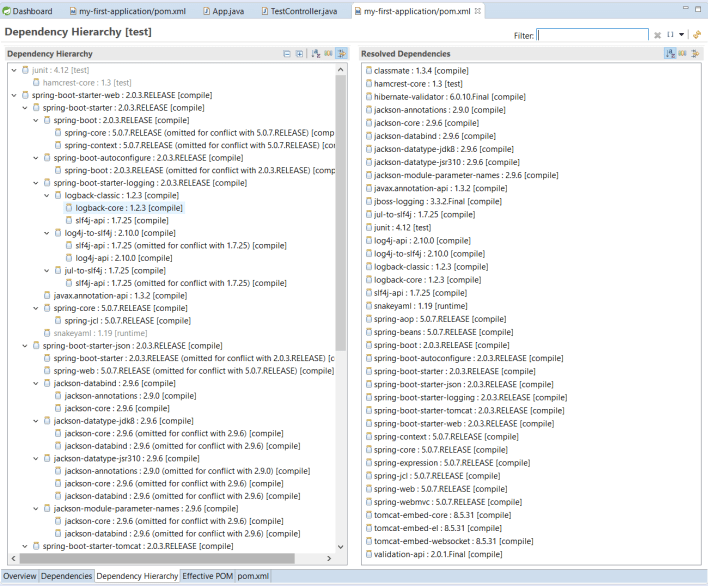
## How Does it Work Behind the Scenes?

### **Dependency Management**

After you added your starting dependencies, Maven downloaded all of the libraries considered to be necessary for our project.

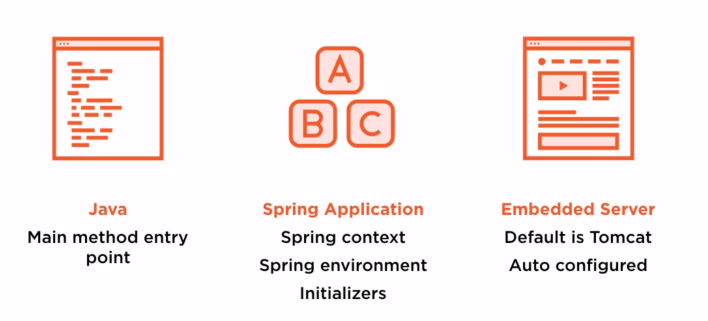
You can check this by opening the pom.xml and clicking on the dependency hierarchy tab shown below:



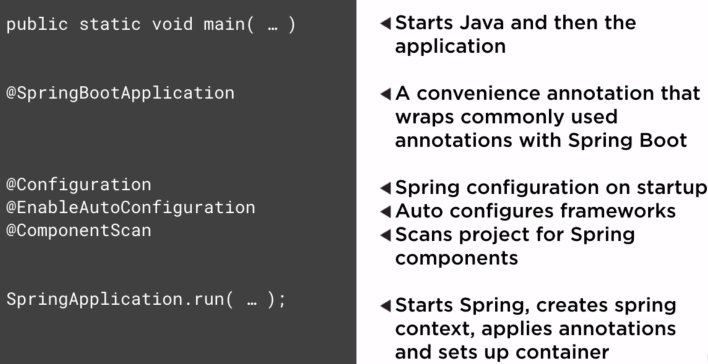
It knows which dependency versions are the latest and stable for each other. This makes our life easier, because you do not need to take care of the different transitive dependencies. You only need to add one starter dependency, and your greatest dependencies start to download automatically.

The  spring-boot-starter-parent that is added to the starter section tells the default version of our Spring Boot which one we would like to use. Since we are developing a web application, we also need to add spring-boot-starter-web dependency.This will include additional dependencies such Spring boot, tomcat etc which are required for this application.  
If you notice, we did not provide any version for specific components. You just need to provide version no.(1.5.3.RELEASE) for spring boot.

### **How Does a Spring Boot Application Start?**



* Firstly, the application starts with a simple Java public static main method.
* Next, Spring starts your Spring context by looking up the auto-config initializers, configurations, and annotations that direct how to initialize and start up the Spring context.
* Lastly, it starts and auto-configures an embedded web server. The default application server is Tomcat for Spring Boot.

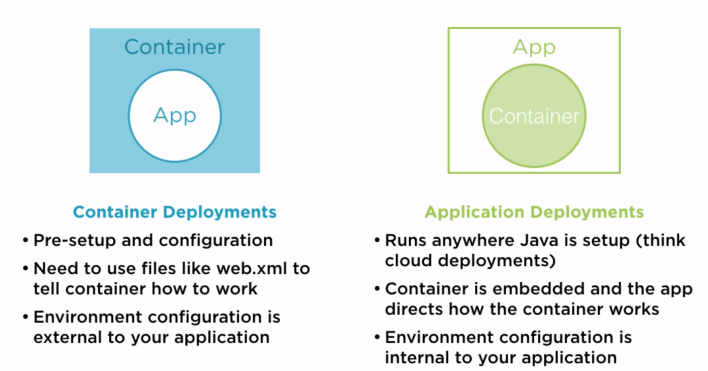


### **Benefits of Containerless Deployment**

As you can see here, we didn’t need to configure our web-container. Therefore, we gain a lot of time and could get rid of some of our configuration files.

The reason that Spring Boot runs an embedded web-server and configures it itself is that it knows that we need it from the web dependency, which we added to our pom.xml.

Spring Boot opens up a bunch of new opportunities for us— we can simply run a web app by copying a basic .jar  file anywhere Java is installed and just run it. This is a big step towards cloud architecture because we can handle our independent deployments more easily than before.



**. Spring Boot 2 REST API Controller**

* In Spring, a controller class, which is capable of serving REST API requests, is called rest controller. It should be annotated with **@RestController** annotation.
* The resource uris are specified in **@RequestMapping** annotations. It can be applied at class level and method level both. Complete URI for an API is resolved after adding class level path and method level path.

## @RestController

@RestController is a convenience annotation for creating Restful controllers. It is a specialization of @Component and is autodetected through classpath scanning. It adds the @Controller and @ResponseBody annotations. It converts the response to JSON or XML. It does not work with the view technology, so the methods cannot return ModelAndView. It is typically used in combination with annotated handler methods based on the @RequestMapping annotation.

The @Controller annotation is used with the view technology.

**How to remove datasource configuration error selecting web and jpa dependency?**

**import** org.springframework.boot.SpringApplication;

**import** org.springframework.boot.autoconfigure.EnableAutoConfiguration;

**import** org.springframework.boot.autoconfigure.SpringBootApplication;

**import** org.springframework.boot.autoconfigure.jdbc.DataSourceAutoConfiguration;

**import** org.springframework.boot.autoconfigure.jdbc.DataSourceTransactionManagerAutoConfiguration;

**import** org.springframework.boot.autoconfigure.orm.jpa.HibernateJpaAutoConfiguration;

@SpringBootApplication

@EnableAutoConfiguration(exclude = {DataSourceAutoConfiguration.**class**, DataSourceTransactionManagerAutoConfiguration.**class**, HibernateJpaAutoConfiguration.**class**})

**public** **class** SpringBootWebProjectApplication {

**public** **static** **void** main(String[] args) {

SpringApplication.*run*(SpringBootWebProjectApplication.**class**, args);

}

}

## **What are the maven dependencies required for Spring Boot Web Project:**

Maven Dependencies

* **spring-boot-starter-parent:** provides useful Maven defaults. It also provides a dependency-management section so that you can omit version tags for existing dependencies.
* **spring-boot-starter-web:** Spring Boot Starter Web provides all the dependencies and the auto configuration needed to develop web applications.
* We want to use JSP as the view. Default embedded servlet container for Spring Boot Starter Web is tomcat. To enable support for JSP’s, we would need to add a dependency on tomcat-embed-jasper.

<dependency>

<groupId>org.apache.tomcat.embed</groupId>

<artifactId>tomcat-embed-jasper</artifactId>

<scope>provided</scope>

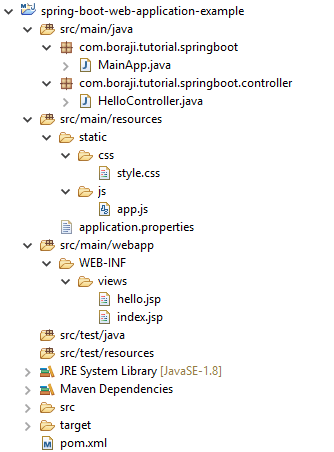
</dependency>

**Spring Boot Starter Web** auto-configures:

* Dispatcher Servlet.
* Error Page.
* Web Jars to manage your static dependencies.
* Embedded Servlet Container - Tomcat is the default.

### **Project Structure**

Review the spring boot project structure.



### **Jar dependencies**

To create and run a Spring MVC web application in spring boot, you need to add the spring-boot-starter dependency in your pom.xml file.

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-web</artifactId>

</dependency>

<dependency>

<groupId>org.apache.tomcat.embed</groupId>

<artifactId>tomcat-embed-jasper</artifactId>

<scope>provided</scope>

</dependency>

### **Controller class**

Create a HelloController class under com.tutorial.springboot.controller package and write the following code in it.

**HelloController.java**

**package** com.tutorial.springboot.controller;

**import** org.springframework.stereotype.Controller;

**import** org.springframework.ui.Model;

**import** org.springframework.web.bind.annotation.PostMapping;

**import** org.springframework.web.bind.annotation.RequestMapping;

**import** org.springframework.web.bind.annotation.RequestParam;

@Controller

**public** **class** **HelloController** {

@RequestMapping("/")

**public** String **index**() {

**return** "index";

}

@PostMapping("/hello")

**public** String **sayHello**(@**RequestParam**("name") String name, Model model) {

model.addAttribute("name", name);

**return** "hello";

}

}

### **JSP Views**

Create index.jsp and hello.jsp  files under src/main/webapp/WEB-INF/views folder as shown in the project structure.

**index.jsp**

<%@ page language="java" contentType="text/html; charset=ISO-8859-1" pageEncoding="ISO-8859-1"%>

**<!DOCTYPE html>**

<html>

<head>

<meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1">

*<!-- Static content -->*

<link rel="stylesheet" href="/resources/css/style.css">

<script type="text/javascript" src="/resources/js/app.js"></script>

<title>Spring Boot</title>

</head>

<body>

<h1>Spring Boot - MVC web application example</h1>

<hr>

<div class="form">

<form action="hello" method="post" onsubmit="return validate()">

<table>

<tr>

<td>Enter Your name</td>

<td><input id="name" name="name"></td>

<td><input type="submit" value="Submit"></td>

</tr>

</table>

</form>

</div>

</body>

</html>

**hello.jsp**

<%@ page language="java" contentType="text/html; charset=ISO-8859-1" pageEncoding="ISO-8859-1"%>

<%@ taglib uri="<http://java.sun.com/jsp/jstl/core>" prefix="c"%>

**<!DOCTYPE html>**

<html>

<head>

<meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1">

<title>Spring Boot</title>

</head>

<body>

<h1>Spring Boot - MVC web application example</h1>

<hr>

<h2>Your name is ${name}</h2>

</body>

</html>

### **Static web resources**

By default Spring Boot serves the static web content from the /static or /public or /resources or /META-INF/resources folder in a classpath. In this example, we will put all static resources such as  stylesheet, JavaScript files under /static folder.

Create a stylesheet file under src/main/resources/static/css folder and write the following code in it.

**style.css**

.form {

background-color: #efefef;

width: 400px;

height: 50px;

border-radius: 7px;

padding: 20px;

}

Now, create a javascript file under src/main/resources/static/js folder and write the following code in it.

**app.js**

**function** **validate**() {

**var** name = document.getElementById("name").value;

**if** (name == '') {

alert('Please enter a valid name.');

**return** false;

} **else** {

**return** true;

}

}

### **Application properties**

Spring boot load the properties from application.properties and add them to the Spring Environment. You can set the properties related to the Spring MVC or static web content in application.properties file.

Create an application.properties file under src/main/resources source folder and write the following properties in it.

**application.properties**

**spring**.mvc.view.prefix = /WEB-INF/views/

**spring**.mvc.view.suffix = .jsp

**spring**.mvc.static-path-pattern=/resources/\*\*

**Note -**spring.mvc.static-path-pattern=/resources/\*\* will map the classpath:/static/css/style.css to /resources/css/style.css. Similarly, the classpath:/static/js/app.js  will be mapped to /resources/js/app.js. You can use these static resources in jsp as follows.

<link rel="stylesheet" href="/resources/css/style.css">

<script type="text/javascript" src="/resources/js/app.js"></script>

### **Main class**

To bootstrap Spring MVC application using Spring Boot, create a main class annotated with @SpringBootApplication annotation as follows.

**MainApp.java**

**package** com.tutorial.springboot;

**import** org.springframework.boot.SpringApplication;

**import** org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication

**public** **class** **MainApp** {

**public** **static** **void** **main**(String[] args) {

SpringApplication.run(MainApp.class, args);

}

}

HashMap has an inner class called an Entry Class which holds the key and values

this inner class has four fields. key, value, next and hash.

**key** : It stores the key of an element and its final.

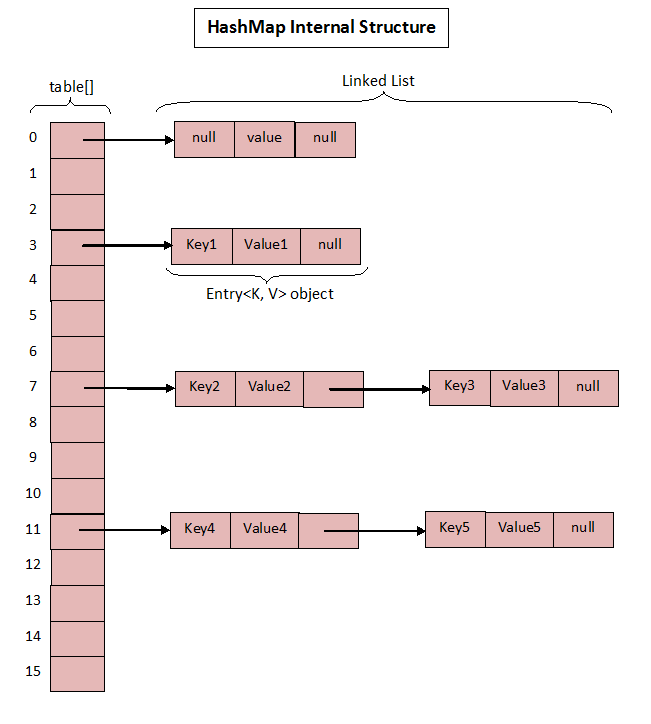
**value** : It holds the value of an element.

**next** : It holds the pointer to next key-value pair. **This attribute makes the key-value pairs stored as a linked list.**

**hash** : It holds the hashcode of the key.

These Entry objects are stored in an array called table[]. This array is initially of size 16

To summarize the whole HashMap structure, each key-value pair is stored in an object of Entry<K, V> class. This class has an attribute called next which holds the pointer to next key-value pair. This makes the key-value pairs stored as a linked list. All these Entry<K, V> objects are stored in an array called table[]. The below image best describes the HashMap structure.



The above image roughly shows how the HashMap stores its elements. Internally it uses an array of Entry<K, V> class called table[] to store the key-value pairs. But how HashMap allocates slot in table[] array to each of its key-value pair is very interesting. It doesn’t inserts the objects as you put them into HashMap i.e first element at index 0, second element at index 1 and so on. Instead it uses the hashcode of the key to decide the index for a particular key-value pair. It is called **Hashing**.

### What Is Hashing?

The whole HashMap data structure is based on the principle of **Hashing**. Hashing is nothing but the function or algorithm or method which when applied on any object/variable returns an unique integer value representing that object/variable. This unique integer value is called **hash code**. Hash function or simply hash said to be the best if it returns the same hash code each time it is called on the same object. Two objects can have same hash code.

Whenever you insert new key-value pair using put() method, HashMap blindly doesn’t allocate slot in the table[] array. Instead it calls hash function on the key. HashMap has its own hash function to calculate the hash code of the key. This function is implemented so that it overcomes poorly implemented hashCode() methods. Below is implementation code of hash().

w, let’s see how put() method works in detail.

### How put() method works?

public V put(K key, V value) {

if (key == null)

return putForNullKey(value);

int hash = hash(key);

int i = indexFor(hash, table.length);

for (Entry<K,V> e = table[i]; e != null; e = e.next) {

Object k;

if (e.hash == hash && ((k = e.key) == key || key.equals(k))) {

V oldValue = e.value;

e.value = value;

e.recordAccess(this);

return oldValue;

}

}

modCount++;

addEntry(hash, key, value, i);

return null;

}

Let’s see how this code works step by step.

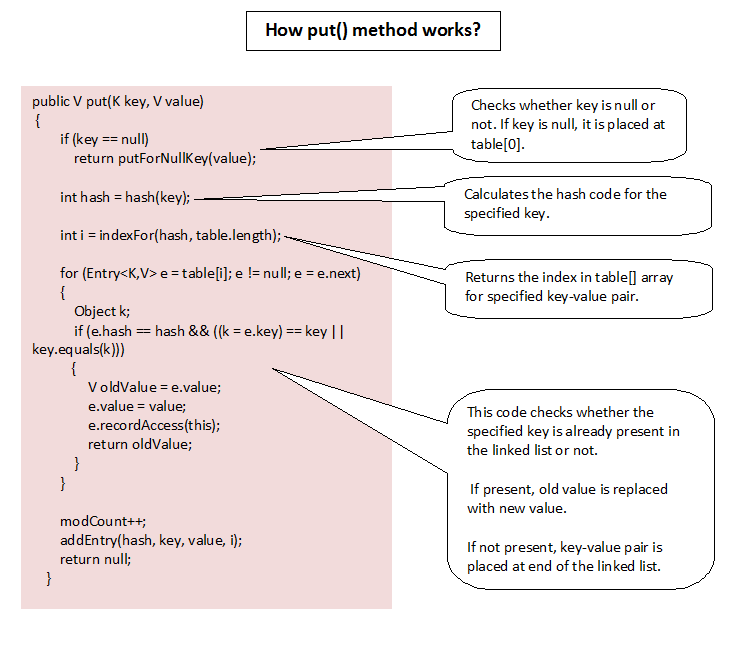
Step 1 : First checks whether the key is null or not. If the key is null, it calls putForNullKey() method. table[0] is always reserved for null key. Because, hash code of null is 0.

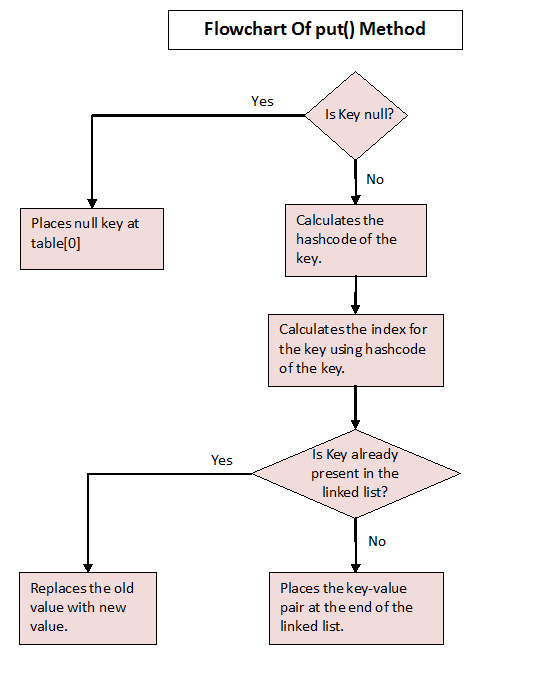
Step 2 : If the key is not null, then it calculates the hash code of the key by calling hash() method.

Step 3 : Calls indexFor() method by passing the hash code calculated in step 2 and length of the table[] array. This method returns index in table[] array for the specified key-value pair.

Step 4 : After getting the index, it checks all keys present in the linked list at that index ( or bucket). If the key is already present in the linked list, it replaces the old value with new value.

Step 5 : If the key is not present in the linked list, it appends the specified key-value pair at the end of the linked list.





### How get() method Works?

Let’s see how get() method has implemented.

public V get(Object key) {

if (key == null)

return getForNullKey();

int hash = hash(key.hashCode());

for (Entry<K , V> e = table[indexFor(hash, table.length)]; e != null; e = e.next) {

Object k;

if (e.hash == hash && ((k = e.key) == key || key.equals(k)))

return e.value;

}

return null;

}

Step 1 : First checks whether specified key is null or not. If the key is null, it calls getForNullKey() method.

Step 2 : If the key is not null, hash code of the specified key is calculated.

Step 3 : indexFor() method is used to find out the index of the specified key in the table[] array.

Step 4 : After getting index, it will iterate though linked list at that position and checks for the key using equals() method. If the key is found, it returns the value associated with it. otherwise returns null.

## Login Controller

* public String showLoginPage(ModelMap model): Mapped to the \login Get Method, this method shows the login page.
* @Autowired LoginService service: LoginService has the validation logic.
* showWelcomePage(ModelMap model, @RequestParam String name, @RequestParam String passwordMapped to the \login Post Method, this method validates the user id and password. Redirects to welcome page if login is successful.

package com.springboot.web.controller;

import org.springframework.beans.factory.annotation.Autowired;

import org.springframework.stereotype.Controller;

import org.springframework.ui.ModelMap;

import org.springframework.web.bind.annotation.RequestMapping;

import org.springframework.web.bind.annotation.RequestMethod;

import org.springframework.web.bind.annotation.RequestParam;

import org.springframework.web.bind.annotation.SessionAttributes;

import com.springboot.web.service.LoginService;

@Controller

@SessionAttributes("name")

public class LoginController {

@Autowired

LoginService service;

@RequestMapping(value="/login", method = RequestMethod.GET)

public String showLoginPage(ModelMap model){

return "login";

}

@RequestMapping(value="/login", method = RequestMethod.POST)

public String showWelcomePage(ModelMap model, @RequestParam String name, @RequestParam String password){

boolean isValidUser = service.validateUser(name, password);

if (!isValidUser) {

model.put("errorMessage", "Invalid Credentials");

return "login";

}

model.put("name", name);

model.put("password", password);

return "welcome";

}

}

## Login Service

Has the basic logic for authentication. Hardcoded business logic.

package com.springboot.web.service;

import org.springframework.stereotype.Component;

import org.springframework.stereotype.Service;

@Service

public class LoginService {

public boolean validateUser(String userid, String password) {

// test/dummy

return userid.equalsIgnoreCase("test")

&& password.equalsIgnoreCase("dummy");

}

}

## Login View - JSP

Simple login page with user id and password form fields. If an error message is populated into the model, ${errorMessage} will show the authentication failure error message.

<html>

<head>

<title>First Web Application</title>

</head>

<body>

<font color="red">${errorMessage}</font>

<form method="post">

Name : <input type="text" name="name" />

Password : <input type="password" name="password" />

<input type="submit" />

</form>

</body>

</html>

## Welcome View - JSP

The welcome page is shown on successful authentication. Shows the name of the login user and a link to manage your todos.

<html>

<head>

<title>First Web Application</title>

</head>

<body>

Welcome ${name}!! <a href="/list-todos">Click here</a> to manage your todo's.

</body>

</html>

## Lambda Example:

package com;

import java.util.ArrayList;

import java.util.List;

interface IArea {

void area();

}

/\*class Rectangle implements IArea

{

public void area()

{

System.out.println("Area of rectangle");

}

}\*/

public class LambdaDemo {

public static void main(String[] args) {

/\* IArea ia=new IArea()

{

public void area()

{

System.out.println("Area of rectangle");

}

};

ia.area();

\*/

IArea ia=() -> System.out.println("Area of rectangle");

ia.area();

List<String> pointList = new ArrayList();

pointList.add("1");

pointList.add("2");

pointList.forEach(p -> {

System.out.println(p);

//Do more work

}

);

}

}

HashMap<String, Integer> hash\_map = **new** HashMap<String, Integer>();

// Mapping int values to string keys

hash\_map.put("Geeks", 10);

hash\_map.put("4", 15);

hash\_map.put("Geeks", 20);

hash\_map.put("Welcomes", 25);

hash\_map.put("You", 30);

// Displaying the HashMap

System.***out***.println("Initial Mappings are: " + hash\_map);

// Using values() to get the set view of values

System.***out***.println("The collection is: " + hash\_map.values());

Collection <Integer> nList=hash\_map.values();

Iterator<Integer> itr=nList.iterator();

**while**(itr.hasNext())

{

Integer ii=(Integer)itr.next();

System.***out***.println(ii);

}

List <String> nameListWithA=**new** ArrayList<String>();

nameListWithA.add("Abir");

nameListWithA.add("Akash");

nameListWithA.add("Amar");

List <String> nameListWithB=**new** ArrayList<String>();

nameListWithB.add("Bikash");

nameListWithB.add("Badal");

nameListWithB.add("Bijoy");

HashMap<String, List <String>> hmap = **new** HashMap<String, List <String>>();

hmap.put("A", nameListWithA);

hmap.put("B", nameListWithB);

System.***out***.println("The map is: " + hmap.values());

// iterating over a map

**for**(Map.Entry<String, List<String>> listEntry : hmap.entrySet()){

System.***out***.println("Iterating list number - " );

// iterating over a list

**for**(String name : listEntry.getValue()){

System.***out***.println("Name - " + name);

}

}

//String pattern = "yyyy-MM-dd";

String pattern = "dd/MM/yyyy";

SimpleDateFormat simpleDateFormat = **new** SimpleDateFormat(pattern);

String date = simpleDateFormat.format(**new** Date());

System.***out***.println(date);

# Spring Cloud Zipkin and Sleuth Example:

[Zipkin](http://zipkin.io/) is very efficient tool for **distributed tracing** in [microservices](https://howtodoinjava.com/microservices/microservices-definition-principles-benefits/) ecosystem. Distributed tracing, in general, is latency measurement of each component in a distributed transaction where multiple microservices are invoked to serve a single business usecase.

What is Distributed Tracing?

Distributed Tracing is crucial for troubleshooting and understanding microservices. It is very useful when we need to track the request passing through multiple microservices. Distributed Tracing can be used to measure the performance of the microservices.

It is easy to identify which microservice is failed or having performance issues whenever there are multiple service calls within a single request.

Let’s say from our application, we have to call 4 different services/components for a transaction. Here with distributed tracing enabled, we can measure which component took how much time.

This is useful during debugging when lots of underlying systems are involved and the application becomes slow in any particular situation. In such case, we first need to identify see which underlying service is actually slow. Once the slow service is identified, we can work to fix that issue. Distributed tracing helps in identifying that slow component among in the ecosystem.

## Zipkin

Zipkin was originally developed at Twitter, based on a concept of a Google paper that described Google’s internally-built distributed app debugger – [dapper](http://research.google.com/pubs/pub36356.html). It manages both the collection and lookup of this data. To use Zipkin, applications are instrumented to report timing data to it.

If you are troubleshooting latency problems or errors in ecosystem, you can filter or sort all traces based on the application, length of trace, annotation, or timestamp. By analyzing these traces, you can decide which components are not performing as per expectations, and you can fix them.

Internally it has 4 modules –

1. **Collector** – Once any component sends the trace data arrives to Zipkin collector daemon, it is validated, stored, and indexed for lookups by the Zipkin collector.
2. **Storage** – This module store and index the lookup data in backend. [Cassandra](https://cassandra.apache.org/), [ElasticSearch](https://www.elastic.co/) and [MySQL](https://howtodoinjava.com/mysql/how-to-installuninstallexecute-mysql-as-windows-service/) are supported.
3. **Search** – This module provides a simple JSON API for finding and retrieving traces stored in backend. The primary consumer of this API is the Web UI.
4. **Web UI** – A very nice UI interface for viewing traces.

#### **How to install Zipkin**

Detailed installation steps can be found for different operating system including [Docker](https://howtodoinjava.com/cloud/docker-hello-world-example/) image at [quickstart page](http://zipkin.io/pages/quickstart.html). For windows installation, just download the latest Zipkin server from [maven repository](https://search.maven.org/remote_content?g=io.zipkin.java&a=zipkin-server&v=LATEST&c=exec) and run the [executable jar](https://howtodoinjava.com/maven/maven-shade-plugin-create-uberfat-jar-example/) file using below command.

|  |
| --- |
| java -jar zipkin-server-1.30.3-exec.jar |

Once Zipkin is started, we can see the Web UI at <http://localhost:9411/zipkin/>.

Above command will start the Zipkin server with default configuration. For advanced configuration, we can configure many other things like storage, collector listeners etc.

To **install Zipkin in spring boot application**, we need to add Zipkin starter dependency in spring boot project.

|  |
| --- |
| <dependency>      <groupId>org.springframework.cloud</groupId>      <artifactId>spring-cloud-starter-zipkin</artifactId>  </dependency> |

## Sleuth

[Sleuth](https://cloud.spring.io/spring-cloud-sleuth/) is a tool from Spring cloud family. It is used to generate the trace id, span id and add these information to the service calls in the headers and MDC, so that It can be used by tools like Zipkin and [ELK](https://howtodoinjava.com/microservices/elk-stack-tutorial-example/) etc. to store, index and process log files. As it is from spring cloud family, once added to the CLASSPATH, it automatically integrated to the common communication channels like –

* requests made with the [RestTemplate](https://howtodoinjava.com/spring/spring-restful/spring-restful-client-resttemplate-example/) etc.
* requests that pass through a [Netflix Zuul](https://howtodoinjava.com/spring/spring-cloud/spring-cloud-api-gateway-zuul/) microproxy
* HTTP headers received at [Spring MVC](https://howtodoinjava.com/spring-mvc-tutorial/) controllers
* requests over messaging technologies like Apache Kafka or RabbitMQ etc.

Using Sleuth is very easy. We just need to add it’s started pom in the spring boot project. It will add the Sleuth to project and so in its runtime.

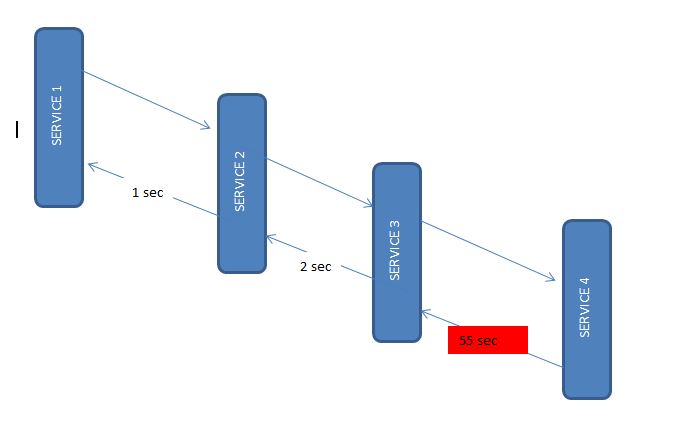
|  |
| --- |
| <dependency>      <groupId>org.springframework.cloud</groupId>      <artifactId>spring-cloud-starter-sleuth</artifactId>  </dependency> |

So far we have integrated Zipkin and Sleuth to microservices and ran Zipkin server. Let’s see how to utilize this setup.

## Zipkin and Sleuth Integration Example

For this demo, lets create 4 spring boot based microservices. They all will have both Zipkin and Sleuth starter dependencies. In each microservice, we will expose one endpoint and from the first service we will call second service, and from second service we will invoke third and so on using rest Template.

And as we have already mentioned, Sleuth works automatically with rest template so it would send this instrumented service call information to attached Zipkin server. Zipkin will then start the book keeping of latency calculation along with few other statistics like service call details.

Microservices Interactions

#### Create Microservice

All the four services will have the same configuration, only difference is the service invocation details where the endpoint changes. Let’s [create Spring boot applications](https://howtodoinjava.com/spring/spring-boot/spring-boot-tutorial-with-hello-world-example/) with Web, Rest Repository, Zipkin and Sleuth dependencies.

I have packaged those services inside a parent project so that those four services can be build together to save time. You can proceed with individual set up if you wish to. Also I have added useful windows scripts so start/stop all the services with a single command.

This is one sample rest controller which exposes one endpoint and also invokes one downstream service using rest template.

|  |
| --- |
| package com.example.zipkinservice1;    import org.apache.log4j.Logger;  import org.springframework.beans.factory.annotation.Autowired;  import org.springframework.boot.SpringApplication;  import org.springframework.boot.autoconfigure.SpringBootApplication;  import org.springframework.cloud.sleuth.sampler.AlwaysSampler;  import org.springframework.context.annotation.Bean;  import org.springframework.core.ParameterizedTypeReference;  import org.springframework.http.HttpMethod;  import org.springframework.web.bind.annotation.GetMapping;  import org.springframework.web.bind.annotation.RestController;  import org.springframework.web.client.RestTemplate;    @SpringBootApplication  public class ZipkinService1Application {        public static void main(String[] args) {          SpringApplication.run(ZipkinService1Application.class, args);      }  }    @RestController  class ZipkinController{        @Autowired      RestTemplate restTemplate;        @Bean      public RestTemplate getRestTemplate() {          return new RestTemplate();      }        @Bean      public AlwaysSampler alwaysSampler() {          return new AlwaysSampler();      }        private static final Logger LOG = Logger.getLogger(ZipkinController.class.getName());        @GetMapping(value="/zipkin")      public String zipkinService1()      {          LOG.info("Inside zipkinService 1..");             String response = (String) restTemplate.exchange("<http://localhost:8082/zipkin2>",                          HttpMethod.GET, null, new ParameterizedTypeReference<String>() {}).getBody();          return "Hi...";      }  } |

#### App Configurations

As all services will run in a single machine, so we need to run them in different ports. Also to identify in Zipkin, we need to give proper names. So configure application name and port information in application.properties file under resources folder.

|  |
| --- |
| server.port = 8081  spring.application.name = zipkin-server1 |

Similarly for other 3 services, we will use ports **8082**, **8083**, **8084** and name will also be like **zipkin-server2**, **zipkin-server3** and **zipkin-server4**.

Also we have intentionally introduced a delay in the last service so that we can view that in Zipkin.

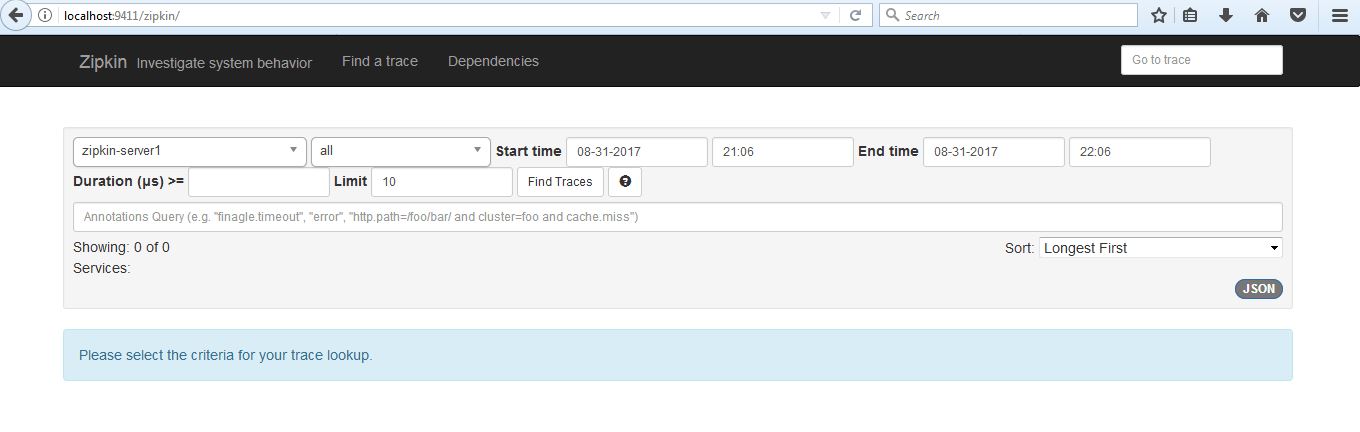
## Demo

Do a final maven build using command mvn clean install in microservices, start all the 4 applications along with the zipkin server.

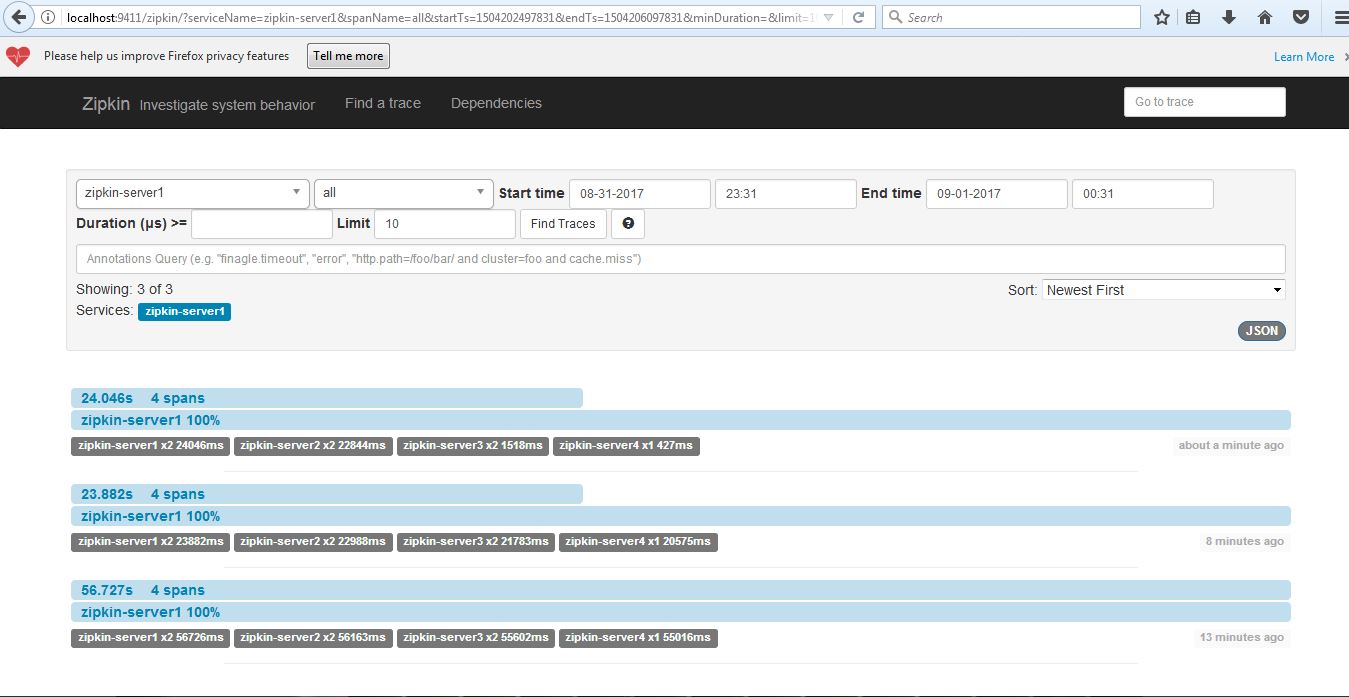
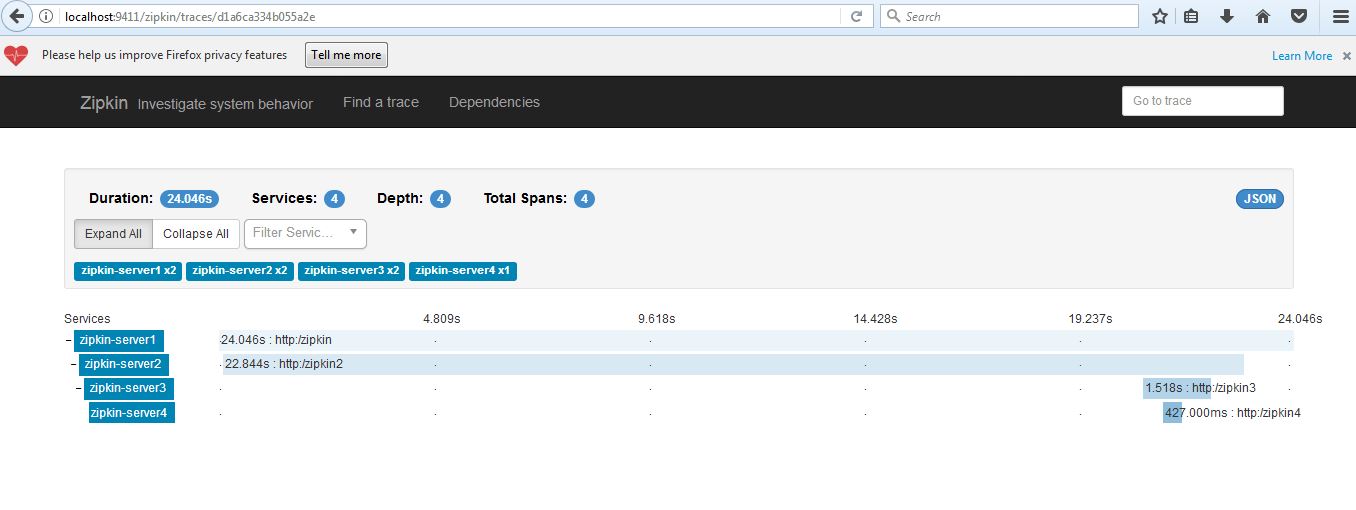
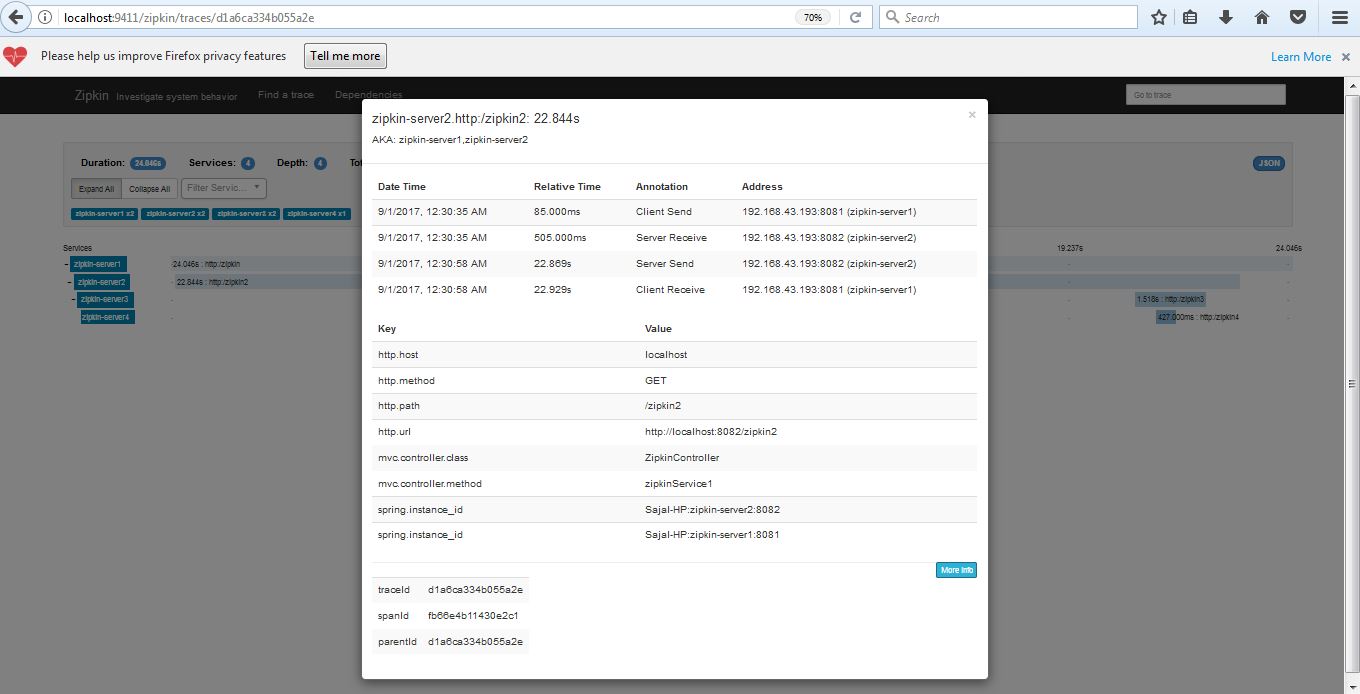
For quick start and stop, use the bat file Start-all.bat and Stop-all.bat.

Now test the first service endpoint couple of time from browser – <http://localhost:8081/zipkin>. Please be aware that there is an intentional delay in one of the above 4 services. So there will be delay is final response which is expected, just don’t give up.

After API call succeed, we can see the latency statistics at zipkin UI <http://localhost:9411/zipkin/>. Choose the first service in the first drop-down, and once click on **Find Traces** button.

Zipkin Home screen

You should see this type of UI where you can do performance analysis by looking at tracing data.

Find Traces UI[](https://howtodoinjava.com/wp-content/uploads/2017/08/7-3.jpg)One particular transaction overview[](https://howtodoinjava.com/wp-content/uploads/2017/08/8-3.jpg)Details of a particular service call statistics

## Summary

In this tutorial, we learned to use Zipkin to analyze latency in the service calls. Also we learned how Sleuth can help us creating the metadata and pass it to Zipkin.

I hope this information will be useful to you to get started with **distributed tracing using Zipkin and Sleuth**.