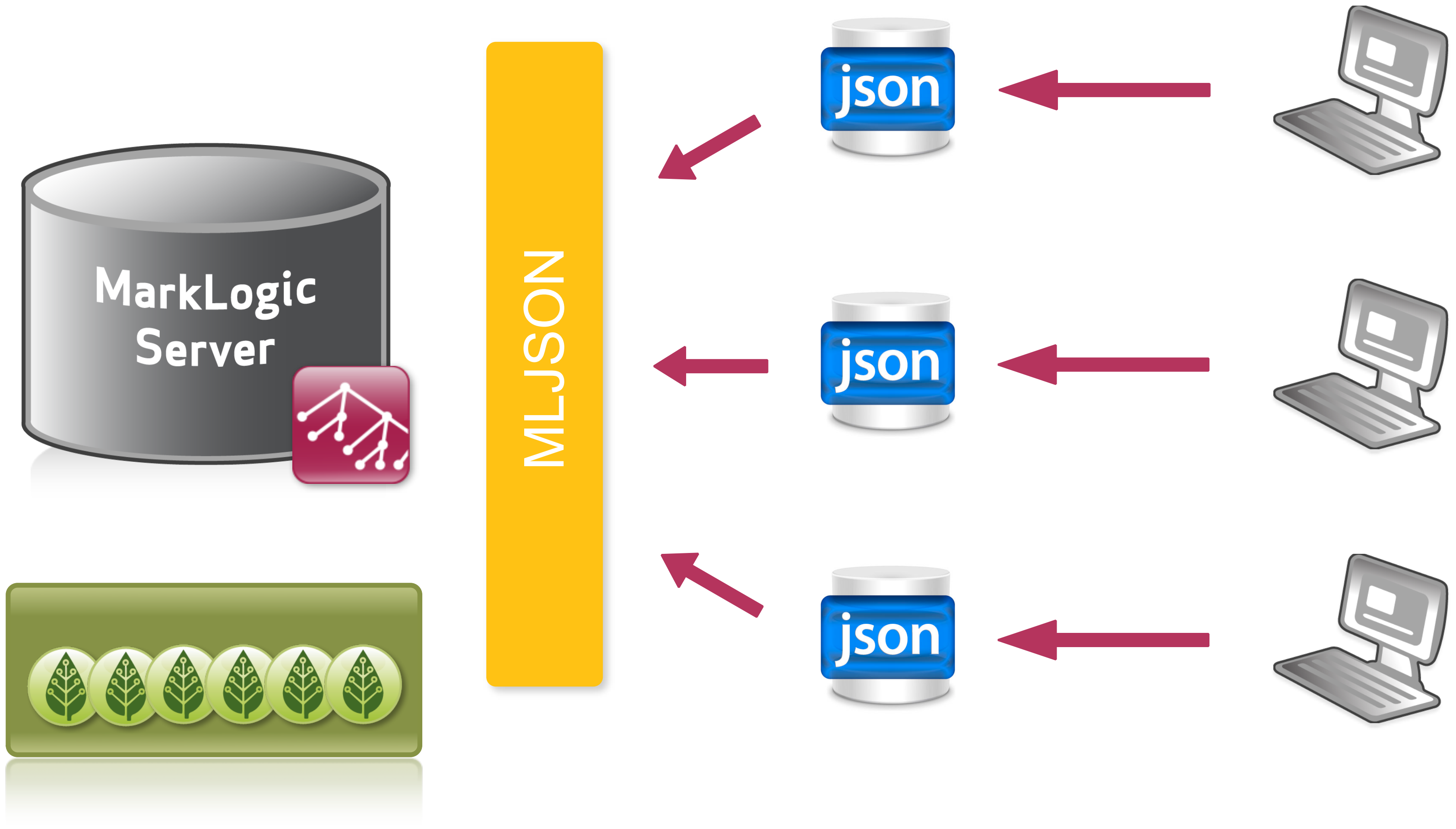
A JSON Facade on MarkLogic Server

Jason Hunter
Principal Technologist, Mark Logic

A facade around MarkLogic making it act as a JSON store



- JSON
  - JavaScript Object Notation
  - A lightweight data-encoding and interchange format
  - Native to JavaScript, now widely utilized across languages
  - Commonly used for passing data to web browsers

```
{
     "firstName": "John",
     "lastName": "Smith",
     "age": 25,
     "address": {
         "street": "21 2nd Street",
         "city": "New York",
         "state": "NY",
     },
     "phoneNumber": [
         {
           "type": "home",
           "number": "212 555-1234"
         }, {
           "type": "fax",
           "number": "646 555-4567"
         }
     ]
}
```

- Document-centric
- Transactional
- Search-centric
- Structure-aware
- Schema-free
- XQuery- and XSLT-driven
- Extremely fast
- Clustered
- Database server

MLJSON design considerations:

1. Approach things from a JSON angle
   - Create the XML to match the JSON, not vice-versa

2. Make good use of MarkLogic indexes
   - Craft the XML so it's fast to query

Expose the power and features of MarkLogic,
but against JSON instead of XML

- Internally there are calls to convert JSON to/from XML

```
declare function json:jsonToXML(
   $json as xs:string
) as element(json)

declare function json:xmlToJSON(
   $element as element(json)
) as xs:string
```

- Internally there are calls to convert JSON to/from XML

```
json:jsonToXML('{
  "iconKeySettings":[],
  "version":3,
  "time":"notime",
  "xAxis":"_NO",
  "duration":15,
  "iconType":"BUBBLE",
  "sizeOption":"_NO",
  "xZoomedDataMin":null,
  "xZoomedIn":false,
  "duration":{
    "multiplier":1,
    "timeUnit":"none"
  }
}')
```

```
<json type="object">
  <iconKeySettings type="array"/>
  <version type="number">3</version>
  <time type="string">notime</time>
  <xAxis type="string">_NO</xAxis>
  <duration type="number">15</duration>
  <iconType type="string">BUBBLE</iconType>
  <sizeOption type="string">_NO</sizeOption>
  <xZoomedDataMin type="null"/>
  <xZoomedIn boolean="false"/>
  <duration type="object">
    <multiplier type="number">1</multiplier>
    <timeUnit type="string">none</timeUnit>
  </duration>
</json>
```

- Rules:
  - Element names become QNames
  - Types get annotated as attributes; values are placed in text nodes
  - (Except booleans, an issue still up for debate)
  - Nulls are identified as a type

```
json:jsonToXML(
  '["hello world",
   [],
   {},
   null,
   false,
   true,
   9.99]'
)
```

```
<json type="array">
  <item type="string">hello world</item>
  <item type="array"/>
  <item type="object"/>
  <item type="null"/>
  <item boolean="false"/>
  <item boolean="true"/>
  <item type="number">9.99</item>
</json>
```

- Element name escaping?
  - An underscore initiates an escape sequence
  - Followed by 4 hex numbers defining the char it represents
  - An underscore itself gets escaped like any other char
  - An empty string gets a special rule: use a single underscore

String	Element Tag Name	Notes
"a"	<a>	
"1"	<_0031>	Element names can't start with a digit
""	<_>	Empty string
":"	<_003A>	
"_"	<_005F>	Underscores are escaped
"foo$bar"	<foo_0024bar>	

- Querying
  - Use native JSON syntax
  - Don't expose the XML internals to users
  - Support full range of MarkLogic indexes

```
declare function jsonquery:execute(
    $json as xs:string
) as element(json)*
```

- An example name/value query:

```
jsonquery:execute(
   '{key: "foo", value: "bar" }'
)
```

- Other JSON queries:

```
{key: "foo", value: ["bar","quux"] }
{innerKey: "foo", value: ["bar","quux"] }
{key: "foo"}
{key: "foo", value: { key: "id", value: "0596000405"}}
{key: "book", or: [{key: "id", value: "0596000405"},
                   {key: "other_id", value: "0596000405"}] }
{key: "price", value: 8.99, comparison:"<" }
```

JSON	Internal XPath Evaluated
{key:"foo", value:"bar"}	/json/foo[. = "bar"]
{key:"foo", value:["bar","quux"]}	/json/foo[. = ("bar","quux")]
{innerKey:"foo", value:["bar","quux"]}	/json//foo[. = ("bar","quux")]
{key:"foo"}	/json[exists(foo)]
{key:"foo", value: {key:"id", value:"0596000405"}}	/json[foo/id = "0596000405"]
{key:"book", or: [{key:"id", value:"0596000405"}, {key:"other_id", value:"0596000405"}]}	/json[exists(book)][id = "0596000405" or other_id = "0596000405"]
{key:"price", value:8.99, comparison:"<"}	/json[price < 8.99]

- A larger example taken from the unit tests:

```
{ fulltext: {
     or: [
         { equals: {
             key: "greeting",
             string: "Hello World",
             weight: 2.0,
             caseSensitive: false,
             diacriticSensitive: true,
             punctuationSensitve: false,
             whitespaceSensitive: false,
             stemmed: false,
             wildcarded: true,
             minimumOccurances: 1,
             maximumOccurances: null
         }},
{not: { contains: {
    key: "para",
    string: "Hello World",
    weight: 1.0
}}},
{andNot: {
    positive: { contains:{ key: "para", string: "excel"}},
    negative: { contains:{ key: "para", string: "pro"}}
}},
{property: { contains: {
    key: "para",
    string: "Hello World"
}}},
{ collection: "recent" },
  { geo: {
      parent: "location",
      latKey: "latitude",
      longKey: "longitude",
      key: "latlong",
      region: [
        {point: {longitude: 12, latitude: 53}},
        {circle: {longitude: 12, latitude: 53, radius: 10}},
        {box: {north: 3, east: 4, south: -5, west: -6}},
        {polygon:[
          {longitude:12, latitude:53},
          {longitude:15, latitude:57},
          {longitude:12, latitude:53}
      ]}
    ]
  }}
  ]
 }
}
```

- It produces this MarkLogic-native cts:query construct:

```
cts:or-query((
  cts:element-value-query(fn:QName("", "greeting"), "Hello World",
    ("case-insensitive","diacritic-sensitive","punctuation-insensitive",
     "whitespace-insensitive","unstemmed","wildcarded","lang=en"), 2),
  cts:not-query(cts:element-word-query(fn:QName("", "para"), "Hello World",
    ("lang=en"), 1), 1),
  cts:and-not-query(
    cts:element-word-query(fn:QName("", "para"), "excel", ("lang=en"), 1),
    cts:element-word-query(fn:QName("", "para"), "proceed", ("lang=en"), 1)),
  cts:properties-query(cts:element-word-query(fn:QName("", "para"),
    "Hello World", ("lang=en"), 1)),
  cts:collection-query("recent"),
  cts:element-pair-geospatial-query(fn:QName("", "location"),
    fn:QName("", "latitude"), fn:QName("", "longitude"),
    (cts:point("53,12"), cts:circle("@10 53,12"), cts:box("[-5, -6, 3, 4]"),
    cts:polygon("53,12 57,15 53,12")), ("coordinate-system=wgs84"), 1)
))
```

- There's a REST interface that hides the XQuery calls
  - A jsonstore.xqy endpoint page
  - You should use URL rewriting to hide this detail

- Includes: insert, fetch, update, delete, query

Command	Raw URL pattern
Insert a document (PUT)	/jsonstore.xqy?uri=http://foo/bar
Delete a document (DELETE)	/jsonstore.xqy?uri=http://foo/bar
Get a document (GET)	/jsonstore.xqy?uri=http://foo/bar
Set property (POST)	/jsonstore.xqy?uri=http://foo/bar &property=foo:bar
Set permissions (POST)	/jsonstore.xqy?uri=http://foo/bar &permission=foo:read&permission=bar:read
Set collections (POST)	/jsonstore.xqy?uri=http://foo/bar &collection=foo&collection=bar
Set document quality (POST)	/jsonstore.xqy?uri=http://foo/bar &quality=10

- Queries go to jsonquery.xqy

Command	Raw URL pattern
Query by price	/jsonquery.xqy?q={key:"price",value:"15",comparison:"<"}

- Results are (naturally) sent back as JSON:

```
{
  "count":1,
  "results":[
     {"book":"The Great Gatsby","author":
                "F. Scott Fitzgerald","price":12.99}
  ]
}
```

- Design review
  - JSON documents are converted and held as XML documents
  - The data model has been optimized for MarkLogic indexing
  - Queries are expressed using native JSON expressions
  - The REST interface hides the XQuery and MarkLogic interfaces
  - The REST interface stores and manages JSON documents
  - The REST interface queries for documents using a JSON syntax, returning a JSON syntax

It's JSON, JSON, JSON -- all on XML

- MLJSON pros and cons?
  - It's an easy and approachable storage model
  - It works well in the cloud
  - It requires basically no specialized expertise for use
- However...
  - It's simplistic compared to native MarkLogic functionality
  - One could implement the same abstracted cloud service using XML as the data type

- Links:
  - https://github.com/isubiker/mljson
  - http://developer.marklogic.com/code

Thank you!

Jason Hunter
jason.hunter@marklogic.com