One solution for execution of JavaScript in Java EE application servers

Milan Vidaković, Stefan Ćosić, Ognjen Ćosić,
Ivan Kaštelan,
Faculty of technical Sciences,
University of Novi Sad,
Novi Sad, Serbia
minja@uns.ac.rs

Gordana Velikić
RT-RK Institute for Computer-Based Systems,
Novi Sad, Serbia
gordana.velikic@rt-rk.com

***Abstract*— This paper describes one solution for execution of JavaScript code inside Java Enterprise Edition (Java EE) application servers. Since Java Virtual Machine (Java VM) is able to execute JavaScript code, it is possible to integrate JavaScript into Java EE. Instead of implementing JavaScript code in Node.js server, we have decided to integrate JavaScript inside Java EE environment. This way, it is possible to call other Java EE modules within JavaScript code, since it is now a part of the enterprise system. To achieve that goal, we have designed an architecture which introduces a Middleware as a link between Enterprise system and JavaScript code. We have also measured performance of three usual platforms for JavaScript execution and concluded that JavaScript inside Java VM is not significantly slower than the Node.js system.**

Keywords—JavaScript; Java EE; Application Server

I. INTRODUCTION

This paper describes one solution for JavaScript execution on Java EE application servers. Although JavaScript is primarily used for web application development (executed in browsers), it is also used for the development of the server-side applications. Most notably, Node.js [1] is used to execute JavaScript programs on the server-side. However, Java VM also has the possibility of executing JavaScript programs. Since there are much more libraries and frameworks for the Java compared to the JavaScript, it could be beneficial to execute JavaScript programs inside the Java VM, because that way it would be possible to access those numerous Java-made libraries from the JavaScript. This is especially important if there is a need to integrate JavaScript code with Java EE environment.

Since Java EE relies on the Java VM, it is possible to execute JavaScript programs inside Java EE application servers. That way, it is possible to utilize many useful concepts existing in Java EE application servers, such as clustering, load balancing and session failover. The main motivation of this paper was to test if the JavaScript code can be integrated into the Java EE environment having reasonable performances.

Node.js is the most common system used for the server-side JavaScript execution. However, we have used the Nashorn[2] engine. This engine compiles JavaScript code into Java bytecode. This way, we are able to invoke Java code within the compiled JavaScript directly. The only downside is the speed of the Nashorn engine. We will address this in the Section 4.

The rest of the paper is organized as follows. Section 2 describes the related work. Section 3 presents details of the architecture presented in this paper. Section 4 presents the evaluation of the solution. The conclusion section outlines the presented work and proposes some future work.

II. RELATED WORK

Integration of JavaScript into the Java VM started as the Rhino project[3]. The main idea was to allow scripting languages like JavaScript to be executed within the Java VM. Besides just executing, the exchange of data in both directions (between Java and JavaScript) was implemented. Rhino was the default scripting engine for the JavaScript until it was replaced with the Nashorn engine[2] in Java 8. Nashorn introduced *invokedynamic* JVM opcode [4, 5], which was designed to enable usage of dynamically typed languages inside Java VM [6, 7]. Before this opcode was introduced, most of the dynamically typed emulations were done using reflection, which was significantly slower compared to *invokedynamic* opcode usage. Since Java 8 introduced lambda expressions, it was also possible to pass Java lambda expression as an argument to the JavaScript function, which would enable Java-made callbacks inside the JavaScript code.

As an alternative to Nashorn, the j2v8 project[8] has been created. This project aims at integrating Node.js server into the Java VM as a Java Native Interface (JNI) module[9]. Node.js is compiled as a dynamic link library and integrated inside the jar archive, with the corresponding API to invoke JavaScript code, enable data exchange and callbacks. This library is almost as fast as the stand-alone Node.js, which we will demonstrate in the Section 4.

III. SCRIPT SERVER ARCHITECTURE

The system consists of a Java EE application server, Script Engine, Middleware and JavaScript scripts, as shown in Figure 1.

ScriptEngine is a part of the Java VM, and therefore it is also a part of the Java EE application server. JavaScript code executes within the ScriptEngine. The link between Java EE code, various Java-based subsystems and JavaScript is implemented within the Middleware subsystem. It links various

Java-based subsystems to the JavaScript in a way that they are available to the JavaScript as normal JavaScript functions.

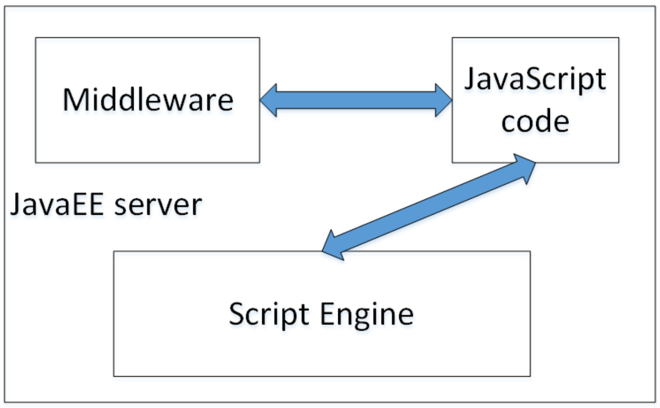


Fig. 1. Architecture of the system for execution of JavaScript in Java EE application servers

The Middleware wraps other Java-based subsystems as JavaScript functions. Each script is stored in the application server file system, while the script meta-data is stored in the database.

The execution of the JavaScript scripts within the server is based on the *ScriptEngineManager* class, which implements the factory design pattern and allows the creation of the *ScriptEngine* for JavaScript. This engine allows the execution of an arbitrary JavaScript program within the Java VM. Listing 1 shows how the JavaScript is executed within the Java VM.

```
ScriptEngineManager manager = new
ScriptEngineManager();

ScriptEngine engine =
manager.getEngineByName("JavaScript");

...

engine.eval(getSrc(getRealPath("middleware.js")));

engine.eval(getSrc(getRealPath(s.getFileName() + ".js")));

Invocable invocable = (Invocable) engine;
```

Listing 1: The execution of the script on the server using ScriptEngine mechanism

The part of the **middleware.js** file is shown in the Listing 2. It is a wrapper for the Java-made subsystems, which are linked to the JavaScript code through the Middleware. The Java-made code is seen from the JavaScript perspective as normal JavaScript functions. From the server-side perspective, these functions are realized as Java lambda functions.

```
var java = {};
var INVALID_PARAM = -1;
var print = function(msg) {};
var javaImport = function() {};
var myLogin =  function (arg) {
    if(arg == undefined){
          return java.userId;
    } else {
          return java.serverLogin(arg);
    }
};

function init(login, logout, getParameters,
saveResult,
enumerateParams,   javaPrint,   raiseAlarm,
getResults,      userId) {
    java.login = myLogin;
    java.serverLogin = login;
    java.logout = logout;
    java.getParameters = getParameters;
    java.saveResult = saveResult;
    print = javaPrint;
    java.raiseAlarm = raiseAlarm;
    java.getResults = getResults;
    java.userId = userId;
    java.params = enumerateParams(1);
}
```

Listing 2: Part of the middleware.js file

When functions from the **middeleware.js** file appear in the user's script, they are executed within the Java VM, as lambda functions. One example of the lambda function invoked from the JavaScript is shown in the Listing 3.

```
Function<Object, Object> getResults = (arg)
-> {

if (arg == null) {
    throw new RuntimeException("Missing
    arguments in getResults function");
}

if (arg instanceof ScriptObjectMirror) {
    ScriptObjectMirror _arg =
    (ScriptObjectMirror) arg;
    Double  sDate = (Double)
    _arg.get("startDate");
    Double  eDate = (Double)
    _arg.get("endDate");
    Long startDate = sDate.longValue();
    Long endDate = eDate.longValue();
    Integer scriptID = (Integer)
    _arg.get("scriptId");
    if (scriptID == null) {
          throw new
          RuntimeException("Missing
          scriptId argument in getResults
          function");
    }
    return
    resultsSubsystem.getResults( scriptID,
    startDate, endDate);
    }
    return DataServerRest.INVALID_PARAM;
};
```

Listing 3: Lambda function as a callback

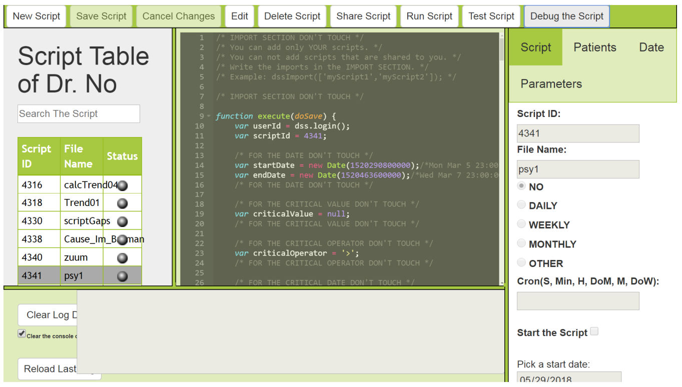
Within the lambda function, the first step is to identify passed parameters, and then to call the subsystem (in this example, stored in the *resultsSubsystem* variable). All defined lambda functions are passed to the script as parameters (the function *init()* in the Listing 2). Invocation of the *init()* function is shown in the Listing 4. This function is called before the script is executed, that way having all the subsystems passed to the script inside the *init()* function.

```
invocable.invokeFunction("init", login,
logout,    getParameters, saveResult,
enumerateParams, print,        raiseAlarm,
getResults, usrId);
```

Listing 4: Passing lambda expressions to the JavaScript as callbacks

A. Client Application

For the purpose of supporting the server, a client application was implemented in a form of control panel. It offers the basic operations for the work with scripts: creating, modifying, deleting, testing, sharing and executing. User-created scripts are shown on the left part of the site, while in the middle is the work area for the user to write his own code. On the right side there are three tabs (Scripts, Date and Parameters) and on the bottom there is a console, shown with the Figure 2. These three tabs are used to set up the script.



Script ID	File Name	Status
4316	calcTrend04	●
4318	Trend01	●
4330	scriptGaps	●
4338	Cause_Im_B●man	●
4340	zuum	●
4341	psy1	●

```
/* IMPORT SECTION DON'T TOUCH */
/* You can add only YOUR scripts. */
/* You can not add scripts that are shared to you. */
/* Write the imports in the IMPORT SECTION. */
/* Example: dssImport(['myScript1','myScript2']); */

/* IMPORT SECTION DON'T TOUCH */

function execute(doSave) {
    var userId = dss.login();
    var scriptId = 4341;

    /* FOR THE DATE DON'T TOUCH */
    var startDate = new Date(1520290800000);/*Mon Mar 5 23:00
    var endDate = new Date(1520463600000);/*Wed Mar 7 23:00:0
    /* FOR THE DATE DON'T TOUCH */

    /* FOR THE CRITICAL VALUE DON'T TOUCH */
    var criticalValue = null;
    /* FOR THE CRITICAL VALUE DON'T TOUCH */

    /* FOR THE CRITICAL OPERATOR DON'T TOUCH */
    var criticalOperator = '>';
    /* FOR THE CRITICAL OPERATOR DON'T TOUCH */

    /* FOR THE CRITICAL DATE DON'T TOUCH */
```

Script ID: 4341
File Name: psy1
[✓ NO] DAILY WEEKLY MONTHLY OTHER
Cron(S, Min, H, DoM, M, DoW):
Start the Script
Pick a start date: 05/29/2018

Fig. 2. Control panel for JavaScript execution within Java EE application server

After the creation of the script, the user can immediately run it. However, during the script development, it is usually better to test and debug the script before running it. Each user's script is saved as a JavaScript (.js) file on the server and when the user wants to run it, the server loads the .js file into the ScriptEngine which runs it then, as shown in the Listing 1.

The application offers the following functionalities:

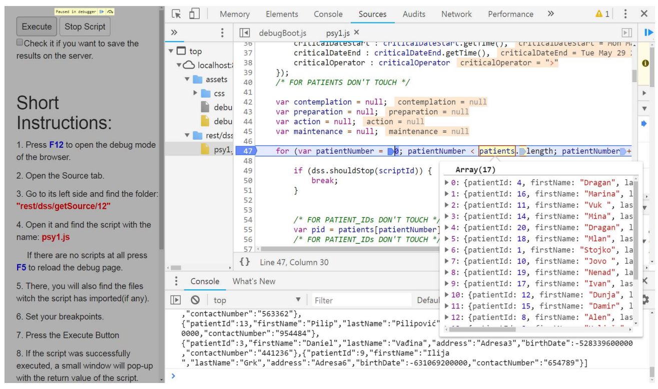
- management of the scripts, data and the results,
- execution of the script,
- script debugging, and
- display of the server console in the navigator.

The application itself creates and saves script files on the server who is running it. The user can change or delete all existing scripts. Each time the user saves the changes, the old content of the script is replaced with the new one on the server. The user can import other scripts into the script, and then run functions from imported scripts. Import of the scripts can be done in multiple levels, having one script importing second, second importing third, and so on, as shown in the Listing 5.

```
import([
    'myScript1',
    'myScript2',
    'myScript3'
]);
```

Listing 5: Importing other scripts from JavaScript code

During the script development, there is an option to test the script to see if it works properly (by pressing the "Test Script" button), or to run the script (by pressing the "Run Script" button). The difference is that the Test option will run the script without saving the data into the database, while the Run will save the data. Instead of testing or running, the user can also debug the script (by pressing the "Debug the Script" button). This will open the script in the built-in browser debugger, and the user would be able to watch variables, execute step-by-step, place breakpoints, etc., as shown in the Figure 3.



Check it if you want to save the results on the server.

Short Instructions:

1. Press F12 to open the debug mode of the browser.
2. Open the Source tab.
3. Go to its left side and find the folder: "rest/dss/getSource/12"
4. Open it and find the script with the name: psy1.js

If there are no scripts at all press F5 to reload the debug page.

5. There, you will also find the files witch the script has imported(if any).
6. Set your breakpoints.
7. Press the Execute Button
8. If the script was successfully executed, a small window will pop-up with the return value of the script.

```
        criticalDateStart : criticalDateStart.getTime(),  criticalDateStart = Mon M
        criticalDateEnd : criticalDateEnd.getTime(),  criticalDateEnd = Tue May 29
        criticalOperator : criticalOperator  criticalOperator = ">"
    });
    /* FOR PATIENTS DON'T TOUCH */

    var contemplation = null;  contemplation = null
    var preparation = null;  preparation = null
    var action = null;  action = null
    var maintenance = null;  maintenance = null

    for (var patientNumber = 0; patientNumber < patients.length; patientNumber++

        if (dss.shouldStop(scriptId)) {
            break;
        }


        /* FOR PATIENT_IDs DON'T TOUCH */
        var pid = patients[patientNumber]
        /* FOR PATIENT_IDs DON'T TOUCH */
```

Line 47, Column 30

```
Array(17)
▶0: {patientId: 4, firstName: "Dragan", la
▶1: {patientId: 16, firstName: "Marina", l
▶2: {patientId: 11, firstName: "Vuk ", las
▶3: {patientId: 14, firstName: "Mina", las
▶4: {patientId: 20, firstName: "Dragan", l
▶5: {patientId: 18, firstName: "Mlan", las
▶6: {patientId: 1, firstName: "Stojko", la
▶7: {patientId: 10, firstName: "Jovo ", la
▶8: {patientId: 19, firstName: "Nenad", la
▶9: {patientId: 17, firstName: "Ivan", las
▶10: {patientId: 12, firstName: "Dunja", l
▶11: {patientId: 15, firstName: "Damir", l
▶12: {patientId: 8, firstName: "Alen", las
```

```
,"contactNumber":"563362"},
{"patientId":13,"firstName":"Pilip","lastName":"Pilipović"
0000,"contactNumber":"954484"},
{"patientId":3,"firstName":"Daniel","lastName":"Vađina","address":"Adresa3","birthDate":-528339600000
,"contactNumber":"441236"},{"patientId":9,"firstName":"Ilija
","lastName":"Grk","address":"Adresa6","birthDate":-631069200000,"contactNumber":"654789"}]
```

Fig. 3. Browser-based debugger of the server-side script

IV. Evaluation

For the purpose of evaluation, we have created a complex JavaScript code which was executed in three environments:

- Wildfly application server with the default Nashorn engine,
- Wildfly application server with the j2v8 engine, and
- Node.js server

We have tried the same code on the single server with one client, on the single server with ten concurrent clients, and on the two-node cluster and ten concurrent clients. We have measured the execution several times and recorded the average execution time. The server in all three cases was i7 processor on 2.5 GHz and 16GB RAM.

Results are given in the Table 1.

TABLE I. AVERAGE EXECUTION TIMES IN THREE ENVIRONMENTS (NASHORN, J2V8 AND NODE.JS)

Configuration	**Server Types**		
	Node.js	***Wildfly-Nashorn***	***Wildfly-j2v8***
Single node-single client	2884ms	3972ms	2950ms
Single node-10 clients	8292ms	23988ms	9351ms
Two node cluster-10 clients	4412ms	12422ms	4845ms

The default JavaScript engine (Nashorn) was several times slower than the other two solutions. However, if the same code is executed several times (more than five) on the Nashorn engine, then the execution time comes close to other two solutions (Nashorn engine "warms up"). We have tried the Java version 8, while Java 9 authors claim that they have made the Nashorn faster. This is the job that we will perform as the future work.

V. CONCLUSION

This paper describes a Java Enterprise system for the JavaScript execution. The main motivation was to integrate JavaScript into the Enterprise environment, so it would be able to use other Java EE subsystems directly from the Java VM, where the script was executed. To perform the integration successfully, we have defined an architecture which consists of: Java EE application server, a custom Middleware and JavaScript engine. The Middleware is used to link the JavaScript engine to the Enterprise environment by exposing the Enterprise subsystems to the JavaScript code as common JavaScript functions. The other motivation for using JavaScript as compiled bytecode was the fact that JavaScript-originated bytecode can be deployed on Java EE application servers, and therefore can benefit from the their advanced features like clustering, load balancing and session failover.

To test the performance, we have two different JavaScript engines: Nashorn (default, already present in Java 8) and j2v8 (third-party library). Then we have compared the execution time of those two platforms to the Node.js-based execution time in a single server configuration, as well as in the clustered environment. Node.js was the fastest platform, j2v8 was just slightly slower than Node.js, while the Nashorn was the slowest. However, if the same code is executed several times on the Nashorn engine, then the execution time comes close to other two solutions, because, as the authors claim, the Nashorn engine "warms up". However, if the code is executed on the Nashorn engine just once or couple of times, then the execution time is several times slower than the other two solutions. Authors of Java 9 claim that they have optimised the Nashorn so its execution time is closer to the Node.js. This will be checked in our future work.

ACKNOWLEDGMENT

This work has been funded by the European Union's Horizon 2020 research and innovation project STARR under grant agreement No. 689947.

REFERENCES

[1] Node.js, https://nodejs.org/en/

[2] Nashorn engine, JSR-292, https://www.jcp.org/en/jsr/detail?id=292

[3] Rhino project, https://developer.mozilla.org/sr/docs/Mozilla/Projects/Rhino

[4] Francisco Ortin, Patricia Conde, Daniel Fernandez-Lanvin, Raul Izquierdo, "The Runtime Performance of invokedynamic: An Evaluation with a Java Library", IEEE Software 31(4), July 2014, DOI: 10.1109/MS.2013.46, pp. 82-90.

[5] Patricia Conde and Francisco Ortin, "JINDY: a Java library to support invokedynamic", Computer Science and Information Systems, 11(1), 2014, DOI: 10.2298/CSIS130129018C, pp. 47-68

[6] Andreas Gabrielsson, "On implementing multiple pluggable dynamic language frontends on the JVM", using the Nashorn runtime, student thesis, School of Computer Science and Communication, http://www.diva-portal.org/smash/get/diva2:860991/FULLTEXT01.pdf

[7] Sharan K. (2014) Scripting in Java. In: Beginning Java 8 APIs, Extensions and Libraries. Apress, Berkeley, CA, ISBN: 978-1-4302-6661-7, DOI: 10.1007/978-1-4302-6662-4_10

[8] j2v8 project, https://github.com/eclipsesource/J2V8

[9] Java Native Interface, https://docs.oracle.com/javase/8/docs/technotes/guides/jni/