Advance Java

1. Java Servlet 1

2. Java Server Pages (JSP) 4

3. Spring (M.V.C) 6

Spring Tiles 10

4. JQuery 13

5. AJAX(Asynchronous JavaScript and XML) 14

6. JSON 15

7. WebServices 16

8. Hibernate 23

9. Utilities 26

Reading from Properties File 26

Logging 27

* **Java Servlet**

A **servlet** is a [Java programming language](http://en.wikipedia.org/wiki/Java_programming_language) [class](http://en.wikipedia.org/wiki/Class_%28computer_programming%29) used to extend the capabilities of [servers](http://en.wikipedia.org/wiki/Server) that can be accessed by a host application via a [request-response](http://en.wikipedia.org/wiki/Request-response) [programming model](http://en.wikipedia.org/wiki/Programming_model). Although servlets can respond to any type of request, they are commonly used to extend the applications hosted by [Web servers](http://en.wikipedia.org/wiki/Web_server). Thus, it can be thought of as a [Java Applet](http://en.wikipedia.org/wiki/Java_Applet) that runs on a [server](http://en.wikipedia.org/wiki/Server_%28computing%29) instead of a [browser](http://en.wikipedia.org/wiki/Web_browser).

A **Servlet** is a Java-based server-side web technology. As the name implies, it serves a client request and receives a response from the server. Technically speaking, a **Servlet** is a [Java class](http://en.wikipedia.org/wiki/Java_class) in [Java EE](http://en.wikipedia.org/wiki/Java_EE) that conforms to the **Java Servlet API**, a protocol by which a Java class may respond to requests. They are not tied to a specific client-server protocol, but are most often used with the [HTTP](http://en.wikipedia.org/wiki/HTTP) protocol. Therefore, the word "Servlet" is often used in the meaning of "HTTP Servlet".Thus, a [software developer](http://en.wikipedia.org/wiki/Software_developer) may use a servlet to add [dynamic content](http://en.wikipedia.org/wiki/Dynamic_web_page) to a [Web server](http://en.wikipedia.org/wiki/Web_server) using the [Java platform](http://en.wikipedia.org/wiki/Java_platform). The generated content is commonly [HTML](http://en.wikipedia.org/wiki/HTML), but may be other data such as [XML](http://en.wikipedia.org/wiki/XML). Servlets are the [Java](http://en.wikipedia.org/wiki/Java_%28software_platform%29) counterpart to non-Java dynamic Web content technologies such as PHP and [ASP.NET](http://en.wikipedia.org/wiki/Active_Server_Pages). Servlets can maintain [state](http://en.wikipedia.org/wiki/State_%28computer_science%29) in [session](http://en.wikipedia.org/wiki/Session_%28computer_science%29) variables across many server transactions by using [HTTP cookies](http://en.wikipedia.org/wiki/HTTP_cookie), or [URL rewriting](http://en.wikipedia.org/wiki/URL_rewriting).

To deploy and run a Servlet, a [Web container](http://en.wikipedia.org/wiki/Web_container) must be used. A Web container (also known as a Servlet container) is essentially the component of a Web server that interacts with the servlets. The Web container is responsible for managing the lifecycle of servlets, mapping a URL to a particular servlet and ensuring that the URL requester has the correct access rights.

The servlet [API](http://en.wikipedia.org/wiki/Application_programming_interface), contained in the [Java package](http://en.wikipedia.org/wiki/Java_package) hierarchy [javax.servlet](http://download.oracle.com/javaee/6/api/javax/servlet/package-summary.html), defines the expected interactions of the [Web container](http://en.wikipedia.org/wiki/Web_container) and a servlet

Life cycle of a servlet

During initialization stage of the Servlet life cycle, the web container initializes the servlet instance by calling the init() method. The container passes an object implementing the [ServletConfig](http://docs.oracle.com/javaee/1.3/api/javax/servlet/ServletConfig.html) interface via the [init](http://docs.oracle.com/javaee/1.3/api/javax/servlet/Servlet.html)() method. This configuration object allows the servlet to access name-value initialization parameters from the web application.

After initialization, the servlet can service client requests. Each [request](http://en.wikipedia.org/wiki/HTTP_request) is serviced in its own separate thread. The Web container calls the service() method of the servlet for every request. The service() method determines the kind of request being made and dispatches it to an appropriate method to handle the request. The developer of the servlet must provide an implementation for these methods. If a request for a method that is not implemented by the servlet is made, the method of the parent class is called, typically resulting in an error being returned to the requester.

Finally, the Web container calls the destroy() method that takes the servlet out of service. The destroy() method, like init(), is called only once in the lifecycle of a servlet.

**Servlets are most often used to**

process or store data that was submitted from an HTML form

provide dynamic content such as the results of a database query

manage state information that does not exist in the stateless HTTP protocol, such as filling the articles into the shopping cart of the appropriate customer.

**Request and Reponse**

Request: An object of ServletRequest is used to provide the client request information to a servlet such as context type, content length, parameter names and values, header information, attributes etc.

Response: An object of ServletResponse is used to send response to client. It is used to send output to client, redirect the page, add cookie.

**Session Tracking**

Session is simply means a particular interval of time. Session tracking is a vay to maintain state of an user. Http protocol is a stateless protocol. Each time user requests to the server, server treats the request as the new request. So we need to maintain the state of an user to recognize to particular user.

**Cookie**

Cookie is a mechanism to maintain the state of an user. A cookie is a small piece of information that is persisted between the multiple client requests. A cookie has a name, a single value , and optional attributes such as a comment, path and domain qualifiers, a maximum age, and a version number.

**Filter**

A filter is an object that is used to perform filtering tasks such as conversion, log maintain, compression, encryption and decryption, input validation etc. A filter is invoked at the preprocessing and post procession of a request. It is pluggable i.e. its entry is defined in the web.xml file, if we remove the entry of filter from the web.xml file, filter will be removed automatically and we don’t need to change the servlet. So it will be easier to maintain the web a**pplication.**

**Servlets 3**

The @WebServlet annotation is used to declare a servlet. The annotated class must extend the javax.servlet.http.HttpServlet class.

*@WebServlet(*

*attribute1=value1,*

*attribute2=value2,*

*...*

*)*

*public class TheServlet extends javax.servlet.http.HttpServlet {*

*// servlet code...*

*}*

The @WebFilter annotation is used to declare a filter in a web application. The annotated class must extend the javax.servlet.Filter interface.

*@WebFilter(*

*attribute1=value1,*

*attribute2=value2,*

*...*

*)*

*public class TheFilter implements javax.servlet.Filter {*

*// implements Filter's methods*

*}*

The following example registers a filter for the URL pattern /admin:

*@WebFilter("/admin")*

*public class MyFilter implements Filter {*

*// implements Filter's methods here...*

*}*

Apply a filter for all URLs:

*@WebFilter("/\*")*

*public class MyFilter implements Filter {*

*// implements Filter's methods here...*

*}*

Register a filter for a specific servlet:

*@WebFilter(servletNames = "MyOwnServlet")*

*public class MyFilter implements Filter {*

*// implements Filter's methods here...*

*}*

Related Course: Servlets and JSPs: Creating Web Applications With Java

Register a filter for multiple servlets:

*@WebFilter(servletNames = {"MyOwnServlet", "UploadServlet"})*

*public class MyFilter implements Filter {*

*// implements Filter's methods here...*

*}*

* **Java Server Pages (JSP)**

JSP technology is used to create web application just like servlet technology. It can be thought of as an extension to servlet because it provides more functionality than servlet.

A JSP page contains HTML code and JSP tags. The jsp pages are easier to maintain than servlet because we can separate designing and development. It provides some additional features such as Expression Language, Custom Tag etc.

**JSP Implicit Obects**

Implicit objects are set of pre-defined Object available on jsp page. There are 9 implicit objects available for the JSP.

Following are them:

i.out

ii.request

iii.response

iv.config

v.application

vi.session

vii.pageContext

viii.page

ix.exception

**JSP Directives**

The directives are messages that tells the web container how to translate a JSP page into corresponding servlet. There are three types of directives:

i.Page

ii.include

iii.taglib

**Page Directive**

The page directive defines attributes that apply to an entire JSP page.

Attributes of JSP page directive

i.import

ii.contentType

iii.extends

iv.info

v.buffer

vi.language

vii.isELIgnored

viii.isThreadSafe

ix.autoFlush

x.session

xi.pageEncoding

xii.errorPage

xiii.isErrorPage

**Scripting Elements**

The scripting elements provides the ability to insert java code inside the jsp. There are three types of scripting elements:

i.scriplet tag:

A scriplet tag is used to execute java source code in JSP. Syntax as follows:

*<% java source code %>*

ii.expression tag:

The code placed within expression tag is written to the output stream of the response. So you need not write out.print() to write data. It is mainly used to print the values of variable or method. Eg, *<%= statement %>*

iii.declaration tag:

The JSP declaration tag is used declare fields and methods. The code written inside the jsp declaration tag is placed outside the service() method of auto generated servlet. So it doesn’t get memory at each request. Eg, *<%! Statement %>*

**JSP Action Tag**

JSP action tag is used to perform some specific tasks. The action tags basically are used to control the flow between pages and to use Java Bean. Most use jsp action tags are as follows:

i.jsp:forward

The jsp:forward action tag is used to forward the request to another resource it may be jsp,html or another resource. Eg, *<jsp:forward page=”nameofpage”>*

ii.jsp:include

The jsp:include action tag is used to include the content of another resource it may be jsp,html or servlet. The jsp include action tag includes the resource at request time so it is better for dynamic pages because there might be changes in future. Eg, *<jsp:include page=”page”/>*

iii.jsp:useBean

The jsp:useBean action tag is used to locate or instantiate a bean class. If bean object of the Bean class is already created, it doesn’t create the bean depending on the scope. But if

object of bean is not created, it instantiates the bean. Eg,

*<jsp:useBean id=”instanceName” scope=”page|request|session|application” class=”packageName.className”/>*

iv.jspSetProperty

v.jsp:getProperty

vi.jsp:param

vii.jsp:plugin

viii.jsp:fallback

* **Spring (M.V.C)**

Spring is a lightweight framework. It can be thought of as a framework of frameworks because it provides support to various frameworks such as Struts, Hibernate, Tapestry, EJB, JSF etc.The framework, in broader sense, can be defined as a structure where we find solution of the various technical problems.

**Advantages of Spring Framework**

1) Lightweight:

Spring framework is lightweight because of its POJO implementation. The Spring Framework doesn't force the programmer to inherit any class or implement any interface. That is why it is said non-invasive.

2) Easy to develop JavaEE application:

The Dependency Injection feature of Spring Framework and it support to various frameworks makes the easy development of JavaEE application.

3) Easy to test:

The Dependency Injection makes easier to test the application. The EJB or Struts application require server to run the application but Spring framework doesn't require server.

4) Loose Coupling:

The Spring applications are loosely coupled because of dependency injection.

5) Powerful abstraction:

It provides powerful abstraction to JavaEE specifications such as JMS, JDBC, JPA and JTA.

6) Declarative support:

It provides declarative support for caching, validation, transactions and formatting.

**Requirement for spring**

Jars:

1.commons-logging-1.1.1.jar

2.jstl-1.2

3.spring-asm-3.0.3.RELEASE

4.spring-beans-3.0.3.RELEASE

5.spring-context-3.0.3.RELEASE

6.spring-core-3.0.3.RELEASE

7.spring-expression-3.0.3.RELEASE

8.spring-web-3.0.3.RELEASE

9.spring-webmvc-3.0.3.RELEASE

10.serverlet-api.jar

mvc-dispatcher-servlet.xml (Contains information about spring related configuration information) -> this should be inside WEB-INF folder

Eg,

*<?xml version="1.0" encoding="UTF-8"?>*

*<beans xmlns="http://www.springframework.org/schema/beans"*

*xmlns:context="http://www.springframework.org/schema/context"*

*xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"*

*xsi:schemaLocation="*

[*http://www.springframework.org/schema/beans*](http://www.springframework.org/schema/beans)

[*http://www.springframework.org/schema/beans/spring-beans-3.0.xsd*](http://www.springframework.org/schema/beans/spring-beans-3.0.xsd)

[*http://www.springframework.org/schema/context*](http://www.springframework.org/schema/context)

[*http://www.springframework.org/schema/context/spring-context-3.0.xsd*](http://www.springframework.org/schema/context/spring-context-3.0.xsd)*">*

*<context:component-scan base-package="com.alu.demo.controller" />*

*<bean*

*class="org.springframework.web.servlet.view.InternalResourceViewResolver">*

*<property name="prefix">*

*<value>/WEB-INF/jsp/</value>*

*</property>*

*<property name="suffix">*

*<value>.jsp</value>*

*</property>*

*</bean>*

*</beans>*

Following entry should be there in web.xml

*<servlet>*

*<servlet-name>mvc-dispatcher</servlet-name>*

*<servlet-class>*

*org.springframework.web.servlet.DispatcherServlet*

*</servlet-class>*

*<load-on-startup>1</load-on-startup>*

*</servlet>*

*<servlet-mapping>*

*<servlet-name>mvc-dispatcher</servlet-name>*

*<url-pattern>\*.htm</url-pattern>*

*</servlet-mapping>*

*<context-param>*

*<param-name>contextConfigLocation</param-name>*

*<param-value>/WEB-INF/mvc-dispatcher-servlet.xml</param-value>*

*</context-param>*

*<listener>*

*<listener-class>*

*org.springframework.web.context.ContextLoaderListener*

*</listener-class>*

*</listener>*

**Now we are all set for spring**

Userfull annotations:

*@Controller* -> To create controller class

*@RequestMapping* -> To create request URL

*@Service* -> To create service class

*@Inject* / @Autowire-> To initialize service class from controller

Handling Request,Response,Session

*@Controller*

*public class LoginController {*

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

*public String loginCheck(HttpServletRequest request, Model model, HttpServletResponse response){*

*return "success";*

*}*

*}*

-> Using request object we can able to get the request parameter values

-> Using model we can able to pass the data to view (Eg, jsp)

-> Using response we can able to work with servlet response

**JSTL Language for VIEW Control**

Required following tag lib

<%@ taglib prefix='c' uri='http://java.sun.com/jstl/core\_rt'%>

if else condition

<c:if test="${condition}">

</c:if>

Loop

<c:forEach items="${products}" var="product">

${product.customerName}

</c:forEach>

here, products is list of object and product is object of current element of loop

product.customerName is property of product class.

C:set

<c:set var="salary" scope="session" value="${2000\*2}"/>

C:Choose

<c:choose>

<c:when test="${salary <= 0}">

Salary is very low to survive.

</c:when>

<c:when test="${salary > 1000}">

Salary is very good.

</c:when>

<c:otherwise>

No comment sir...

</c:otherwise>

</c:choose>

NOTE: For Working with session,cookie follow same way as of Servlets or JSP

Creating service.

Create an interface

Create a service class that should implement the service interface

Use @Service annotation to create service class

Use @Inject to initialize the service class

For Eg,

*public interface IEmployeeService{*

*public void addEmployee(Employee obj);*

*}*

Now Create Service Class

package com.demo.dao.ora;

*@Service*

*public class EmplyeeServiceOracle implements IEmployeeService{*

*public void addEmployee(Employee obj){*

*//Perform the add operation*

*}*

*}*

Now Access this service class from controller

@Controller

public class EmployeeController{

@Inject IEmployeeService employeeService; //This will initialize the EmployeeService while loading this controller

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

public String addEmployee(HttpServletRequest request, Model model, HttpServletResponse response){

Employee e=new Employee();

employeeService.addEmployee(e); //Call the addEmployee method

return "success";

}

}

Add following on dispatcher-servlet.xml to scan the service class

*<context:component-scan base-package="com.demo.dao.mysql" /> //This will load the service classes inside com.demo.dao package*

**Spring Tiles**

Tiles is framework to manage the layout. Tiles framework is provided by Apache and Spring provides integration support with apache tiles framework. So we can simply manage the layout of the spring mvc application by the help of spring tiles support.

Advantage of tiles support

i.Reusability

ii.Centralized control

iii. Easy to change the layout

**Requirement**

Tiles jar

Update following in spring configuration file

*<bean id="viewResolver" class="org.springframework.web.servlet.view.tiles3.TilesViewResolver"/>*

*<bean id="tilesConfigurer" class="org.springframework.web.servlet.view.tiles3.TilesConfigurer">*

*<property name="definitions">*

*<list>*

*<value>/WEB-INF/layouts/layouts.xml</value>*

*<value>/WEB-INF/layouts/views.xml</value>*

*</list>*

*</property>*

*</bean>*

Layouts.xml defines the main layout with default page

Views.xml defines the views to be generated by spring controller

**Tiles Tag:**

**<tiles:insertAttribute name=*"menu"* /> to create place holder**

<put-attribute name=*"menu"* value=*"/WEB-INF/views/template/menu.jsp""* /> //To add page in place holder

SitesTemplate.jsp (Main template file)

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

pageEncoding=*"ISO-8859-1"*%>

<%@ taglib uri=*"http://tiles.apache.org/tags-tiles"* prefix=*"tiles"*%>

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "<http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd>">

<html xmlns=*"http://www.w3.org/1999/xhtml"*>

<head>

<meta http-equiv=*"Content-Type"* content=*"text/html; charset=UTF-8"*>

<title>Spring MVC - Tiles Integration</title>

<link rel=*"stylesheet"* href=*"resources/css/screen.css"* type=*"text/css"* media=*"screen, projection"*></link>

<link rel=*"stylesheet"* href=*"resources/css/print.css"* type=*"text/css"* media=*"print"*></link>

<!--[if IE]>

<link rel="stylesheet" href="resources/css/ie.css" type="text/css" media="screen, projection">

<![endif]-->

<style>

**body**{ margin-top:*20px*; margin-bottom:*20px*; background-color:*#DFDFDF*;}

</style>

</head>

<body>

<div class=*"container"* style="border: *#C1C1C1 solid 1px*; border-radius:*10px*;">

<!-- Header -->

**<tiles:insertAttribute name=*"header"* />**

<!-- Menu Page -->

<div class=*"span-5 border"* style="height:*400px*;background-color:*#FCFCFC*;">

**<tiles:insertAttribute name=*"menu"* />**

</div>

<!-- Body Page -->

<div class=*"span-19 last"*>

**<tiles:insertAttribute name=*"body"* />**

</div>

<!-- Footer Page -->

**<tiles:insertAttribute name=*"footer"* />**

</div>

</body>

</html>

Layouts.xml

<?xml version=*"1.0"* encoding=*"UTF-8"*?>

<!DOCTYPE tiles-definitions PUBLIC

"-//Apache Software Foundation//DTD Tiles Configuration 3.0//EN"

"<http://tiles.apache.org/dtds/tiles-config_3_0.dtd>">

<tiles-definitions>

<definition name=*"DefaultTemplate"* template=*"/WEB-INF/views/template/SiteTemplate.jsp"*>

<put-attribute name=*"title"* value=*"Home"* />

<put-attribute name=*"header"* value=*"/WEB-INF/views/template/header.jsp"* />

<put-attribute name=*"menu"* value=*"/WEB-INF/views/template/menu.jsp"* />

<put-attribute name=*"body"* value=*""* />

<put-attribute name=*"footer"* value=*"/WEB-INF/views/template/footer.jsp"* />

</definition>

</tiles-definitions>

Views.xml

<?xml version=*"1.0"* encoding=*"UTF-8"*?>

<!DOCTYPE tiles-definitions PUBLIC

"-//Apache Software Foundation//DTD Tiles Configuration 3.0//EN"

"<http://tiles.apache.org/dtds/tiles-config_3_0.dtd>">

<tiles-definitions>

<definition name=*"index"* extends=*"DefaultTemplate"*>

**<put-attribute name=*"body"* value=*"/WEB-INF/views/index.jsp"* />**

</definition>

<definition name=*"personList"* extends=*"DefaultTemplate"*>

**<put-attribute name=*"body"* value=*"/WEB-INF/views/personList.jsp"* />**

</definition>

<definition name=*"footer1"* extends=*"DefaultTemplate"*>

**<put-attribute name=*"footer"* value=*"/WEB-INF/views/footer1.jsp"* />**

</definition>

<definition name=*"header1"* extends=*"DefaultTemplate"*>

**<put-attribute name=*"header"* value=*"/WEB-INF/views/abc.jsp"* />**

</definition>

</tiles-definitions>

Now if controller returns “index” it will add the index.jsp to body or if controller returns the personList it will add the personList.jsp to body.

Example on PC

* **JQuery**

JQuery is java script library, Used to simplify the client side scripting of HTML.Its free and open source. It is the most popular java scripting library used today. jQuery's syntax is designed to make it easier to navigate a document, select DOM elements, create animations, handle events, and develop Ajax applications.

**How to integrated jquery in html**

Include the jquery.js file with the help of script src tag

For eg,

*<html>*

*<head>*

*<script src=”jquery-<version>.js”>*

*</script> <!—include the jquery --- >*

*<script>*

***function*** *hide(){*

*$('#abc').hide();*

*}*

***function*** *show(){*

*$("#mytext").val("Give you name");*

*$('#abc').show();*

*}*

*</script>*

*</head>*

*<body>*

*<div id=”abc”>hello</div>*

*<a href=”#” onclick=”hide()”>hide</a>*

*<a href=”#” onclick=”show()”>show</a>*

*<input type=”text” id=”mytext”>*

*</body>*

*</html>*

Explore more on : <https://jquery.com/>

* **AJAX(Asynchronous JavaScript and XML)**

Ajax is a set of web development techniques using many web technologies on the client-side to create asynchronous Web applications. With Ajax, web applications can send data to and retrieve from a server asynchronously (in the background) without interfering with the display and behavior of the existing page. By decoupling the data interchange layer from the presentation layer, Ajax allows for web pages, and by extension web applications, to change content dynamically without the need to reload the entire page. In practice, modern implementations commonly substitute JSON for XML due to the advantages of being native to JavaScript.

JQuery Ajax

Used to implement the Ajax call using jquery $.ajax function. Eg,

$.ajax({

url : 'ajaxdata.htm',

data: {name:"ram"},

success : **function**(data) {

$('#result').html(data);

},

error: **function**(e){

console.log("error :"+e);

}

});

Here,

url -> url of the page to call

data-> parameter to pass to page

success -> perform required action on successful call

error: perform required action on error

ajaxdata.htm

@Controller

**public** **class** AjaxController {

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

**public** String getAjaxPage(HttpServletRequest req,Model model,HttpServletResponse res){

**return** "ajaxdemo";

}

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

**public** @ResponseBody

String getTime(HttpServletRequest req) {

System.*out*.println("Data:"+req.getParameter("name"));

**return** **new** Date().toString();

}

}

* **JSON**

JSON (JavaScript Object Notation) is a minimal, readable format for structuring data. It is used primarily to transmit data between a server and web application, as an alternative to XML.

JSON stands for **J**ava**S**cript **O**bject **N**otation

JSON is a lightweight data-interchange format

JSON is language independent

JSON is "self-describing" and easy to understand

Eg,

Simple JSON

{

"Name": "Ram",

"Age": 20,

"class": "BEIT"

}

JSON Array

{

"Name": "Ram",

"Age": 20,

"class": "BEIT",

"course": [

{

"name": "java",

"credits": 3

},

{

"name": "Oracle",

"credits": 2

}

]

}

For More:

<http://developers.squarespace.com/what-is-json/>

<http://www.w3schools.com/json/>

* **WebServices**

**Web services** are client and server applications that communicate over the World Wide Web’s (WWW) HyperText Transfer Protocol (HTTP). As described by the World Wide Web Consortium (W3C), web services provide a standard means of interoperating between software applications running on a variety of platforms and frameworks. Web services are characterized by their great interoperability and extensibility, as well as their machine-processable descriptions, thanks to the use of XML. Web services can be combined in a loosely coupled way to achieve complex operations. Programs providing simple services can interact with each other to deliver sophisticated added-value services.

**Types of WebService**

Big

Restfull

BIG:

 Big web services use XML messages that follow the Simple Object Access Protocol (SOAP) standard, an XML language defining a message architecture and message formats. Such systems often contain a machine-readable description of the operations offered by the service, written in the Web Services Description Language (WSDL), an XML language for defining interfaces syntactically.

The SOAP message format and the WSDL interface definition language have gained widespread adoption. Many development tools, such as NetBeans IDE, can reduce the complexity of developing web service applications.

REST Full (Representational State Transfer)

REST is well suited for basic, ad hoc integrations scenarios, RESTFul web services, often better integrated with HTTP than SOAP-based services are. Do not require XML messages or WSDL service-API definitions.

Ref: <http://docs.oracle.com/javaee/6/tutorial/doc/giqsx.html>

Some frameworks for Restfull webservice using java

i.Jersey : <https://jersey.java.net/>

ii.Dropwizard : <http://www.dropwizard.io/0.9.2/docs/>

iii.Spring: <https://spring.io/guides/gs/rest-service/>

**Spring Restfull Webservices**

Building the webservice that will accept HTTP Get request at:

<http://localhost:8080/greeting>

and respond with a [JSON](https://spring.io/understanding/JSON) representation of a greeting:

{"id":1,"content":"Hello, World!"}

Requirement:

JDK 1.8 or later

Eclipse Editor

Maven build tool

Lets Begin

**Building project with Maven**

First you set up a basic build script. You can use any build system you like when building apps with Spring, but the code you need to work with [Maven](https://maven.apache.org/) is included here. If you’re not familiar with Maven, refer to [Building Java Projects with Maven](https://spring.io/guides/gs/maven).

**Create the directory structure**

In a project directory of your choosing, create the following subdirectory structure; for example, with mkdir -p src/main/java/hello :

└── src

└── main

└── java

└── hello

pom.xml

<?xml version="1.0" encoding="UTF-8"?>

<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 <http://maven.apache.org/xsd/maven-4.0.0.xsd>">

<modelVersion>4.0.0</modelVersion>

<groupId>org.springframework</groupId>

<artifactId>gs-rest-service</artifactId>

<version>0.1.0</version>

<parent>

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

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

<version>1.3.3.RELEASE</version>

</parent>

<dependencies>

<dependency>

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

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

</dependency>

</dependencies>

<properties>

<java.version>1.8</java.version>

</properties>

<build>

<plugins>

<plugin>

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

<artifactId>spring-boot-maven-plugin</artifactId>

</plugin>

</plugins>

</build>

<repositories>

<repository>

<id>spring-releases</id>

<url>https://repo.spring.io/libs-release</url>

</repository>

</repositories>

<pluginRepositories>

<pluginRepository>

<id>spring-releases</id>

<url>https://repo.spring.io/libs-release</url>

</pluginRepository>

</pluginRepositories>

</project>

The [Spring Boot Maven HYPERLINK "https://github.com/spring-projects/spring-boot/tree/master/spring-boot-tools/spring-boot-maven-plugin" HYPERLINK "https://github.com/spring-projects/spring-boot/tree/master/spring-boot-tools/spring-boot-maven-plugin" HYPERLINK "https://github.com/spring-projects/spring-boot/tree/master/spring-boot-tools/spring-boot-maven-plugin"plugin](https://github.com/spring-projects/spring-boot/tree/master/spring-boot-tools/spring-boot-maven-plugin) provides many convenient features:

It collects all the jars on the classpath and builds a single, runnable "über-jar", which makes it more convenient to execute and transport your service.

It searches for the public static void main() method to flag as a runnable class.

It provides a built-in dependency resolver that sets the version number to match [Spring Boot dependencies](https://github.com/spring-projects/spring-boot/blob/master/spring-boot-dependencies/pom.xml). You can override any version you wish, but it will default to Boot’s chosen set of versions.

Create a resource representation class

Now that you’ve set up the project and build system, you can create your web service.

Begin the process by thinking about service interactions.

The service will handle GET requests for /greeting, optionally with a name parameter in the query string. The GET request should return a 200 OK response with JSON in the body that represents a greeting. It should look something like this:

{

"id": 1,

"content": "Hello, World!"

}

The id field is a unique identifier for the greeting, and content is the textual representation of the greeting.

To model the greeting representation, you create a resource representation class. Provide a plain old java object with fields, constructors, and accessors for the id and content data:

src/main/java/hello/Greeting.java

package hello;

public class Greeting {

private final long id;

private final String content;

public Greeting(long id, String content) {

this.id = id;

this.content = content;

}

public long getId() {

return id;

}

public String getContent() {

return content;

}

}

|  |  |
| --- | --- |
|  | As you see in steps below, Spring uses the [Jackson JSON](http://wiki.fasterxml.com/JacksonHome) library to automatically marshal instances of type Greeting into JSON. |

Next you create the resource controller that will serve these greetings.

Create a resource controller

In Spring’s approach to building RESTful web services, HTTP requests are handled by a controller. These components are easily identified by the [@ HYPERLINK "http://docs.spring.io/spring/docs/current/javadoc-api/org/springframework/web/bind/annotation/RestController.html" HYPERLINK "http://docs.spring.io/spring/docs/current/javadoc-api/org/springframework/web/bind/annotation/RestController.html" HYPERLINK "http://docs.spring.io/spring/docs/current/javadoc-api/org/springframework/web/bind/annotation/RestController.html"RestController](http://docs.spring.io/spring/docs/current/javadoc-api/org/springframework/web/bind/annotation/RestController.html) annotation, and the GreetingController below handles GET requests for /greeting by returning a new instance of the Greeting class:

src/main/java/hello/GreetingController.java

package hello;

import java.util.concurrent.atomic.AtomicLong;

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

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

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

@RestController

public class GreetingController {

private static final String template = "Hello, %s!";

private final AtomicLong counter = new AtomicLong();

@RequestMapping("/greeting")

public Greeting greeting(@RequestParam(value="name", defaultValue="World") String name) {

return new Greeting(counter.incrementAndGet(),

String.format(template, name));

}

}

This controller is concise and simple, but there’s plenty going on under the hood. Let’s break it down step by step.

The @RequestMapping annotation ensures that HTTP requests to /greeting are mapped to the greeting() method.

|  |  |
| --- | --- |
|  | The above example does not specify GET vs. PUT, POST, and so forth, because@RequestMapping maps all HTTP operations by default. Use @RequestMapping(method=GET)to narrow this mapping. |

@RequestParam binds the value of the query string parameter name into the name parameter of the greeting() method. This query string parameter is optional (required=false by default): if it is absent in the request, the defaultValue of "World" is used.

The implementation of the method body creates and returns a new Greeting object with idand content attributes based on the next value from the counter, and formats the givenname by using the greeting template.

A key difference between a traditional MVC controller and the RESTful web service controller above is the way that the HTTP response body is created. Rather than relying on a [view technology](https://spring.io/understanding/view-templates) to perform server-side rendering of the greeting data to HTML, this RESTful web service controller simply populates and returns a Greeting object. The object data will be written directly to the HTTP response as JSON.

This code uses Spring 4’s new [@ HYPERLINK "http://docs.spring.io/spring/docs/current/javadoc-api/org/springframework/web/bind/annotation/RestController.html" HYPERLINK "http://docs.spring.io/spring/docs/current/javadoc-api/org/springframework/web/bind/annotation/RestController.html" HYPERLINK "http://docs.spring.io/spring/docs/current/javadoc-api/org/springframework/web/bind/annotation/RestController.html"RestController](http://docs.spring.io/spring/docs/current/javadoc-api/org/springframework/web/bind/annotation/RestController.html) annotation, which marks the class as a controller where every method returns a domain object instead of a view. It’s shorthand for@Controller and @ResponseBody rolled together.

The Greeting object must be converted to JSON. Thanks to Spring’s HTTP message converter support, you don’t need to do this conversion manually. Because [Jackson 2](http://wiki.fasterxml.com/JacksonHome) is on the classpath, Spring’s [MappingJackson2HttpMessageConverter](http://docs.spring.io/spring/docs/current/javadoc-api/org/springframework/http/converter/json/MappingJackson2HttpMessageConverter.html) is automatically chosen to convert the Greeting instance to JSON.

Make the application executable

Although it is possible to package this service as a traditional [WAR](https://spring.io/understanding/WAR) file for deployment to an external application server, the simpler approach demonstrated below creates a standalone application. You package everything in a single, executable JAR file, driven by a good old Javamain() method. Along the way, you use Spring’s support for embedding the [Tomcat](https://spring.io/understanding/Tomcat) servlet container as the HTTP runtime, instead of deploying to an external instance.

src/main/java/hello/Application.java

package hello;

import org.springframework.boot.SpringApplication;

import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication

public class Application {

public static void main(String[] args) {

SpringApplication.run(Application.class, args);

}

}

@SpringBootApplication is a convenience annotation that adds all of the following:

@Configuration tags the class as a source of bean definitions for the application context.

@EnableAutoConfiguration tells Spring Boot to start adding beans based on classpath settings, other beans, and various property settings.

Normally you would add @EnableWebMvc for a Spring MVC app, but Spring Boot adds it automatically when it sees **spring-webmvc** on the classpath. This flags the application as a web application and activates key behaviors such as setting up a DispatcherServlet.

@ComponentScan tells Spring to look for other components, configurations, and services in the hello package, allowing it to find the GreetingController.

The main() method uses Spring Boot’s SpringApplication.run() method to launch an application. Did you notice that there wasn’t a single line of XML? No **web.xml** file either. This web application is 100% pure Java and you didn’t have to deal with configuring any plumbing or infrastructure.

Build an executable JAR

If you are using Gradle, you can run the application using ./gradlew bootRun.

You can build a single executable JAR file that contains all the necessary dependencies, classes, and resources. This makes it easy to ship, version, and deploy the service as an application throughout the development lifecycle, across different environments, and so forth.

./gradlew build

Then you can run the JAR file:

java -jar build/libs/gs-rest-service-0.1.0.jar

If you are using Maven, you can run the application using mvn spring-boot:run. Or you can build the JAR file with mvn clean package and run the JAR by typing:

java -jar target/gs-rest-service-0.1.0.jar

|  |  |
| --- | --- |
|  | The procedure above will create a runnable JAR. You can also opt to [build a classic WAR file](https://spring.io/guides/gs/convert-jar-to-war/) instead. |

Logging output is displayed. The service should be up and running within a few seconds.

Test the service

Now that the service is up, visit <http://localhost:8080/greeting>, where you see:

{"id":1,"content":"Hello, World!"}

Provide a name query string parameter with <http://localhost:8080/greeting?name=User>. Notice how the value of the content attribute changes from "Hello, World!" to "Hello User!":

{"id":2,"content":"Hello, User!"}

This change demonstrates that the @RequestParam arrangement in GreetingController is working as expected. The name parameter has been given a default value of "World", but can always be explicitly overridden through the query string.

Notice also how the id attribute has changed from 1 to 2. This proves that you are working against the same GreetingController instance across multiple requests, and that itscounter field is being incremented on each call as expected.

Summary

Congratulations! You’ve just developed a RESTful web service with Spring.

Want to write a new guide or contribute to an existing one? Check out our [contribution guidelines](https://github.com/spring-guides/getting-started-guides/wiki).

NOTE: Above Guide has been taken from <https://spring.io/guides/gs/rest-service/> only for study porpose

**Calling the Rest Webservice using Spring**

//Calling get method

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

***public*** *String callRest(HttpServletRequest req, Model model,*

*HttpServletResponse res) {*

*RestTemplate restTemplate =* ***new*** *RestTemplate();*

*ObjectMapper mapper =* ***new*** *ObjectMapper();*

*HttpHeaders headers =* ***new*** *HttpHeaders();*

*headers.setContentType(MediaType.APPLICATION\_JSON);*

*String response = restTemplate.getForObject("http://localhost:8080/greeting", String.****class****);*

*System.out.println("Data:" + response);*

***try*** *{*

*Greeting gr = mapper.readValue(response, Greeting.****class****); // Converts the Json String to Object*

*System.out.println("Content:" + gr.getContent());*

*}* ***catch*** *(Exception e) {*

*e.printStackTrace();*

*}*

*model.addAttribute("MESSAGE",response );*

***return*** *"restdemo";*

*}*

//Calling Post Method

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

***public*** *String callRestPost(HttpServletRequest req, Model model,*

*HttpServletResponse res) {*

***try*** *{*

*RestTemplate restTemplate =* ***new*** *RestTemplate();*

*ObjectMapper mapper =* ***new*** *ObjectMapper();*

*HttpHeaders headers =* ***new*** *HttpHeaders();*

*headers.setContentType(MediaType.APPLICATION\_JSON);*

*Greeting greetingObj =* ***new*** *Greeting();*

*greetingObj.setContent("testdata");*

*greetingObj.setId(1L);*

*String requestJson;*

*requestJson = mapper.writeValueAsString(greetingObj); // Converts the object to JSonString*

*HttpEntity<String> entity =* ***new*** *HttpEntity<String>(requestJson, headers);*

*String response = restTemplate.postForObject("http://localhost:8080/greeting1", entity, String.****class****);*

*System.out.println("Data:" + response);*

*Greeting gr = mapper.readValue(response, Greeting.****class****);*

*System.out.println("Content:" + gr.getContent());*

*model.addAttribute("MESSAGE",response );*

*}* ***catch*** *(Exception e) {*

*e.printStackTrace();*

*}*

***return*** *"restdemo";*

*}*

* **Hibernate**

Hibernate is an Object-Relational Mapping(ORM) solution for JAVA and it raised as an open source persistent framework created by Gavin King in 2001. It is a powerful, high performance Object-Relational Persistence and Query service for any Java Application.

Hibernate maps Java classes to database tables and from Java data types to SQL data types and relieve the developer from 95% of common data persistence related programming tasks.

Hibernate sits between traditional Java objects and database server to handle all the work in persisting those objects based on the appropriate O/R mechanisms and patterns.



**Requirement for Hibernate:**

Libraries:

Hibernate-core-3.6.3.jar

Hibernate-commons-annotations-3.2.0.Final.jar

Hibernate-jpa-2.0-api-1.0.0.Final.jar

Javassist-3.12.1.GA.jar

Slfj-api.1.6.1.jar

Jta-1.1.jar

Antlr-2.7.1.jar

Dom4j-1.6.1.jar

Hibernate configuration file(configuration settings related to database and other related parameters)

Hibernate.cfg.xml

<?xml version=*"1.0"* encoding=*"utf-8"*?>

<!DOCTYPE hibernate-configuration PUBLIC

"-//Hibernate/Hibernate Configuration DTD 3.0//EN"

"<http://hibernate.sourceforge.net/hibernate-configuration-3.0.dtd>">

<hibernate-configuration>

<session-factory>

<property name=*"hibernate.connection.driver\_class"*>oracle.jdbc.driver.OracleDriver</property>

<property name=*"hibernate.connection.url"*>jdbc:oracle:thin:@localhost:1521:xe</property>

<property name=*"hibernate.connection.username"*>user</property>

<property name=*"hibernate.connection.password"*>password</property>

<property name=*"hibernate.dialect"*>org.hibernate.dialect.Oracle10gDialect</property>

<property name=*"hibernate.default\_schema"*>test</property>

<property name=*"show\_sql"*>true</property>

<mapping class=*"com.demo.user.DBUser"*></mapping>

</session-factory>

</hibernate-configuration>

**List of Hibernate Dilect:**

|  |  |
| --- | --- |
| DB2 | org.hibernate.dialect.DB2Dialect |
| HSQLDB | org.hibernate.dialect.HSQLDialect |
| HypersonicSQL | org.hibernate.dialect.HSQLDialect |
| Informix | org.hibernate.dialect.InformixDialect |
| Ingres | org.hibernate.dialect.IngresDialect |
| Interbase | org.hibernate.dialect.InterbaseDialect |
| Microsoft SQL Server 2000 | org.hibernate.dialect.SQLServerDialect |
| Microsoft SQL Server 2005 | org.hibernate.dialect.SQLServer2005Dialect |
| Microsoft SQL Server 2008 | org.hibernate.dialect.SQLServer2008Dialect |
| MySQL | org.hibernate.dialect.MySQLDialect |
| Oracle (any version) | org.hibernate.dialect.OracleDialect |
| Oracle 11g | org.hibernate.dialect.Oracle10gDialect |
| Oracle 10g | org.hibernate.dialect.Oracle10gDialect |
| Oracle 9i | org.hibernate.dialect.Oracle9iDialect |
| PostgreSQL | org.hibernate.dialect.PostgreSQLDialect |
| Progress | org.hibernate.dialect.ProgressDialect |
| SAP DB | org.hibernate.dialect.SAPDBDialect |
| Sybase | org.hibernate.dialect.SybaseDialect |
| Sybase Anywhere | org.hibernate.dialect.SybaseAnywhereDialect |

**Creating Session.**

SessionFactory factory=new Configuration().configure().buildSessionFactory();

**Create a transaction class:**

Session session = factory.openSession();

**To insert data in table:**

Session.save(Object);

**To fetch data from table:**

session.createQuery("FROM Employee").list();

**To Update data:**

Session.update(Object)

**To Delete Data**

Session.delete(Object)

**To create transaction:**

Transaction tx=session.beginTransaction()

tx.commit()

**Usefull annotation**

@Entity -> To create entity class

@Table -> To specify table name for entity class

@Id -> To mention about the primary key of table

@Column -> To specify about the column name (to map with java property)

* **Utilities**

**Reading from Properties File**

Properties files are used to load the data from configuration file.

Sample properties file

dbinfo.properties

*driver.class=com.mysql.jdbc.Driver*

*driver.url=*

*db.user=root*

*db.password=root*

To Read data from properties file

*Properties prop=new Properties();*

*InputStream input=new FileInputStream(“dbInfo.properties”);*

*Prop.load(input);*

*Prop.getProperty(“driver.class”);*

Examle:

import java.io.FileInputStream;

import java.io.IOException;

import java.io.InputStream;

import java.util.Properties;

public class App {

public static void main(String[] args) {

Properties prop = new Properties();

InputStream input = null;

try {

input = new FileInputStream("config.properties");

// load a properties file

prop.load(input);

// get the property value and print it out

System.out.println(prop.getProperty("database"));

System.out.println(prop.getProperty("dbuser"));

System.out.println(prop.getProperty("dbpassword"));

} catch (IOException ex) {

ex.printStackTrace();

} finally {

if (input != null) {

try {

input.close();

} catch (IOException e) {

e.printStackTrace();

}

}

}

}

}

**Logging**

Logging is used to add the logger in program. Adding logger we can able to add the log in a file, so that it will be easy to trap various information which includes warning, error, info and debug related information.

**Logging with log4j**

Required Jar

i.log4j.jar

Configuration File

i.log4j.xml

Using Log4j in Code

Private static final Logger logger=Logger.getLogger(YourClass.class); //Initialized the logger class

*logger*.info("Add Product Called...."); //Using the logger level info

*logger*.debug("Add Product Called...."); //Using the logger level debug

*logger*.error("Error! Occured...."); //Using the logger level error

Eg,

*@Controller*

***public******class*** *ProductController {*

*@Autowired*

***private*** *ProductOraDataDao productDao;*

**private** **static** **final** Logger logger = Logger.getLogger(ProductController.**class**);

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

**public** String add(HttpServletRequest request,HttpServletResponse response,Model model){

logger.info("Add Product Called....");

**return** "addproduct";

}

}

**Happy Coding**