those types.

`Value` supports `valueOf` and `toObject` methods for converting the underlying value to its closest native JavaScript value or object. For more details, see Value in the *MarkLogic Server-Side JavaScript Function Reference*.

Any builtin function whose signature indicates an XML atomic return type such as `xs.date`, `xs.int`, or `xs.string` returns a `Value` object. A function whose signature indicates a native JavaScript type such as number, boolean, or string returns a simple, native value.

For example, the `fn.currentDate` builtin function returns a `Value` representing an `xs:date`, and its return type in the *MarkLogic Server-Side JavaScript Function Reference* is `xs.date`. It returns a Value object that contains an XQuery `xs:date` value. This enables you to pass the result to date-specific functions such as `xdmp.weekFromDate`.

Similarly, lexicon functions such as `cts.values`, `cts.words`, and `cts.geospatialBoxes` return a `Sequence` of `Value` objects rather than native JavaScript objects in order to preserve type and support passing the `Sequence` items to `cts.frequency`.

See the following topics for more details:

- Example: xs:date as Value
- Comparison to Native JavaScript Values
- Example: Comparison between a Value and a Number

2.3.1 Example: xs:date as Value

JavaScript has no native equivalent to `xs:date`. Such values can only be represented natively as a string, which loses the "dateness" of the value. The string has to be parsed back into a date before you can use it as input to a function that expects an `xs:date`:

```
xdmp.daynameFromDate(xs.date('1997-07-20'));
```

A DateTime function such as `fn.currentDate` returns a `Value` object representing an XQuery `xs:date` value. The following test returns true:

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

The `Value` returned by `fn.currentDate` can be passed directly into functions that expect an `xs.date`, such as `xdmp.daynameFromDate` without conversion from a string:

```
xdmp.daynameFromDate(fn.currentDate());
```

If you probe the value returned by `fn.currentDate`, you can see it is not a native JavaScript type:

```
typeof fn.currentDate();                            // object
Object.prototype.toString.call(fn.currentDate());  // [object xs.date]
```

For more details about dates, see Dates in Server-Side JavaScript.

2.3.2 Comparison to Native JavaScript Values

You can use a `Value` like a native JavaScript value in contexts in which loose equality is sufficient. For example, when comparing a `Value` to a number using the "==" operator:

```
someDecimalValue == 1
```

You cannot successfully compare a `Value` to a native JavaScript value in contexts where strict equality or "same value" equality is used, such as the "===" operator, `Array.prototype.indexOf`, or `Array.prototype.includes`.

For more details, see "Example: Comparison between a Value and a Number" on page 26.

2.3.3 Example: Comparison between a Value and a Number

Suppose you call `cts.values` on a lexicon over `xs:int` values. The return value will be a Sequence containing Value objects that represent integer values. Supposed the first item in the returned `Sequence` contains a `Value` object equivalent to the number 10. Then following expressions evaluate to the results shown:

```
const mlValues = cts.values(cts.pathReference('/my/int/property'));

fn.head(mlValues) == 10;              // true
fn.head(mlValues) === 10;             // false
fn.head(mlValues).valueOf() === 10;   // true
mlValues.toArray().includes(10);      // false
mlValues.toArray().indexOf(10);       // -1 (no match)
fn.head(mlValues) instanceof Value;   // true
typeof fn.head(mlValues);             // 'object'
typeof fn.head(mlValues).valueOf();   // 'number'
```

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 Sequence

A `Sequence` is a JavaScript `Iterable` object that represents a set of values. Many MarkLogic functions that return multiple values do so in Server-Side JavaScript by returning a `Sequence`. An iterable is a JavaScript object which returns an iterator object from the `@@iterator` method.

You can iterate over the values in a `Sequence` using a `for..of loop`. For example:

```
for (const doc of fn.collection('/my/coll')) {
  // do something with doc
}
```

If you want to extract just the first (or only) item in a `Sequence` without iteration, use `fn.head`. For example, the `xdmp.unquote` function returns a `Sequence`, but in many cases it is a `Sequence` containing only one item, so you could extract that single result with code like the following:

```
const node = fn.head(xdmp.unquote('<data>some xml</data>'))
```

You can create your own `Sequence` object from an array, an array-like, or another iterable using `Sequence.from`. For example, the following code snippet creates a `Sequence` from an array:

```
const mySeq = Sequence.from([1,2,3,4]);
```

Use `fn.count` to count the number of items in a `Sequence`. For example:

```
const mySeq = Sequence.from([1,2,3,4]);
fn.count(mySeq);       // returns 4
```

For more details, see *Sequence Object* in the *MarkLogic Server-Side JavaScript Function Reference*.

2.6 ValueIterator

Note: This interface is deprecated. As of MarkLogic 9, no MarkLogic functions return a `ValueIterator` or accept a `ValueIterator` as input. Use the guidelines in this section to transition your code to Sequence.

Code that manipulates `ValueIterator` results as described in the following table will continue to work transparently when the return type is `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	`const uris = cts.uris('/');`
	`for (const u of uris) {`
	`  // do something with u`
	`}`
		`const 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` cannot used with a `Sequence` value.

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.

```
const results = cts.uris('/', ['limit=10']);
const 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 (const val of clone) {
  // do something with val
}
```

2.7 JavaScript instanceof Operator

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

```
const a = Sequence.from(["saab", "alfa romeo", "tesla"]);
a instanceof Sequence;
// returns true
```

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

An `xs.date` object type:

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

Not an `xs.date` object type:

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

A JavaScript `Date` object type:

```
const 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`
- `Sequence`
- `Text`
- `Element`
- `ProcessingInstruction`
- `XMLDocument`
- `ValueIterator`

The following is an example using an XML document:

```
// assume "/one.xml" has content <name>value</name>
const a = fn.head(fn.doc("/one.xml")).root;
const 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."}
const res = new Array();
const a = fn.head(fn.doc("/car.json"));
res.push(a instanceof Document);
const 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.

2.8 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:

2.8.1 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 31.
`toString()`	A formatted error message populated with data.

2.8.2 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.
`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)

2.8.3 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
```

2.9 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`

2.10 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

2.10.1 Arithmetic Methods on Durations

Arithmetic methods are available on the following duration objects:

- xs.yearMonthDuration Methods
- xs.dayTimeDuration Methods

2.10.1.1 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:

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

const 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}
*/
```

2.10.1.2 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:

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

const 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}
*/
```

2.10.2 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

2.10.2.1 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:

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

const 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"}
*/
```

2.10.2.2 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:

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

const 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"}
*/
```

2.10.2.3 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:

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

const 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"}
*/
```

2.10.3 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

2.10.3.1 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:

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

const 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}
*/
```

2.10.3.2 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:

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

const 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}
*/
```

2.10.3.3 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:

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

const 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}
*/
```

2.10.3.4 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:

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

const 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}
*/
```

2.10.3.5 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:

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

const 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}
*/
```

2.10.3.6 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:

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

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

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

2.10.3.7 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:

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

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

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

2.10.3.8 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:

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

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

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

2.10.3.9 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:

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

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

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

2.10.3.10xs.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:

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

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

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

2.11 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 50.

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 automatically 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();
const 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});
const 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 52.

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:

```
const 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 local name, 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 local name `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:

```
const 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 52.

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 11.

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 *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.

Use the import.meta.amp() function to amp with ES6 modules. Use the style shown in the following sample statement:

```
export function fileExists(filename) {
xdmp.filesystemFileExists(filename);
}
export const fileExistsAmped = import.meta.amp(fileExists);
```

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 local name (`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();
const 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 local name 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 Converting JavaScript Scripts to Modules

Evaluating a JavaScript program may generate side-effects on the JavaScript global environment; therefore, each JavaScript program is evaluated in a separate v8 context. The overhead of creating such a context is significant, and in the recent v8 version that overhead has increased by roughly 40%.

To compensate for this overhead, it is suggested that you convert your JavaScript scripts to JavaScript modules.

4.1 Benefits of JavaScript Modules

In addition to the performance improvements, there are some other side benefits to using JavaScript modules, such as:

- Modules may be executed any number of times, but are loaded only once, thus improving performance.
- Module scripts may be shared by multiple applications.
- Modules help identify and remove naming conflicts. The content page will not load if there are naming clashes across different modules. This helps identify conflicts early in your development cycle.
- If anything changes in the module dependency chain, the issue is identified quickly during module parsing.
- Your code may be created as a set of small, maintainable files, which will help with very large projects.
- For convenience, you may then bring together under all those small modules under the scope of a single module.

4.2 Other differences between JavaScript Scripts and Modules

In addition to the performance improvements, there are some other side benefits to using JavaScript modules, such as:

- Modules are always executed in strict mode, regardless of whether strict mode is declared.
- Modules may both import and export.
- Just about any object may be exported, including class, function, let, var or const and any top-level function.
- Module code is (with the exception of `export`) regular JavaScript code and can use any object or other functionality available to any other JavaScript program.

- By default, everything declared in an ES6 module is private, and runs in strict mode. Only functions, classes, variables and constants that are exposed using `export` are public.
- Module objects are frozen and there is no way to modify them once they are loaded.
- All import and export declarations must be made at the top level. So you cannot declare an export inside a function, or as part of a conditional clause. You also cannot export items programmatically by iterating through an array or on demand.
- There is no way to recover from an import error. Program execution will stop as soon as any module in the dependency tree fails to load. All module dependencies are loaded eagerly, and there is no programmatic way to load a module on demand.

4.3 Performance Considerations

Evaluating a JavaScript program may generate side-effects on the JavaScript global environment; therefore, each JavaScript program is evaluated in a separate v8 context. The overhead of creating such a context is significant, and in the recent v8 version that overhead has increased by roughly 40%.

To compensate for this overhead, it is suggested that you convert your JavaScript scripts to JavaScript modules. A JavaScript module program is one that adheres to the following:

1. A program that uses strict JavaScript syntax and can be compiled as a JavaScript module.

2. A program that contains a main module with the ".mjs" extension.

3. Any ad hoc program that uses import or export syntax.

For further reading on the details of converting script programs into module programs, please see the chapter on modules in the ECMASCRIPT Language Specification.

4.4 Creating and Using ES6 Modules

Creating an ES6 module may be accomplished by adding an `export` statement to any JavaScript script file: This will make the objects being exported available for all scripts to import.

Public variables, functions and classes are exposed using an `export` statement. By default, all declared objects within an ES6 module are private. The module runs in strict mode with no need for the `use strict` declaration. Here is a simple example:

```
// myMathLib.mjs

export const PI = 3.14159265359;

export const E = 2.718281828459;

export const GAMMA = 0.577215;

export const reducer = (accumulator, currentValue) => accumulator +
currentValue;
```

```
// The following objects are private

const G = 0.915965594177;

const x = 0.11000100000000000000001;


export function adder(arguments) {

  console.log('Grand Total: ', arguments);

  return arguments.reduce(reducer);

}
```

You may also dispense with all individual `export` statements and use a single `export` line to define them all:

```
// myMathLib.mjs

const PI = 3.14159265359;

const E = 2.718281828459;

const GAMMA = 0.577215;

const reducer = (accumulator, currentValue) => accumulator +
currentValue;


// The following objects will remain private

const G = 0.915965594177;

const x = 0.11000100000000000000001;


export function adder(arguments) {

  console.log('Grand Total: ', arguments);

  return arguments.reduce(reducer);

}


export { PI, E, GAMMA, reducer, adder, ... };
```

You then use an `import` statement to pull items from a module into another script or module:

```
// myScript.js
import { adder } from './myMathLib.mjs';

console.log( adder(1,2,3,4,5) );
=> 15
```

From the `import` statement, we know that myMathLib.mjs resides in the same directory as myScript.js. In addition to the relative path shown above, you may also use:

- Full URLs - starting with HTTPS or FILE.
- Absolute file references - starting with /.

Multiple items may be imported in the same `export` line:

```
import { adder, aglomerator } from './myMathLib.mjs';

console.log( adder(1,2,3,4,5) );  // 15
console.log( aglomerator(1,2,3,4,5) ); // 12345
```

To resolve naming collisions, imported functions may be aliased as follows:

```
import { adder as sum, aglomerator as glob } from './myMathLib.mjs';

console.log( sum(1,2,3,4,5) );      // 15
console.log( glob(1,2,3,4,5) ); // 12345
```

Lastly, you are able to import all public items by providing a namespace:

```
import * as trans from './myMathLib.mjs';


console.log( trans.PI ); // 3.14159265359

console.log( trans.E );  // 2.718281828459

console.log( trans.GAMMA ); // 0.577215
```

4.5 Dynamic Imports are not Allowed

As stated above: All import and export declarations must be made at the top level. So you cannot declare an export inside a function, or as part of a conditional clause. You also cannot export items programmatically by iterating through an array or on demand

```
import ... from someFunction(); // ERROR: may only import from "string"
```

The following statements will also not work:

```
if(some-condition) {

  import ...; // ERROR: can't import conditionally

}

{

  import ...; // ERROR: import is only allowed at the top level

}
```

The net result is that, by only allowing `import` at the top level, your code is much more easy to parse, analyze and to bundle. In the process, unused functions will be "shakes out," potentially reducing the size of the executable. This is possible because of the strict rules under which modules operate.

4.6 Using JavaScript Modules in the Browser

On the web, you can tell browsers to treat a <script> element as a module by setting the `type` attribute to `module`.

```
<script type="module" src="optic.mjs"></script>
```

```
<script nomodule src="optic.sjs"></script>
```

This works because modern browsers understand `type="module"` and will also ignore scripts with a `nomodule` attribute. This allows the programmer to serve a module-based payload to module-supporting browsers while providing a failover mode to older browsers. Only older browsers will get the `nomodule` payload.

4.7 New Mimetype for JavaScript Modules

In order to support JavaScript module programs, a new server mimetype has been created. All module URIs must conform to the new mimetype:

- name: “application/vnd.marklogic-js-module”
- Extension: “mjs”

You may view this new mimetype by navigating to the Admin UI and selecting the Mimetypes from the explorer pane.

The extension for a module URI in the import statement may be omitted. When the module URI in an import statement doesn’t contain an extension, an extension mapping to any of the above MIME types must be added to resolve the specified module. For example:

```
import { square, diag } from 'lib/top'; // map to lib/top.js or lib/top.mjs
```

4.8 Importing MarkLogic Built-In Modules

Based on these Performance Considerations, it is recommended that you import the following modules:

jsearch.mjs instead of jsearch.sjs.

optic.mjs instead of optic.sjs

Here is an example of importing optic.mjs:

```
'use strict';

import op from '/MarkLogic/optic.mjs';

op.fromLiterals([

            {group:1, val:2},

            {group:1, val:4},

            {group:2, val:3},

            {group:2, val:5},
```

```
              {group:2, val:7}

              ])

            .groupBy('group', op.avg('valAvg', 'val'))

            .result();

=>

{"group":1, "valAvg":3}

{"group":2, "valAvg":5}
```

Here is an example of importing jsearch.mjs:

```
// Find all documents where the "author" JSON property value is "Mark
Twain"

import jsearch from '/MarkLogic/jsearch.mjs';


jsearch.documents()

  .where(jsearch.byExample({author: 'Mark Twain'}))

  .result()

=>

{

"results": ...,

"estimate": 25

}
```

4.9 Evaluating Variables with ES6 Modules

Sometimes you will need to evaluate a JavaScript snippet from within an XQuery program. Prior to ES6, you could simply pass the names of the required variables as a map of string and value pairs. For instance, given the following program:

hello.sjs:

```
'use strict';
```

```
function specialFunction(

    x, y

){

    return "Hello " + x + " and " + y;

};

module.exports.specialFunction = specialFunction;
```

You could invoke it via the following code:

```
xquery version “1.0”;

let $x := "'use strict'; var ext=require('/hello.sjs');
ext.specialFunction(x,y);"

return

xdmp:javascript-eval($x,map:map()=>map:with("x","Laurel")=>map:with("y
","Hardy"));

=>

Hello Laurel and Hardy
```

In ES6 modules, the same code will complain about variable x being undefined. This is because in ES6, variables are available as properties on the `external` global object. So you will need to re-write your module in the following manner:

hello.mjs:

```
'use strict';

export function specialFunction(x,y)

{

    return "Hello " + x + " and " + y;

};
```

And your code that imports this module will look like this:

```
xquery version "1.0-ml";

let $x := "'use strict'; import {specialFunction} from '/hello.mjs';
specialFunction(external.x,external.y);"

return
```

```
xdmp:javascript-eval($x,map:map()=>map:with("x","Tom")=>map:with("y","
Jerry"));

=>

Hello Tom and Jerry
```

Note that use of `external.x` and `external.y` in the function call. This is the way to reference variables in the `external` global object.

5.0 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 technical questions, we encourage you to ask your question on Stack Overflow.

6.0 Copyright

MarkLogic Server 10.0 and supporting products.
Last updated: February, 2022

This technology is protected by U.S. Patent No. 7,127,469B2, U.S. Patent No. 7,171,404B2, U.S. Patent No. 7,756,858 B2, and U.S. Patent No 7,962,474 B2, US 8,892,599, and US 8,935,267.

The MarkLogic software is protected by United States and international copyright laws, and incorporates certain third party libraries and components which are subject to the attributions, terms, conditions and disclaimers set forth below.

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