JavaScript Applications with Oracle Database 12c using Nashorn and Avatar.js

Kuassi Mensah
Oracle Corporation
Redwood Shores

Keywords: JavaScript, Node.js, Java, JVM, Nashorn, Avatar.js

Introduction

The latest "*RedMonk Programming Languages Rankings*"[1] shows JavaScript and Java as the top two most popular programming languages. With Node.js and other frameworks, server-side JavaScript has become a popular choice for implementing Web, Mobile, and Cloud based applications.
The dream of most developers is the ability to use the same programming language across all tiers and layers; something Java has already accomplished. Can JavaScript follow Java footsteps and thrive on the middle and database[2] tiers?
With the advent of the Nashorn engine on the JVM, and Avatar.js it becomes tempting to co-locate Java and Node applications on the JVM. This paper describes the steps for running plain JavaScript stored procedures directly in Oracle database 12c using Nashorn and the steps for running Node.js compatible applications on the JVM using Nashorn, Avatar.js, JDBC and UCP.

JavaScript and the Evolution of Web Applications Architecture

At the beginning, JavaScript was exclusively used in browsers while business logic, presentations and back-end services connectivity where handled in middle-tiers using Java or other languages and frameworks. Then came browser independent JavaScript engines (Google's V8, Rhino), and server-side JavaScript frameworks such as Node.js and others.

Node Programming Model

Node.js brings a single-threaded, event-driven, and non-blocking programming model[3] to JavaScript. This model is being praised for its performance, ease of development, and rapid prototyping. We've heard the same eulogy for Ruby on Rails but experience has proved that you always need to dig deeper any language/framework, once you get the basic stuff magically working. One common concern with Node.js programming model is the so called "callback hell[4]" which requires some best practices. Also, unlike Java, Node lacks standardization in many areas such as database access (i.e., there is no JDBC equivalent, there are database specific drivers). Nonetheless, Node has a growing community and large set of frameworks (just search the web for "*Node.js frameworks*").

[1] http://redmonk.com/sogrady/2014/01/22/language-rankings-1-14/
[2] I'll discuss the rationale for running programming languages in the database, later in this paper.
[3] Request for I/O and resource intensive components run in separate proces then invoke a Callback in the mai/single Node thread, when done.
[4] http://callbackhell.com/

Node Impact on Web Application Architecture

With the advent of Node, REST and Web Sockets, the architecture of Web applications has evolved. The typical Web architecture comprises: (i) JavaScript client-side frameworks such as Angulatr.js on browsers, mobiles, tablets, desktops; (ii) Node.js, server-side JavaScript frameworks (e.g., ORM frameworks), Java business logic, and database connectivity on middle-tiers.

Nashorn: JavaScript Engine on the JVM.

Introduced in Java 7 but "production" in Java 8[5], the goal of project Nashorn (JEP 174), is to enhance the performance and security of the Rhino JavaScipt engine on the JVM. It integrates with javax.script API (JSR 223) and allows seamless interaction between Java and JavaScript (i.e., invoking Nashorn from Java and invoking Java from Nashorn).
To illustrate the reach of Nashorn on the JVM and the interaction between Java and JavaScript, let's run some JavaScript directly in Oracle database 12c.

Running JavaScript in Oracle database 12c Using Nashorn

Before looking into this proof of concept, why would anyone run JavaScript in the database?
For the same reasons than running Java in the database since 8*i*; namely: (i) reuse skills and code; (ii) avoid data shipping[6]; (iii) combine SQL with standard or 3rd party libraries to achieve new database capability thereby extending SQL and the reach of the RDBMS (e.g., uncommon data processing, Web Services callout).
Not every developer agrees with those reasons; some developers would prefer a tight separation between the RDBMS and applications code; in other words, no programming language in the database[7]. I respect such vision /principle but there are many pragmatic developers and architects out there who run programming languages code near data, when it is more efficient than shipping data to external infrastructure. Co-locating functions with data on the same compute engine is shared by many programming models such as Hadoop.
To conclude, running Java, JRuby, Python, JavaScript, Scala, or other programming language on the database is not a strange or scary thing to do. Best practice suggestions: (i) partition your application into data-bound and compute-bound modules; (ii) data-bound modules are good candidates for running in the database; (iii) understand DEFINER's vs INVOKER's right[8] and grant only the necessary privilege and/or permission.

Steps

The following steps allow implementing JavaScipt stored procedure running in Oracle database; these steps represent an enhancement from the ones presented at JavaOne and OOW 2014 -- which consisted in reading the JavaScript from the database file system; such approach required granting extra privileges to the database schema for reading from RDBMS file system something not recommended from security perspective. Here is a safer approach:

---

[5] Performance being one of the most important aspect
[6] My rule of thumb is: if you need to access and manipulate more than ~20-25% of data, better do it where data resides (i.e., function shipping).
[7] Other than database's specific procedural language, e.g., Oracle's PL/SQL
[8] I discuss this in chapter 2 of my book http://www.amazon.com/exec/obidos/ASIN/1555583296; see also Oracle database docs.

1. Nashorn is part of Java 8 but early editions can be built for Java 7; the embedded JavaVM in Oracle database 12c supports Java 6 (the default) or Java 7. For this proof of concept, install Oracle database 12c with Java SE 7 [9]

2. Build a standard Nashorn.jar[10]; (ii) modify the Shell code to interpret the given script name as an OJVM resource; this consists mainly in invoking `getResourceAsStream()` on the current thread's context class loader ; (iii) rebuild Nashorn.jar with the modified Shell

3. Load the modified Nashorn jar into an Oracle database shema e.g., HR

   ```
   loadjava -v -r -u hr/<password> nashorn.jar
   ```

4. Create a new `dbms_javascript` package for invoking Nashorn's Shell with a script name as parameter

   ```
   create or replace package dbms_javascript as
     procedure run(script varchar2);
   end;
   /
   create or replace package body dbms_javascript as
     procedure run(script varchar2) as
     language java name 'com.oracle.nashorn.tools.Shell.main(java.lang.String[])';
   end;
   /
   ```

   The goal is to call `dbms_javascript,run('myscript.js')` from SQL which will invoke Nashorn Shell to execute the previously loaded `myscript.js` (which contains plain JavaScript code).

5. Create a custom role, we will name it NASHORN, as follows, connected as `SYSTEM`

   ```
   SQL> create role nashorn;
   SQL> call dbms_java.grant_permission('NASHORN', 'SYS:java.lang.RuntimePermission',
   'createClassLoader', '' );
   SQL> call dbms_java.grant_permission('NASHORN', 'SYS:java.lang.RuntimePermission',
   'getClassLoader', '' );
   SQL> call dbms_java.grant_permission('NASHORN',
   'SYS:java.util.logging.LoggingPermission', 'control', '' );
   ```

   Best practice: insert those statements in a `nash-role.sql` file and run the script as `SYSTEM`

6. Grant the `NASHORN` role created above to the `HR` schema as follows (connected as `SYSTEM`):

   ```
   SQL> grant NASHORN to HR;
   ```

7. Insert the following JavaScript code in a file e.g., `database.js` stored on your client machine's (i.e., a machine from which you will invoke loadjava as explained in the next step).
   This script illustrates using JavaScript and Java as it uses the server-side JDBC driver to execute a `PreparedStatement` to retrieve the first and last names from the `EMPLOYEES` table.

[9] See Multiple JDK Support in http://docs.oracle.com/database/121/JJDEV/E50793-03.pdf

[10] Oracle does not furnish a public download of Nashorn.jar for Java 7; search "*Nashorn.jar for Java 7*".

```
var Driver = Packages.oracle.jdbc.OracleDriver;
var oracleDriver = new Driver();
var url = "jdbc:default:connection:";   // server-side JDBC driver
var query ="SELECT first_name, last_name from employees";
// Establish a JDBC connection
var connection = oracleDriver.defaultConnection();
// Prepare statement
var preparedStatement = connection.prepareStatement(query);
// execute Query
var resultSet = preparedStatement.executeQuery();
// display results
     while(resultSet.next()) {
     print(resultSet.getString(1) + "== " + resultSet.getString(2) + " " );
     }
// cleanup
resultSet.close();
preparedStatement.close();
connection.close();
```

8. Load `database.js` in the database
   `loadjava -v -r -u hr/<password> database.js`
9. To run the loaded script

```
sqlplus hr/<password>
SQL>set serveroutput on
SQL>call dbms_java.set_output(80000)
SQL>call dbms_javascript.run(‘database.js’);
```

The Nashorn Shell reads ‘`database.js`’ script stored as Java Resource from internal table; the JavaScript in its turn invokes JDBC to execute a `PreparedStatement` and the result set is displayed on the console. The message “*ORA=29515: exit called from Java code with status 0*” is due to the invocation of `java.lang.Runtime.exitInternal`; and status “0” means normal exit (i.e., no error).

Potential Enhancements

Some of the enhancements on the OOW/JavaOne wish list have already been folded into the steps described above but we will continue to look into more enhancements to the proof of concept such as:
(i) the ability to call JavaScript stored procedures as `CallableStatement` returning `refCursors` to middle-tier components. Typical use case: with JSON support in Oracle database 12.1.02, JavaScript stored procedures process JSON documents in the database and return the result sets. This is simply a matter of programming;
(ii) we should see dramatic performance improvement when/if Java 8 Nashorn is hopefully supported in future Oracle database releases.

And much more!

Node.js on the JVM: projects Avatar.js and Avatar

As discussed earlier, Node.js is becoming the man-in-the-middle between Web applications front ends and back-end components. Because many companies have invested in Java, it is highly desirable to co-locate Node.js and back-end Java components on the same JVM for better integration and the reduction/elimination of the communication overhead. Avatar.js, a Node.js compatibility framework running on top of Nashorn on the JVM accomplishes just that.

Project Avatar[11]

The goal of this Oracle research project is to furnish "*Enterprise Node.js on the JVM*"; in other words:

- Multiple Node event loops threads
- Shared sockets enable server applications to open the same port on multiple threads
- Coordination via JavaScript state sharing APIs (messaging and map)
- Persistence via JavaScript model APIs (SQL and NoSQL)
- Integration with Oracle Java EE products including WLS, Coherence, and so on

You may look at this as Node on steroids, a powerful combination of Java EE and Node.js! However, as of this writing, the details of the evolution of this research project have not been made public.

Project Avatar.js[12]

The goal of project Avatar.js is to furnish "*Node.js on the JVM*"; in other words, an implementation of Node.js APIs, which runs on top of Nashorn and enables the co-location of Node.js programs and Java components. Although inspired by project Avatar, as an enabling technology, it is technically independent (i.e., can be used stand-alone on Java SE) and has been outsourced by Oracle under GPL license[13]. Many Node frameworks and/or applications have been certified to run unchanged or slightly patched, on Avatar.js.
There are binary distributions for Oracle Enterprise Linux, Windows and MacOS (64-bits). These builds can be downloaded from https://maven.java.net/index.html#welcome. Search for `avatar-js.jar` and platform specific `libavatar-js` libraries (`.dll, .so, dylib`). Get the latest and rename the jar and the specific native libary accordingly. For example: on Linux, rename the libary to `avatar-js.so`; on Windows, rename the dll to `avatar-js.dll` and add its location to your PATH (or use `-Djava.library.path=<path to dll>).`

RDBMSes in general and Oracle database in particular remain the most popular persistence engines and there are RDBMS specific Node drivers[14] as well as ORMs frameworks. However, as we will demonstrate in the following section, with Avatar.js, we can simply reuse existing Java APIs including JDBC and UCP for database access. There as some limitations or enhancements requests discussed in this paper, but the integration with Java makes everything possible, right away!

[11] https://avatar.java.net/
[12] https://avatar-js.java.net/
[13] https://avatar-js.java.net/license.html
[14] The upcoming Oracle Node.js driver was presented at OOW 2014.

Oracle Database access using Avatar.js, JDBC and UCP

The goal of this proof of concept is to illustrate the co-location of a Node.js application, Avatar.js, the Oracle JDBC driver and the Universal Connection Pool (UCP) on the same Java 8 JVM (i.e., JDK). The sample application consists in a Node.js application which performs the following actions:

(i) Request a JDBC-Thin connection from the Java pool (UCP)
(ii)Create a PreparedStatement object for "`SELECT FIRST_NAME, LAST_NAME FROM EMPLOYEES`"
(iii)Execute the statement and return the ResultSet in a callback
(iv)Retrieve the rows and display in a browser on port 4000
(v) Perform all steps above in a non-blocking fashion – this is Node.js's *raison d'être*. The demo uses `Apache ab` load generator to simulate concurrent users running the same application in the same/single JVM instance.

However, for the JavaScript application to run in Node.js fashion, and in the absence of asynchronous JDBC APIs, we need to turn synchronous calls into non-blocking ones and retrieve the result set via callback.

Turning Synchronous JDBC Calls into Non-Blocking Calls

We will use the following wrapper functions to turn any JDBC call into a non-blocking call i.e., put the JDBC call into a thread pool and free up the Node event loop thread.

```
var makeExecutecallback = function(userCallback) {
      return function(name, args){
          ...
          userCallback(undefined, args[1]);
       }
     }
 function submit(task, callback, msg) {
         var handle = evtloop.acquire();
         try {     var ret = task();
                   evtloop.post(new EventType(msg, callback, null, ret)); {catch{}
    evtloop.submit(r);
     }
```

Let's apply these wrapper functions to executeQuery JDBC call, to illustrate the concept

```
exports.connect = function(userCallback) {..} // JDBC and UCP settings
Statement.prototype.executeQuery = function(query, userCallback) {
            var statement = this._statement;
             var task = function() {
             return statement.executeQuery(query);
           }
         submit(task, makeExecutecallback(userCallback), "jdbc.executeQuery");
     }
```

Similarly the same technique will be applied to other JDBC statement APIs.

```
Connection.prototype.getConnection = function() {…}
Connection.prototype.createStatement = function() {..}
Connection.prototype.prepareCall = function(storedprocedure) {..}
Statement.prototype.executeUpdate = function(query, userCallback) {..}
```

Returning Query ResultSet through a Callback

The application code fragment hereafter shows how: for every HTTP request: (i) a connection is requested, (ii) the PreparedStatement is executed, and (iii) the result set printed on port 4000.

```
...
var ConnProvider = require('./connprovider').ConnProvider;
var connProvider = new ConnProvider(function(err, connection){.. });

var server = http.createServer(function(request, response) {
  connProvider.getConn(function(name,data){..});
  connProvider.prepStat(function(resultset) {
                while (resultset.next()) {
                   response.write(resultset.getString(1) + " --" +
resultset.getString(2));
                   response.write('<br>');
    }
    response.write('</body></html>');
  response.end();
}
server.listen(4000, '127.0.0.1');
```

Conclusions

Through this paper, i discussed the rise of JavaScript for server-side programming and how Java is supporting such evolution; then – something we set out to demonstrate – furnished step by step details for implementing and running JavaScript stored procedures in Oracle database 12c using Nashorn as well as running Node.js applications using Avata.js, Oracle JDBC, UCP against Oracle database 12c.
As server-side JavaScript (typified by Node.js) gains in popularity -- i do not believe it'll takeover Java (COBOL is still alive!!) – it'll have to integrate with existing components. Companies, developers, architects will have to look into co-locating it with Java in middle and database tiers.

Contact address:

Kuassi Mensah
Oracle Corporation
400 Oracle Parkway
94065, Redwood Shores, CA
USA

Phone:	+1 415-8069937
Fax:	+1 650-5067525
Email	kuassi.mensah@oracle.com
Internet:	db360.blogspot.com