# WA2579 Technical Introduction to Microservices

## Student Labs

## Web Age Solutions Inc.

## Table of Contents

Lab 1 - Monolith vs Microservices Design....3
Lab 2 - Getting Started With Node.js....7
Lab 3 - Getting Started with Spring Boot ....10
Lab 4 - Enable Basic Security....22
Lab 5 - Using AWS Lambda....26
Lab 6 - Angular Communication with REST Services....53
Lab 7 - Getting Started with Docker....72
Lab 8 - Getting Started with Kubernetes....84
Lab 9 - CI/CD with Jenkins, Docker, and OpenShift....95

# Lab 1 - Monolith vs Microservices Design

In this lab, you will be presented with two types of application architecture for a fictitious Java EE application: the monolith and a microservices-based one. You will be required to identify the main differences in their designs and answer some questions. For certain questions there is no single answer; feel free to share your answer(s) with the rest of the group.

## Part 1 - Breaking the Monolith

__1. Compare the Monolith Application Architecture 1 (Fig.1) with Microservices Architecture 2 (Fig.2), and name some of the differences.

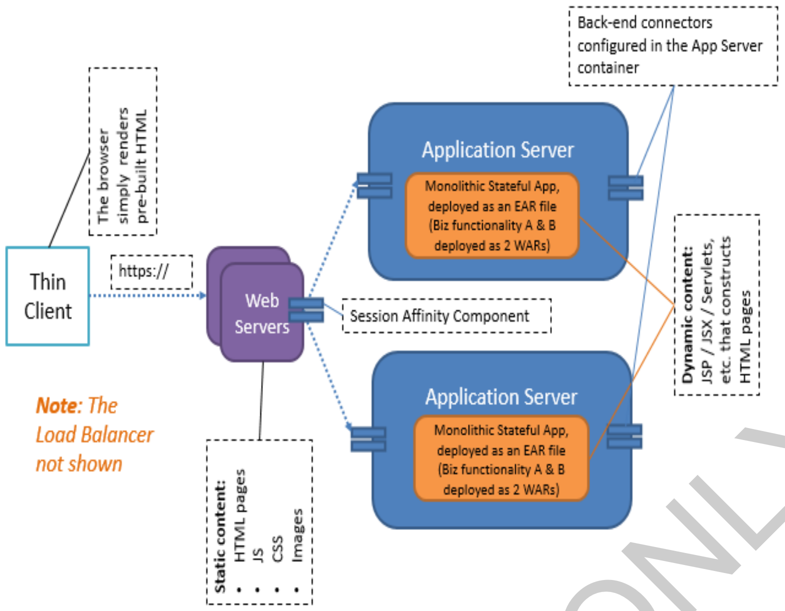

Back-end connectors configured in the App Server container
The browser simply renders pre-built HTML
Application Server
Monolithic Stateful App, deployed as an EAR file (Biz functionality A & B deployed as 2 WARs)
https://
Thin Client
Web Servers
Session Affinity Component
Dynamic content: JSP / JSX / Servlets, etc. that constructs HTML pages
Application Server
Monolithic Stateful App, deployed as an EAR file (Biz functionality A & B deployed as 2 WARs)
Note: The Load Balancer not shown
Static content:
• HTML pages
• JS
• CSS
• Images

Fig 1. Application Architecture 1: Traditional (Java) Enterprise Application Architecture Example.

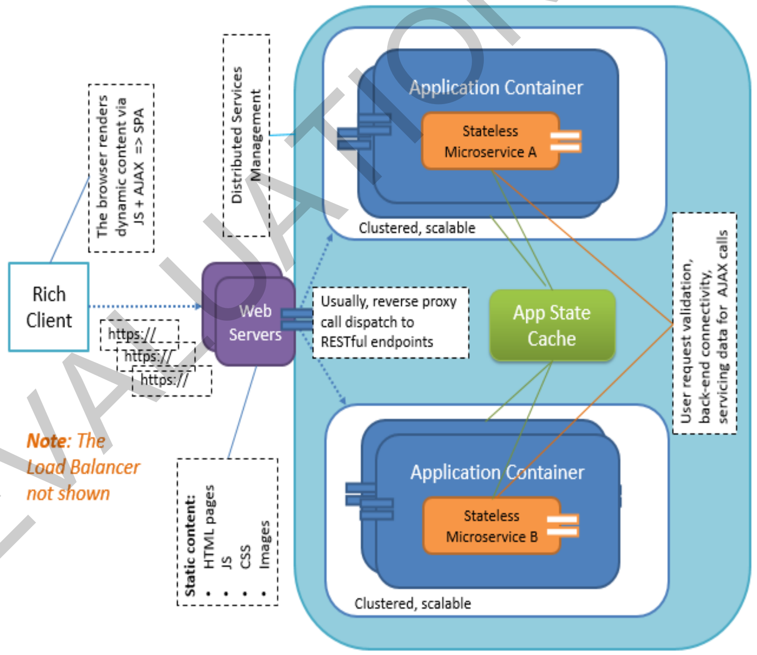

Distributed Services Management
The browser renders dynamic content via JS + AJAX => SPA
Application Container
Stateless Microservice A
Clustered, scalable
Rich Client
Web Servers
https://
https://
https://
Usually, reverse proxy call dispatch to RESTful endpoints
App State Cache
User request validation, back-end connectivity, servicing data for AJAX calls
Note: The Load Balancer not shown
Application Container
Stateless Microservice B
Static content:
• HTML pages
• JS
• CSS
• Images
Clustered, scalable

Fig. 2 Application Architecture 2: Microservices-based Architecture Example.

__2. What kind of Quality of Service properties we gain / lose by going from Architecture 1 to  Architecture 2?

__3. Which type of architecture more easily support scalability?

__4. What would be involved if you need to change a single Java file (e.g. a servlet)?

__5. In which type of architecture you need to have more powerful computers? Is it about CPU, RAM, or both?

__6. You need to recycle the machines hosting the applications. In which case you will have a shorter application start up time?

## Part 2 - The 12-Factor App

Go through the twelve-factor app principles [http://12factor.net] and try to see how, if at all, that methodology is applied in both App Architecture (Fig. 1 and Fig.2).

For example, you can notice that the Application in Fig.1 may be hard to scale through the process principle (VIII) that is broken as all code is bundled together (in one bundle / archive).

Mind you that not all factors can be identified here, though.

## Part 3 - Microservices Pros and Cons

Which of the following statements you believe may be related to microservices, and, if so, which ones should be viewed as being an argument in favor or against using microservices.  Compare/share your findings with the class, if you wish so.

1. As my app is deployed and executed in its own run-time (often in a container or a VM of sorts), I can get stronger process isolation properties.
2. Change cycles for different services can be more easily decoupled.  For example, I can re-deploy Service A without affecting Service B.
3. I have to deal with performance overhead related to inter-process communication.
4. Deployment cycles and developer velocity are faster (due to the smaller footprint of the service).
5. With this type of app, I can do scaling per service / per tier.
6. My app boasts a fantastic performance and fast service interactions; it also has a nice and small security perimeter.
7. Now I have to deal with a considerable operational overhead (the accidental complexity) with more distinct moving parts that require precisely orchestrated deployment and distributed monitoring.
8. My application has all the required services housed in one place and they get deployed using a single archive.  The app is designed using the standard SOA

principles: its services are mapped to distinct business capabilities, the services are narrow, composable, and accessible through a well-defined WSDL-based web service interface.

9. Now it is more of a nanoservice, really. Which is really cool, right?
10. Smaller development teams can be set up to develop and maintain the services which leads to faster getting-up-to-speed and development cycles.
11. Now, with this type of design, you must persist any state in a reliable external (e.g. caching) service.

## Part 4 - Review

In this lab, we compared two types of application architecture: the monolith and one based on microservices.

# Lab 2 - Getting Started With Node.js

In this lab we will explore how Node.js is installed and create a very simple Node.js application.

## Part 1 - Explore the Installation

To save time Node.js is already installed for you in your desktop. Let's verify to make sure things are installed correctly.

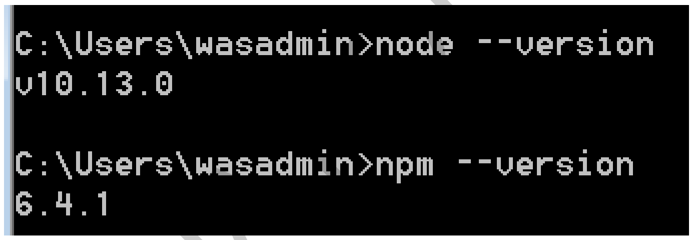
__1. Open a command prompt window.

__2. Enter this command to check the version of Node.js installed.

```
node --version
```

__3. In Windows installing Node.js also installs npm the package manager. Enter this command to check its version.

```
npm --version
```



```
C:\Users\wasadmin>node --version
v10.13.0

C:\Users\wasadmin>npm --version
6.4.1
```

__4. Open file explorer. Go to **C:\Program Files\nodejs** folder. This is where Node.js is installed. This folder is added to the PATH environment variable by the installer so that you can run the **node** and **npm** commands from anywhere.

## Part 2 - Create a Command Line Application

We will now create a very simple command line application using Node.js.

__1. Open a file browser and navigate to **C:\LabFiles**.

__2. In that folder create a file called **hello.js**

__3. Open the file using Visual Studio Code or any other editor. Visual Studio Code has been provided for you to use when editing code and related files. This editor includes excellent support for JavaScript files. That being said, feel free to use any editor that you are comfortable with.

__4. In hello.js enter this line.

```
console.log("Hello World!");
```

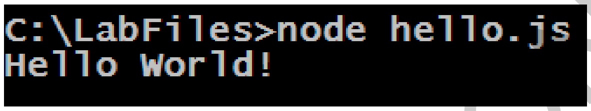
__5. Save the file.

__6. In the command prompt window change directory to **C:\LabFiles**.

```
cd C:\LabFiles
```

__7. From the command prompt enter this command to execute the program.

```
node hello.js
```



```
C:\LabFiles>node hello.js
Hello World!
```

It's that simple!

Let's make things a little more interesting. How about we supply the name of the planet to say hello as a command line argument? In Node.js you can access the command line parameters using the **process** global variable.

__8. Change the code of your program as follows.

```
var planet = "World"; //Default planet

if (process.argv.length > 2) {
  planet = process.argv[2]; //3rd parameter
}

console.log("Hello %s!", planet);
```

__9. Save changes.

__10. Try running the program in these ways.

```
node hello.js

node hello.js Mars
```

Even though JavaScript is an interpreted language, the V8 JavaScript engine used by Node.js has a Just in Time compiler that can compile certain sections of your code for better performance.

__11. Close the editor.

__12. Leave the command prompt open.

## Part 3 - Review

In this short lab we reviewed the installation of Node.js. We also wrote a simple application and executed it.

## Part 4 - Optional: Execute Node.js CLI with various arguments.

In this part, you will perform various operations by using the appropriate Node.js CLI argument. Each step of this part is independent of each other. You can consult the arguments by following this URL: https://nodejs.org/api/cli.html. Solutions are provided at the end of this lab guide.

__1. In the previous parts of this lab, you created C:\LabFiles\hello.js script. **node hello.js** was used to execute the script. Although, the syntax errors are checked by this command, your goal in this part is to check for syntax errors without executing the script. Use the appropriate argument and pass it to the node CLI for performing the syntax check.

__2. Pass an appropriate argument to the node CLI for executing the following code, without saving the script to a file:

```
console.log(5 + 6)
```

__3. Pass an appropriate argument to the Node.js CLI for print the value of the following expression:

```
5 + 6
```

## Solutions:

Check syntax: node --check hello.js

Execute code: node --eval "console.log(5 + 6)"

Print expression result: node --print "5 + 6"

__4. Close all.

# Lab 3 - Getting Started with Spring Boot

In this lab, you will learn how to create, build, run, and monitor a Spring Boot application.

To save your time, the code for the application has already been written for you.

To build our Spring Boot Java project, we will use the Apache Maven build tool (*https://maven.apache.org/* ) that has already been downloaded, installed, and configured on your lab machine.

## Part 1 - The Lab Working Directory

All the steps in this lab will be performed in the *C:\Works\* directory.

__1. Start a new command prompt window and type in the following command:

```
cd c:\Works
```

**Note**: If the directory does not exist, create it.

## Part 2 - Set Up the Build Project

A Maven build process requires your Java projects have the standard directory structure, which we are going to create as the first order of business.

__1. Enter the following command:

```
mkdir src\main\java\lab
```

__2. Enter the following command:

```
dir /b /s
```

You should see the following directory structure:

```
c:\Works\src
c:\Works\src\main
c:\Works\src\main\java
c:\Works\src\main\java\lab
```

__3. Copy the **pom.xml** file from the **C:\LabFiles\Spring Boot** directory to **C:\Works\**

**Note**: The pom.xml file is the Project Object Model descriptor that contains all the information Maven needs to know about your project (the dependencies, software versions, etc.)  This pom.xml has already been configured to build your Spring Boot application.

__4. Copy the **LicenseOfficeController.java** file from **C:\LabFiles\Spring Boot** to **C:\Works\src\main\java\lab\**

__5. Open the text editor of your choice and review the file's content, which is shown below for your reference:

```
package lab;

import java.util.concurrent.atomic.*;
import java.time.*;

import org.springframework.boot.*;
import org.springframework.boot.autoconfigure.*;
import org.springframework.stereotype.*;
import org.springframework.web.bind.annotation.*;

@SpringBootApplication
@RestController
public class LicenseOfficeController {

    private final AtomicLong buildingPermitId = new AtomicLong(999);

    @RequestMapping(value="/build-permit", method={RequestMethod.GET, RequestMethod.POST})
    public String getBuildingPermit(@RequestParam(value="ownerId", required=false, 
defaultValue="WildCat") String owner) {
        String timeStamp = LocalDate.now() + ":" + LocalTime.now() + " ";
        String permitId = "BuildingPermit-" + buildingPermitId.incrementAndGet() + "-" + 
owner;
        String response = timeStamp + permitId;
        System.out.println (response);
        return response;
    }

   public static void main(String[] args) throws Exception {
        SpringApplication.run(LicenseOfficeController.class, args);
    }
}
```

## Part 3 - Build the Project

In this lab part, we are going to use the **mvn** command line tool shipped with Maven to build an executable JAR file of our Spring Boot application that you can submit for execution by the Java Virtual Machine (JVM).

__1. In your command prompt window, enter the following command (you should be in the **c:\Works** folder)

```
mvn clean package
```

Wait for the build process to complete and display this message:

```
------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO]
------------------------------------------------------------------------
```

__2. Enter the following command:

```
dir /s *.jar
```

In the file listing produced, look for this entry which is the executable JAR file we are after:

```
c:\Works\target\SpringBootProject-1.0.1.jar
```

**Note**: The name of the file is generated from this configuration in pom.xml:

```
    <artifactId>SpringBootProject</artifactId>
    <version>1.0.1</version>
```

__3. Enter the following command:

```
jar -tvf  c:\Works\target\SpringBootProject-1.0.1.jar | findstr /i Tom
```

In the command output, you should see the embedded Tomcat JAR files:

```
239734 Thu Oct 06 21:16:28 EDT 2016 BOOT-INF/lib/tomcat-embed-el-8.5.6.jar
241565 Thu Oct 06 21:16:28 EDT 2016 BOOT-INF/lib/tomcat-embed-websocket-8.5.6.jar
2990538 Thu Oct 06 21:16:26 EDT 2016 BOOT-INF/lib/tomcat-embed-core-8.5.6.jar
```

## Part 4 - Run the Spring Boot Application

If all the instructions were followed properly so far, you should have an executable JAR file of your Spring Boot application runnable from command line.

__1. Enter the following command:

```
java -jar  c:\Works\target\SpringBootProject-1.0.1.jar
```

Look for the Tomcat web server start-up time in the command output (your timing will differ):

```
Started LicenseOfficeController in 4.331 seconds
```

This is one of the critical metric that you should consider for rapid provisioning of the HTTP web service endpoint.

**Note**: The **LicenseOfficeController.java** file contains the **main** method that is invoked by the JVM and bootstrapping the Spring Boot code.

__2. Open your browser and navigate to **http://localhost:8080/build-permit**

You should see the following message on the page returned by the server (your timestamp will differ):

```
2016-11-25:13:07:02.030 BuildingPermit-1000-WildCat
```

Since we have not provided the **ownerId**, the system plugged in the **WildCat** default value for the requester. All this is automatically handled by Spring MVC in this line:

```
@RequestParam(value="ownerId", required=false, defaultValue="WildCat")
```

__3. Switch to the command prompt window where we started the Spring Boot application.

Look for this message:

```
...INFO 2216 --- [nio-8080-exec-1] o.s.web.servlet.DispatcherServlet
FrameworkServlet 'dispatcherServlet': initialization started
```

The Spring MVC framework uses the *DispatcherServlet* web component to accept HTTP client calls, identify the target end-point by matching the path and HTTP verb (GET, PUT, etc.) and forward the HTTP request for processing there.

__4. Switch back to your browser open on **http://localhost:8080/build-permit**

__5. Update the URL to **http://localhost:8080/build-permit?ownerId=abc123** and submit request.

You should get the following server response:

```
2016-11-25:13:17:10.843 BuildingPermit-1001-abc123
```

As you can see, the *@RequestParam(value="ownerId", required=false, defaultValue="WildCat") String **owner**)* annotation does its job for extracting the **ownerId** query parameter and mapping it to the *String* **owner** variable.

## Part 5 - Submit a POST Request

Our application claims to process not only GET requests but also requests submitted with the POST HTTP verb as indicated in this REST annotation:

```
 @RequestMapping(..., method={RequestMethod.GET, RequestMethod.POST})
```

Let's quickly verify this with the handy **curl** tool.

__1. Open a new command prompt window and enter the following command in one line:

```
C:\LabFiles\curl -i -X POST -d "ownerId=u2"
http://localhost:8080/build-permit
```

Our  RESTful web service should get back to your with the following response:

```
HTTP/1.1 200
Content-Type: text/plain;charset=UTF-8
Content-Length: 46
Date: Fri, 25 Nov 2016 18:38:50 GMT

2016-11-25:13:38:50.427 BuildingPermit-1005-u2
```

**Note**: In the above command we instructed *curl* to include the HTTP response headers (with the **-i** flag).  Notice that the embedded Tomcat web container has not identified itself in any way – which is a good practice from the security standpoint, as a potential hacker would be none the wiser about the web server type and its potential vulnerabilities she or he would want to exploit.

Now let's see what would happen should we submit a request via an unsupported HTTP verb – **PUT**

__2. Enter the following command:

```
C:\LabFiles\curl -i -X PUT -d "ownerId=u2" http://localhost:8080/build-
permit
```

You should get a 405 HTTP  "Method Not Allowed" error code reserved by REST protocol for handling such situations:

```
HTTP/1.1 405
Allow: GET, POST
Content-Type: application/json;charset=UTF-8
Transfer-Encoding: chunked
Date: Fri, 25 Nov 2016 18:48:05 GMT

{"timestamp":1480099685874,"status":405,"error":"Method Not
Allowed","exception":"org.springframewor
k.web.HttpRequestMethodNotSupportedException","message":"Request method
'PUT' not supported","path":
"/build-permit"}
```

__3. Keep this command prompt window open.

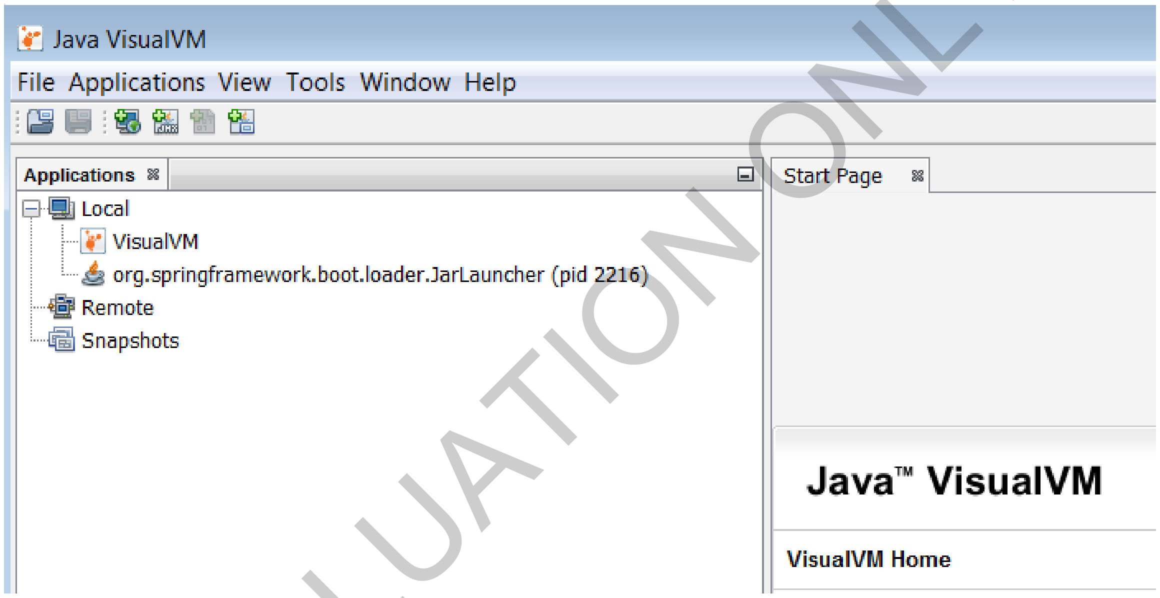
## Part 6 - Use JMX Console to Get Run-time Metrics

In this lab part, you are going to use the Java Management Extension tool bundled with the recent Java Development Kit (JDK) versions for the purposes of collecting vital run-time metrics of Java applications which you can also use for checking applications' health.

__1. In the command prompt window you used for running **curl**, enter the following command:

```
jvisualvm.exe
```

The Java VisualVM console should open.

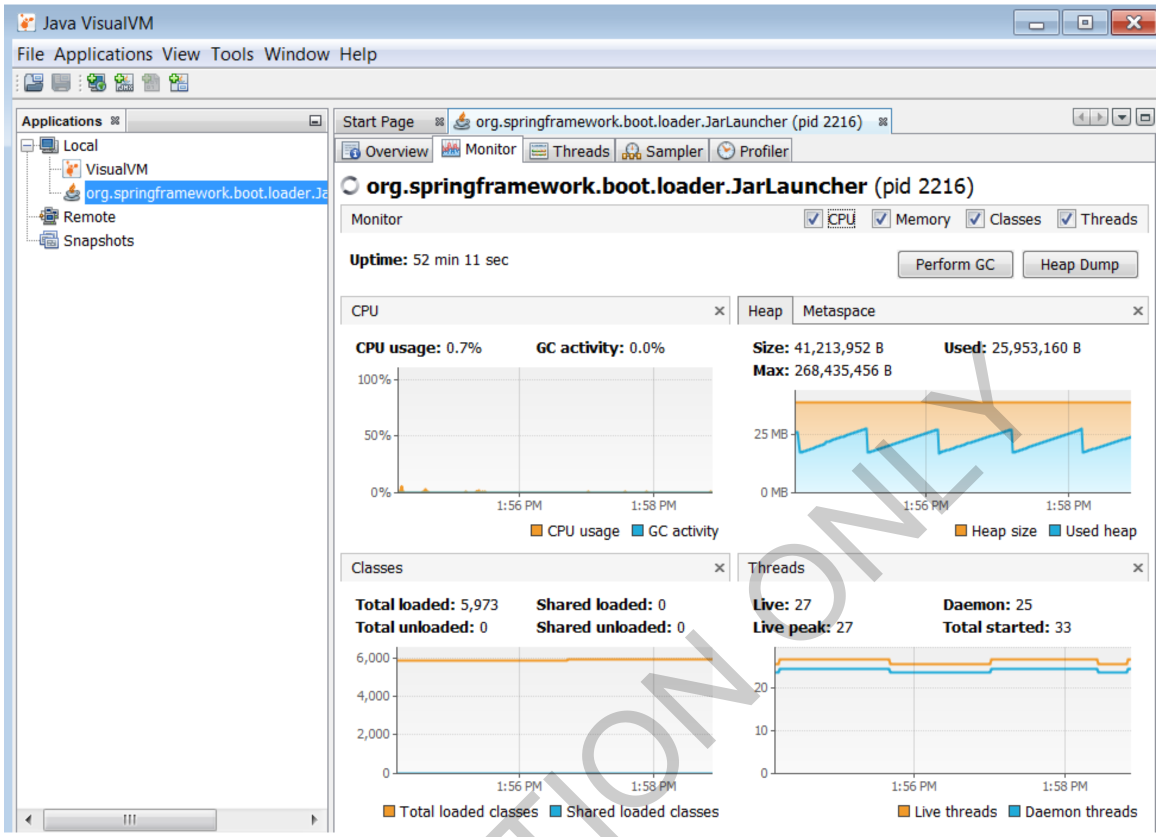

Java VisualVM
File Applications View Tools Window Help
Applications
Local
VisualVM
org.springframework.boot.loader.JarLauncher (pid 2216)
Remote
Snapshots
Start Page
Java™ VisualVM
VisualVM Home

__2. Double-click the **org.springframework.boot.loader.JarLauncher (pid xxxx)** entry under the **Local** node.

The run-time metrics dashboard of the selected process should open in the **Overview** tab.

__3. Click the **Monitor** tab.

You should see all the process vitals: CPU usage, Heap load, etc. captured and displayed dynamically:



The *Heap* section of the *Monitor* pane is the place to monitor for any memory leaks – situations when the Garbage Collector tool (GC ) is not able to reclaim the memory over a long period of time – that you may have.

Take some time to familiarize yourself with the Java VisualVM tool. See if you can answer this questions: How many threads are allocated by Spring Boot (more specifically, by embedded Tomcat) to process client requests (the name of such a thread should be *http-nio-xxxxx* and most of them, if not all, should be in the **Park** state.)

If you find yourself on the **Threads** tab, feel free to click the **Tread Dump** button and observe the stack traces (the active stack frames) of all the threads in a new tab that would open. It might give you some additional insights.

__4. When you are done, click the **x** icon in the **org.springframework.boot.loader.JarLauncher** tab (it should be next to the **Start Page** tab).

__5. Close the Java VisualVM application.

__6. Keep this command prompt window running.

## Part 7 - Change Port

By default, Spring Boot web applications, when started, listen on port 8080 (the default port used by Tomcat). Sometimes you may want to change it to prevent port conflicts or for other such reasons. Let's change the default port 8080 to another value, say, 18080.

__1. Switch to the command prompt window where you are running the application (you should be in the **c:\Works** directory), press **Ctrl-C** to stop it .

__2. Enter the following command:

```
java -jar  -Dserver.port=18080 c:\Works\target\SpringBootProject-1.0.1.jar
```

In the diagnostic messages printed out to the console, locate this line confirming that the change took effect as intended:

```
Tomcat initialized with port(s): 18080 (http)
```

__3. Switch to the command prompt window where you ran *curl* and the Java VisualVM application.

__4. Enter the following command (use the arrow keys, or use the **F7** selector to retrieve the command from the command history):

```
C:\LabFiles\curl -i -X POST -d "ownerId=u2"
http://localhost:8080/build-permit
```

You should get

```
Failed to connect to localhost port 8080: Connection refused
```

__5. Correct the port in the above *curl* command (it should be **18080** now) and re-submit the request.

```
C:\LabFiles\curl -i -X POST -d "ownerId=u2"
http://localhost:18080/build-permit
```

You should be able to go through now.

__6. Switch to the command prompt window where you are running the application (your should be in the **c:\Works** directory), press **Ctrl-C** to stop it.

## Part 8 - Refactor the Application

Our *LicenseOfficeController* application has both the RESTful web service code written as per Spring MVC API and the bootstrapping code as per the Spring Boot framework. While it may be OK for quick prototypes, it may become a maintenance problem as your application becomes larger with more RESTful end-points added.

Let's refactor our code such that we will have two separate modules: a bootstrapping module (with the **main** method), named **LicenseOfficeRunner**, and a RESTful web service module, named **LicenseOfficeController** (named as the current module).

__1. In your text editor open on the *LicenseOfficeController.java* file, apply the following changes:

```
1. Comment out the main method
2. Comment out the @SpringBootApplication annotation
```

The updated sections of the *LicenseOfficeController.java* file are shown in bold font below:

```
...
//@SpringBootApplication
@RestController
public class LicenseOfficeController {
...
/*
   public static void main(String[] args) throws Exception {
        SpringApplication.run(LicenseOfficeController.class, args);
    }
*/
...
```

After you have removed the Spring Boot-specific bootstrapping code, the module becomes a regular Spring MVC module, which is easier to maintain and extend by adding new methods to handle user requests.

__2. Save the file.

__3. Now save the **LicenseOfficeController.java** file as **LicenseOfficeRunner.java** (it should be saved in the same directory as *LicenseOfficeRunner.java*).

__4. Edit the **LicenseOfficeRunner.java** file as follows:

```
1. Remove the getBuildingPermit() method
2. Uncomment the main method
3. Uncomment the @SpringBootApplication annotation
4. Remove the AtomicLong var
5. Remove the RestController annotation
6. Rename the class
```

Now your updated code should look as such:

```
package lab;

import java.util.concurrent.atomic.*;
import java.time.*;

import org.springframework.boot.*;
import org.springframework.boot.autoconfigure.*;
import org.springframework.stereotype.*;
import org.springframework.web.bind.annotation.*;

@SpringBootApplication
public class LicenseOfficeRunner {

    public static void main(String[] args) throws Exception {
        SpringApplication.run(LicenseOfficeRunner.class, args);
    }
}
```

__5. Make sure you replace this line:

```
    SpringApplication.run(LicenseOfficeController.class, args);
```

with that one:

```
    SpringApplication.run(LicenseOfficeRunner.class, args);
```

After our refactoring exercise, we, overall, have a better structure of our application as you can now add more methods that serve RESTful endpoints and more separate modules to your application as may be required.

__6. Save the file.

__7. In your command prompt window where you ran the application, enter the following command (you should be in the **c:\Works** folder)

```
mvn clean package
```

The system will go ahead and compile two files:

```
BOOT-INF/classes/lab/LicenseOfficeController.class
BOOT-INF/classes/lab/LicenseOfficeRunner.class
```

and will build the application JAR file.

__8. Enter the following command to start the refactored Spring Boot application:

```
java -jar c:\Works\target\SpringBootProject-1.0.1.jar
```

Spring Boot will correctly identity the *LicenseOfficeRunner* class with the **main** method that bootstraps our application.

```
[           main] lab.LicenseOfficeRunner
Started LicenseOfficeRunner in 4.157 seconds (JVM running for 4.805)
```

__9. Switch to the command prompt window where you ran *curl* and issue this command (in one line):

```
C:\LabFiles\curl -X GET http://localhost:8080/build-permit
```

You should get the expected server response:

```
... BuildingPermit-1000-WildCat
```

## Part 9 - Working Area Clean-up

We are almost done in the lab.  Time for doing some clean-up.

__1. Close all files opened in your text editor; close the text editor.

__2. In the command prompt window where you run the Spring Boot application, press **Ctrl-C** to stop the web server.

__3. Close the open command prompt windows.

__4. Close the browser.

## Part 10 - Review

In this lab, you learned how to create, build, run, and monitor a Spring Boot application. We used a number of tools to do that: Maven for building the executable JAR file of our application, the Java VisualVM tool for application monitoring at run-time, and the *curl* tool as a REST web service client.