Manejo de Cookies y Sesión

Raúl Estrada

Octubre 2020

1. Overview

In this tutorial, we'll cover **the handling of cookies and sessions in Java, using Servlets**.

Additionally, we'll shortly describe what a cookie is, and explore some sample use cases for it.

2. Cookie Basics

Simply put, **a cookie is a small piece of data stored on the client-side which servers use when communicating with clients**.

They're used to identify a client when sending a subsequent request. They can also be used for passing some data from one servlet to another.

2.1. Create a Cookie

The *Cookie* class is defined in the ***javax.servlet.http*** package.

To send it to the client, we need to **create one and add it to the response**:

```
1  Cookie uiColorCookie = new Cookie("color", "red");
2  response.addCookie(uiColorCookie);
```

However, its API is a lot broader – let's explore it.

2.2. Set the Cookie Expiration Date

We can set the max age (with a method *maxAge(int)*) which defines how many seconds a given cookie should be valid for:

```
1 uiColorCookie.setMaxAge(60*60);
```

We set a max age to one hour. After this time, the cookie cannot be used by a client (browser) when sending a request and it also should be removed from the browser cache.

2.3. Set the Cookie Domain

Another useful method in the *Cookie* API is *setDomain(String)*.

This allows us to specify domain names to which it should be delivered by the client. It also depends on if we specify domain name explicitly or not.

Let's set the domain for a cookie:

```
1 uiColorCookie.setDomain("example.com");
```

The cookie will be delivered to each request made by *example.com* and its subdomains.

If we don't specify a domain explicitly, it will be set to the domain name which created a cookie.

For example, if we create a cookie from *example.com* and leave domain name empty, then it'll be delivered to the *www.example.com* (without subdomains).

Along with a domain name, we can also specify a path. Let's have a look at that next.

2.4. Set the Cookie Path

The path specifies where a cookie will be delivered.

If we specify a path explicitly, then a *Cookie* will be delivered to the given URL and all its subdirectories:

```
1 uiColorCookie.setPath("/welcomeUser");
```

Implicitly, it'll be set to the URL which created a cookie and all its subdirectories.

Now let's focus on how we can retrieve their values inside a *Servlet*.

2.5. Read Cookies in the Servlet

Cookies are added to the request by the client. The client checks its parameters and decides if it can deliver it to the current URL.

We can get all cookies by calling *getCookies()* on the request (*HttpServletRequest*) passed to the *Servlet*.

We can iterate through this array and search for the one we need, e.g., by comparing their names:

```
public Optional<String> readCookie(String key) {
    return Arrays.stream(request.getCookies())
      .filter(c -> key.equals(c.getName()))
      .map(Cookie::getValue)
      .findAny();
}
```

2.6. Remove a Cookie

To remove a cookie from a browser, we have to add a new one to the response with the same name, but with a *maxAge* value set to 0:

```
Cookie userNameCookieRemove = new Cookie("userName", "");
userNameCookieRemove.setMaxAge(0);
response.addCookie(userNameCookieRemove);
```

A sample use case for removing cookies is a user logout action – we may need to remove some data which was stored for an active user session.

Now we know how we can handle cookies inside a *Servlet*.

Next, we'll cover another important object which we access very often from a *Servlet* – a *Session* object.

3. *HttpSession* Object

The *HttpSession* is another option for storing user-related data across different requests. A session is a server-side storage holding contextual data.

Data isn't shared between different session objects (client can access data from its session only). It also contains key-value pairs, but in comparison to a cookie, a session can contain object as a value. The storage implementation mechanism is server-dependent.

A session is matched with a client by a cookie or request parameters. More info can be found here.

3.1. Getting a Session

We can obtain an *HttpSession* straight from a request:

```
HttpSession session = request.getSession();
```

The above code will create a new session in case it doesn't exist. We can achieve the same by calling:

```
request.getSession(true)
```

In case we just want to obtain existing session and not create a new one, we need to use:

```
request.getSession(false)
```

If we access the JSP page for the first time, then a new session gets created by default. We can disable this behavior by setting the *session* attribute to *false:*

```
<%@ page contentType="text/html;charset=UTF-8" session="false" %>
```

In most cases, a web server uses cookies for session management. When a session object is created, then a server creates a cookie with *JSESSIONID* key and value which identifies a session.

3.2. *Session* Attributes

The session object provides a bunch of methods for accessing (create, read, modify, remove) attributes created for a given user session:

- *setAttribute(String, Object)* which creates or replaces a session attribute with a key and a new value
- *getAttribute(String)* which reads an attribute value with a given name (key)
- *removeAttribute(String)* which removes an attribute with a given name

We can also easily check already existing session attributes by calling *getAttributeNames()*.

As we already mentioned, we could retrieve a session object from a request. When we already have it, we can quickly perform methods mentioned above.

We can create an attribute:

```
HttpSession session = request.getSession();
session.setAttribute("attributeKey", "Sample Value");
```

The attribute value can be obtained by its key (name):

```
session.getAttribute("attributeKey");
```

We can remove an attribute when we don't need it anymore:

```
session.removeAttribute("attributeKey");
```

A well-known use case for a user session is to invalidate whole data it stores when a user logs out from our website. The session object provides a solution for it:

```
session.invalidate();
```

This method removes the whole session from the web server so we cannot access attributes from it anymore.

HttpSession object has more methods, but the one we mentioned are the most common.

4. Conclusion

In this article, we covered two mechanism which allows us to store user data between subsequent requests to the server – the cookie and the session.

Keep in mind that the HTTP protocol is stateless, and so maintaining state across requests is a must.

As always, code snippets are available over on Github.