Cookies

Sending and receiving cookies

Tracking returning visitors through cookies.

# 1. Handling Cookies

The *javax.servlet.http* contains a Cookie class that allows us to quickly set up and process HTTP cookies and a HttpSession class that provides session tracking.

A cookie is a small piece of information. A server sends a cookie to a browser to be stored in the client computer and some attributes such as a comment, path, a maximum age, and a version number.After this process if the client makes a request from the server the cookie will be sent to the server. One of the benefits of cookies is remembering the user names and passwords. In this case the user does not have to enter username/password when she/he wants to request from a server which needs a username/password.

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. Some Web browsers have bugs in how they handle the optional attributes, so use them sparingly to improve the interoperability of your servlets.

The servlet sends cookies to the browser by using the [HttpServletResponse (Servlet API Documentation)](https://tomcat.apache.org/tomcat-5.5-doc/servletapi/javax/servlet/http/HttpServletResponse.html#addCookie(javax.servlet.http.Cookie)) method, which adds fields to HTTP response headers to send cookies to the browser, one at a time. The browser is expected to support 20 cookies for each Web server, 300 cookies total, and may limit cookie size to 4 KB each.

The browser returns cookies to the servlet by adding fields to HTTP request headers. Cookies can be retrieved from a request by using the [HttpServletRequest.getCookies()](https://tomcat.apache.org/tomcat-5.5-doc/servletapi/javax/servlet/http/HttpServletRequest.html" \l "getCookies())method. Several cookies might have the same name but different path attributes.

The server program creates a cookie object by ***new*** *Cookie(name, value)* and adds the cookie by calling *addCookie(cookie)* to the browser. If the client requests from the same server then the server reads the client’s cookie by calling *getCookies()*. Some methods that are useful in processing cookies are *getCookies(), getName(), getValue(), setValue(aString), setMaxAge (anInteger), setPath(URL), getPath(),getDomain( ), and setDomain(aDomain)* to retrieve, get the name, get the value, set the value, set the maximum age in seconds, specify path, get the path, get the domain, and to set the domain of a cookie respectively.

You can delete one or all the cookies from your window. For example in the IE browser we hit Ctrl-Shift-Delete to delete cookies. For Firefox: Tools 🡪Options 🡪 Privacy 🡪remove individual cookies.

Advantage of Cookies are 1) a simple technique to maintain a state 2) it resides on client.

Disadvantage of Cookies are 1) it can be disabled on the client side browser 2) cookie object can only hold textual data

## Example 1: The following program sends a cookie to a client.

### A: The html document:

<!--The name of the html document is: Main.html-->

<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN" "http://www.w3.org/TR/html4/loose.dtd">

<html>

<head>

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

<title>Example 1</title>

</head>

<body>

<form action="http://localhost:8080/Project/Main">

<center>

<br><br>

<input type=*"submit"* value=*"Submit Query"*>

</center>

</form>

</body>

</html>

### B: The servlet:

**import** java.io.\*;

**import** javax.servlet.ServletException;

**import** javax.servlet.annotation.WebServlet;

**import** javax.servlet.http.HttpServletRequest;

**import** javax.servlet.http.HttpServletResponse;

**import** javax.servlet.http.Cookie;

@WebServlet("/Main")

**public** **class** Main **extends** javax.servlet.http.HttpServlet **implements** javax.servlet.Servlet {

**public** Main() {

**super**();

}

**protected** **void** doGet(HttpServletRequest request,

HttpServletResponse response)**throws** ServletException, IOException {

Cookie aCookie = **new** Cookie("Visitor", "yes");

aCookie.setMaxAge(60\*60\*24\*365); // 1 year

response.addCookie(aCookie);

response.setContentType("text/html");

PrintWriter out = response.getWriter();

out.println("<html>\n" +

"<head><title>Processing Cookies</title></head>\n" +

"<body bgcolor=\"#fdf5e6\">\n" +

"<h1 align=\"center\"> Thanks I will remember you</h1>\n" +

"</body></html>");

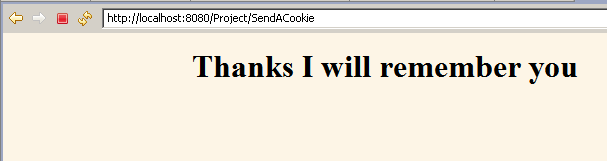
}

}

When you click on the submit button:

****

You get the following page.

****

### **Description**: The instruction:

Cookie aCookie = **new** Cookie("Visitor", "yes");

Creates a Cookie object that passes the name Visitor and the value yes to the object through the constructor. Note that the value also is a string.

Note: If you have this program, and want to write another program, it will be a good idea to use different strings for name and value – don’t choose Visitor for name and yes for value.

The instruction:

aCookie.setMaxAge(60\*60\*24\*365);

The cookie remains in the client computer for a year by the next instruction (provided the user does not delete it!).

Finally the instruction:

response.addCookie(aCookie);

Adds the cookie to the client computer.

We can track a client by a cookie that has been added to his/her computer before.

## Example 2: Check if a client has a cookie sent by the server:

### A: The html document:

<!--The name of the html document is: Main.html-->

<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN" "http://www.w3.org/TR/html4/loose.dtd">

<html>

<head>

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

<title>Example 2</title>

</head>

<body>

<form action="http://localhost:8080/Project/Main">

<center>

<br><br>

<input type="submit">

</center>

</form>

</body>

</html>

### B: The servlet:

**import** java.io.\*;

**import** javax.servlet.ServletException;

**import** javax.servlet.annotation.WebServlet;

**import** javax.servlet.http.HttpServletRequest;

**import** javax.servlet.http.HttpServletResponse;

**import** javax.servlet.http.Cookie;

@WebServlet("/Main")

**public** **class** Main **extends** javax.servlet.http.HttpServlet

**implements** javax.servlet.Servlet {

**public** Main() {

**super**();

}

**protected** **void** doGet(HttpServletRequest request,

HttpServletResponse response) **throws** ServletException, IOException {

String message = "Thank you for visiting us again";

Cookie[] cookies = request.getCookies();

**if** (cookies != **null** && cookies.length > 0)

**for**(**int** i=0; i<cookies.length; i++) {

Cookie c = cookies[i];

**if** ((c.getName().equals("Visitor")) &&

(c.getValue().equals("yes")))

**break**;

}

**else**

message = "I am sorry you have not visited us before.";

response.setContentType("text/html");

PrintWriter out = response.getWriter();

out.println("<html>\n" +

"<head><title>Processing Cookies</title></head>\n" +

"<BODY BGCOLOR=\"#FDF5E6\">\n" +

"<h1 align\"center\">" + message + "</h1>\n" +

"</body></html>");

}

}

If you have not deleted the cookies through IE browser and click on the Submit:



On the other hand, if you delete the cookies through IE you get:



### **Description**: In the server the instruction:

Cookie[] cookies = request.getCookies();

Obtains all the cookies from the client.

Provided there are cookies, the for-loop:

**for**(**int** i=0; i<cookies.length; i++) {

Cookie c = cookies[i];

**if** ((c.getName().equals("Visitor")) &&

(c.getValue().equals("yes")))

**break**;

}

Looks for a specific cookie by the name "Visitor" and the value "yes". If there is such a cookie breaks out of the loop and issues a message: "Thank you for visiting us again". Otherwise issues the message: "I am sorry you have not visited us before."

# 2. Serialization

Serialization is a technique of converting the state of an object into a byte stream. Deserialization is the reverse process where the byte stream is used to recreate the actual Java object in memory. This mechanism is used to persist with the object.

The byte stream created is platform independent. So, the object serialized on one platform can be de-serialized on a different platform.

Both ObjectInputStream and ObjectOutputStream are high level classes that extend java.io.InputStream and java.io.OutputStream respectively. ObjectOutputStream can write primitive types and graphs of objects to an OutputStream as a stream of bytes. These streams can subsequently be read using ObjectInputStream.

To make a Java object serializable we implement the **java.io.Serializable** interface.  
The ObjectOutputStream class contains **writeObject()** method for serializing an Object.

## Example:

public class Employee implements java.io.Serializable {

public String name;

public String address;

public transient int SSN;

public int number;

public void mailCheck() {

System.out.println("Mailing a check to" + name + " " + address);

}

}

The class must implement the java.io.Serializable interface.

All of the fields in the class must be serializable. If a field is not serializable, it must be marked **transient**.

Classes that do not implement this interface will not have any of their state serialized or deserialized. All subtypes of a serializable class are themselves serializable. The serialization interface has no methods or fields and serves only to identify the semantics of being serializable.

When traversing a graph, an object may be encountered that does not support the Serializable interface. In this case the NotSerializableException will be thrown and will identify the class of the non-serializable object.

# 3. Java Beans

JavaBeans are classes that encapsulate many objects into a single object (the bean). They are serializable, have a zero-argument constructor, and allow access to properties using getter and setter methods. The name "Bean" was given to encompass this standard, which aims to create reusable software components for Java.

Java Bean is like any other normal Java class which has private properties with its public getter and setter method. Java Beans are generally used as helper class. Implementing Serializable is not mandatory but is very useful if you'd like to persist or transfer Javabeans outside Java's memory, e.g. in hard disk or over network.

The properties, events, and methods of a bean can be exposed to another application. Getter and Setter methods are used to access those in a java bean class.

## An example:

//Employee.java

package mypack;

public class Employee implements java.io.Serializable{

private int id;

private String name;

public Employee(){}

public void setId(int id){this.id=id;}

public int getId(){return id;}

public void setName(String name){this.name=name;}

public String getName(){return name;}

}

Accessing a Java bean class:

package mypack;

public class Test{

public static void main(String args[]){

Employee e=new Employee();//object is created

e.setName("Jane");//setting value to the object

System.out.println(e.getName());

}}

# 4. Enterprise Java Beans

Enterprise Java Bean (EJB) is a specification to develop secured, robust and scalable distributed applications. EJB applications are deployed on the server, so it is often called server side component. It is run on application server (EJB Container) such as Jboss, Glassfish, Weblogic, Websphere etc. It performs: life cycle management, security, transaction management, and object pooling.

Object pooling is a collection of particular objects that an application may create and keep those on hand for those situations where creating each instance is expensive. A database connector is a good example.

**When to use EJB:** When an application needs Remote Access. In other words, it is distributed. When an application needs to be scalable. EJB applications supports load balancing, clustering and fail-over. And when an application needs encapsulated business logic. EJB application is separated from presentation and persistent layer.

**Types of enterprise bean in Java:**

Session Bean: Session bean contains business logic that can be invoked by local, remote or webservice client.

Message Driven Bean: Like Session Bean, it contains the business logic but it is invoked by passing message.

Entity Bean: It encapsulates the state that can be persisted in the database. It is deprecated (a replacement of it is JPA (Java Persistent API)).

**Types of Enterprise Java Bean:** Remote Method Invocation (RMI) and EJB

**Difference between RMI and EJB:** Both RMI and EJB, provides services to access an object running in another JVM (known as remote object) from a JVM. The differences between RMI and EJB are as follows:

RMI is not a server-side component. It is not required to be deployed on the server. EJB is a server-side component, it is required to be deployed on the server.

In RMI, middleware services such as security, transaction management, object pooling etc. need to be done by the java programmer. In EJB, middleware services are provided by EJB Container automatically.

RMI is built on the top of socket programming. EJB technology is built on the top of RMI.

**Disadvantages of EJB:** It requires application server. It is complex to understand and develop an EJB application. It requires only client side application written in Java and not compatible for clients written in other languages.