

10 Extending JAAS Integration in Web Applications

As the last chapter demonstrated, the Servlet specification provides an API for authenticating users, and testing the user's membership in roles that represent the user's Principals. Since JAAS is not tightly coupled to the Servlet spec, using the other features of JAAS, such as permission checking, are not as easily accomplished. To use all of JAAS, you need access to the authenticated `javax.security.auth.Subject` instance, allowing you to create Subject-based privileged blocks. This chapter demonstrates one way to get an authenticated Subject, and then goes over using that Subject to perform authorization checks.

10.1 The AuthenticationFilter

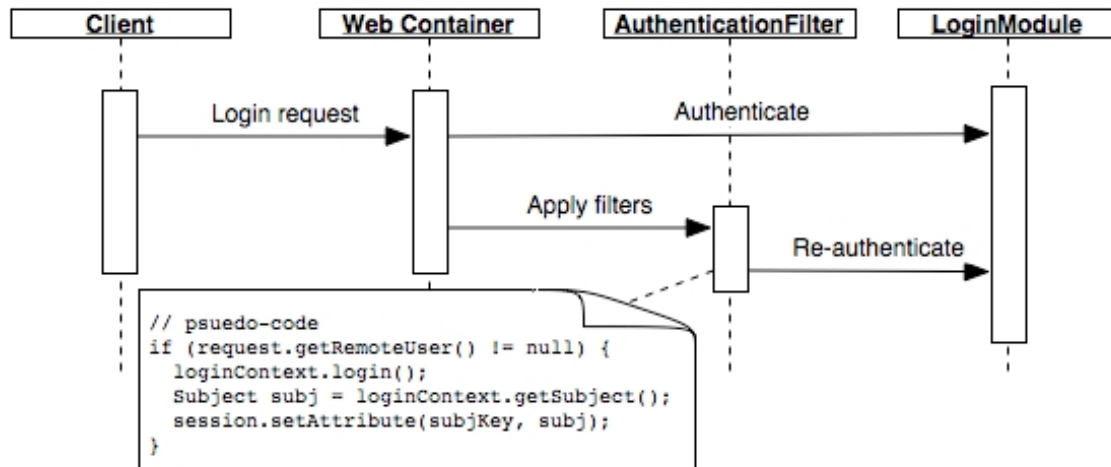
If there is no other way for you to obtain the authenticated Subject from your web container, you can use a `ServletFilter` to create one yourself. In this strategy, the user is authenticated twice: once when the web container logs the user in where you *cannot* have access to the Subject, and again in a `ServletFilter` where you *can* access to the Subject. The second round of authentication is done automatically, without the user having to provide their credentials. The ability to do this relies on the way that `HttpServletRequest`'s `getRemoteUser()` works. The `getRemoteUser()` method returns the username of the authenticated user for the request. If there is no authenticated user, the method returns null. Thus, when `getRemoteUser()` returns a non-null value, you know that the container has already authenticated the user by prompting for the username and password. Knowing this, you can go through the process of authenticating the user without actually asking for their credentials. This step allows you to easily get access to an authenticated Subject instance by going through the process of authenticating a user without having to prompt the user for their username and password.

The diagram below illustrates the process:



This work is licensed under a Creative Commons Attribution-NonCommercial 2.5

License: <http://creativecommons.org/licenses/by-nc/2.5/>



When the client first logs in, the web container catches the special request (by looking for the action `j_security_check`) and uses your JAAS `LoginModule(s)` to authenticate the user. Once the user has been successfully authenticated, the web container will apply any `ServletFilters` configured. To re-authenticate the user, allowing us to acquire a reference to the authenticated `Subject`, we configure the `AuthenticationFilter` to be applied to all requests, meaning that the filter will be run after the web container has performed its own authentication. As the pseudo-code in the diagram shows, each time the `AuthenticationFilter` runs, it checks if the user has been authenticated in the current request by calling `getRemoteUser()`. If the request is authenticated, the filter re-authenticates the user with its own `LoginContext` instance, allowing the Filter to access the authenticated `Subject`. Once the `AuthenticationFilter` authenticates the `Subject`, it caches the `Subject` in the session so that other components in the web application can access the `Subject`.

The code for the `AuthenticationFilter` is below:

```

package chp10;

import java.io.IOException;
import java.sql.SQLException;
import java.util.logging.Level;
import java.util.logging.Logger;

import javax.security.auth.Subject;
import javax.security.auth.login.LoginContext;
import javax.security.auth.login.LoginException;
import javax.servlet.Filter;
import javax.servlet.FilterChain;
import javax.servlet.FilterConfig;
import javax.servlet.ServletException;
import javax.servlet.ServletRequest;
import javax.servlet.ServletResponse;
import javax.servlet.http.HttpServletRequest;
    
```



```
import chp04.UserService;
import chp07.BundleCallbackHandler;

public class AuthenticationFilter implements Filter {

    public void init(FilterConfig config) throws ServletException { #1
        appName_ = config.getInitParameter("app-name");
        subjectKey_ = config.getInitParameter("subject-key");
        if (subjectKey_ == null) {
            subjectKey_ = DEFAULT_SUBJECT_KEY;
        }
    }

    public void doFilter(ServletRequest request,
        ServletResponse response, FilterChain chain)
        throws IOException, ServletException {
        HttpServletRequest httpRequest = (HttpServletRequest) request;

        String remoteUser = httpRequest.getRemoteUser();
        if (remoteUser != null) { #2

            if (LOGGER.isLoggable(Level.FINE)) {
                Subject subj = (Subject) httpRequest.getSession()
                    .getAttribute(subjectKey_);

                LOGGER.logp(Level.FINE, LOG_TOPIC, "doFilter()",
                    "Subject found under key {0}:\n{1}", new Object[] {
                        subjectKey_, subj });
            }

            String password = null;
            try {
                password = UserService.lookupPassword(remoteUser); #3
            } catch (SQLException e) {
                throw new ServletException(
                    "Error retrieving credentials for " + remoteUser, e);
            }
            BundleCallbackHandler cb = new BundleCallbackHandler(
                remoteUser, password);
            try {
                LoginContext ctx = new LoginContext(appName_, cb);
                ctx.login();
                Subject subj = ctx.getSubject();

                httpRequest.getSession().setAttribute(subjectKey_, subj);

                LOGGER.info("Authenticated Subject " + subj
                    + ". Under session key " + subjectKey_);
            } catch (LoginException e) {
                LOGGER
                    .logp(
```



```
        Level.WARNING,
        LOG_TOPIC,
        "doFilter()",
        "LoginException thrown when validating user {0}."
Exception:\n{1}",
        new Object[] { remoteUser, e });
    }

    }

    chain.doFilter(request, response);
}

public void destroy() {
}

static private String LOG_TOPIC = AuthenticationFilter.class
    .getName();

static private Logger LOGGER = Logger.getLogger(LOG_TOPIC);

static private final String DEFAULT_SUBJECT_KEY = "subject";

private String appName_;

private String subjectKey_;

}
(annotation) <#1: [init()]: The AuthenticationFilter is configured with 2 filter parameters: the name of the
JAAS AppConfigurationEntry group to use, and the session key to place the authenticated Subject
under. A default value of "subject" is used for subject-key if that parameter isn't specified.>
(annotation) <#2: [Check for remoteUserName]: Authentication is done for every request, assuring that the changes to
the Subject's Principal set take effect while the user is logged in.
(annotation) <#3: [Lookup password]: to simplify the example, the filter looks up a user's password so that we can
reuse the same TomcatLoginModule as used in chapter 9. Alternatively, to avoid looking up the plaintext
password, you could use a LoginModule that doesn't require a password to authenticate a user. The assumption with
this type of LoginModule would be that the user was already authenticated.>
```

Sidebar: Obtaining the Subject in Different Application Servers.

Depending on the application server your application is running in, you may be able to use one of the below methods to get the authenticated Subject:

JBoss: the class `org.jboss.security.SecurityAssociation` provides the static method `getSubject()`.

WebSphere: the class `com.ibm.websphere.security.auth.WSSubject` provides the static method `getCallerSubject()`.

WebLogic: the class `weblogic.security.Security` provides the static method `getCurrentSubject()`.



This work is licensed under a Creative Commons Attribution-NonCommercial 2.5
License: <http://creativecommons.org/licenses/by-nc/2.5/>

End Sidebar

10.2 The DoAsPrivilegedFilter

With a reference to the authenticated Subject available, our web application can use all of the features JAAS provides. To make checking permissions easier, we use another ServletFilter to wrap the entire request in a privileged block using the Subject's static method `doAsPrivileged()`. Once this filter is applied, calls in code to `AccessController.checkPermission()` will use the authenticated Subject when checking for access.

The code for `DoAsPrivilegedFilter` is below:

```
package chp10;

import java.io.IOException;
import java.security.PrivilegedActionException;
import java.security.PrivilegedExceptionAction;
import java.util.logging.Level;
import java.util.logging.Logger;

import javax.security.auth.Subject;
import javax.servlet.Filter;
import javax.servlet.FilterChain;
import javax.servlet.FilterConfig;
import javax.servlet.ServletException;
import javax.servlet.ServletRequest;
import javax.servlet.ServletResponse;
import javax.servlet.http.HttpServletRequest;

public class DoAsPrivilegedFilter implements Filter {

    public void init(FilterConfig config) throws ServletException { #1
        subjectKey_ = config.getInitParameter("subject-key");
        if (subjectKey_ == null) {
            subjectKey_ = DEFAULT_SUBJECT_KEY;
        }
    }

    public void doFilter(final ServletRequest request,
        final ServletResponse response, final FilterChain chain)
        throws IOException, ServletException {
        HttpServletRequest httpRequest = (HttpServletRequest) request;
        Subject subj = (Subject) httpRequest.getSession().getAttribute(
            subjectKey_);
        if (subj == null) {
            LOGGER
                .logp(
                    Level.FINE,
```



This work is licensed under a Creative Commons Attribution-NonCommercial 2.5 License: <http://creativecommons.org/licenses/by-nc/2.5/>

```
        LOG_TOPIC,
        "doFilter()",
        "No Subject found under key {0}, so creating "+
        "new Subject.",
        subjectKey_);
    subj = new Subject();
}
try {
    if (LOGGER.isLoggable(Level.FINE)) {
        LOGGER.log(Level.FINE, LOG_TOPIC, "doFilter()",
            "Running doAsPrivileged block with Subject: {0}", subj);
    }

    Subject.doAsPrivileged(subj, new PrivilegedExceptionAction() { #2

        public Object run() throws Exception {
            chain.doFilter(request, response);
            return null;
        }

    }, null);
} catch (PrivilegedActionException e) {
    LOGGER
        .logp(
            Level.SEVERE,
            LOG_TOPIC,
            "doFilter()",
            "Exception executing filter with Subject:\n"+
                "{0}\nException: {1}",
            new Object[] { subj, e });
    throw new ServletException(e);
}
}

public void destroy() {
}

static private String LOG_TOPIC = DoAsPrivilegedFilter.class
    .getName();
static private Logger LOGGER = Logger.getLogger(LOG_TOPIC);
static private final String DEFAULT_SUBJECT_KEY = "subject";
private String subjectKey_;
}
```

(annotation) <#1 [init()]: **DoAsPrivilegedFilter** is configured with one optional init parameter, **subject-key**, which specifies session key to look for the authenticated Subject under. If this parameter is not specified, the default value of "subject" is used.>

(annotation) <#2 [Subject.doAsPrivileged() call]: **DoAsPrivilegedFilter** passes in a null **AccessControllerContext** as the third argument to **doAsPrivileged**, assuring that only authenticated **Subject** is used for authorization.>



This work is licensed under a Creative Commons Attribution-NonCommercial 2.5

License: <http://creativecommons.org/licenses/by-nc/2.5/>

10.2.1 Advantages & Limitations of the DoAsPrivilegedFilter

Aesthetically, the `DoAsPrivilegedFilter` is appealing because it wraps an entire request in a secure block of code. The chain of code from the UI all the way down to the back-end is protected in this block. Also, instead of passing the `Subject` down the entire execution chain, you can rely on JAAS to provide the `Subject` as needed.

Unfortunately, wrapping your entire request in a privileged block can cause several challenging problems in an application that uses a lot of 3rd party libraries. Because the entire execution stack is executed in the privileged block, the `Subject` must be granted all the permissions needed by each piece of code, or each block of code must be wrapped in its own privileged block and granted those permissions. With the amount of third party libraries in most Java programming, assuring either of these two states can be incredibly time consuming and tedious. Most Java code is not written with such JAAS-centric concerns in mind, let alone documented with the permissions needed for normal execution. The result is that if you want to wrap an entire request in a privileged block, you'll have several days, if not weeks, of tedious trial and error in front of you as you try out each path in your application to discover which permissions a `Subject` must be granted. This approach may work for small applications, or applications that depend on few third party libraries, but will be cost prohibitive for medium to large applications.

If you're concerned about securing your application to the hilt, going through this process may be worth it to you. Otherwise, should strongly consider not using the `DoAsPrivileged` filter, and instead performing security checks in the sensitive parts of your code, such as the code that retrieves or updates sensitive data.

10.3 The Permission Tag Libraries

With the request wrapped in a privileged block, we can now use JAAS authorization methods, namely `AccessController.checkPermission()`. As our example, we'll create two custom tag libraries that perform two very useful functions:

- The `perm:granted` tag which will only display its body if the logged in user has been granted the permission.
- The `perm:notGranted` tag which will only display its body if the user hasn't been granted the permission.

For example, the following JSP page will display different text if the user has been granted the permission to read and write the file `/tmp/test.txt`:

```
<%@ taglib uri="perm-tags" prefix="perm" %>
<html>
<head><title>Permission Check</title></head>
<body>
<perm:granted type="java.io.FilePermission"
               name="/tmp/test.txt"
               actions="read,write">
Granted FilePermission to read and write to /tmp/test.txt
</perm:granted>
```



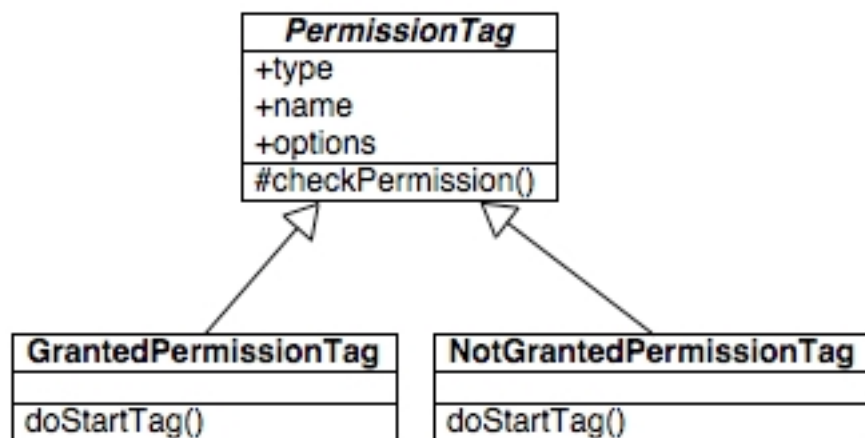
```
<perm:notGranted type="java.io.FilePermission"
                  name="/tmp/test.txt"
                  actions="read,write">
Not granted FilePermission to read and write to /tmp/test.txt
</perm:notGranted>
</body>
</html>
```

If the authenticated Subject can read and write the file, they'll see the first block of text. If they cannot read and write the file, they'll see the second block of text.

10.3.1 Permission Tag Classes

The implementation of the two permission tags, `perm:granted` and `perm:notGranted`, is provided largely by the abstract class `PermissionTag`. This class has the responsibility of collecting the tag attributed, instantiating the permission with the appropriate name and optional actions, and then checking if the Subject has been granted the permission. The two sub-classes `GrantedPermissionTag` and `NotGrantedPermissionTag` implement `doStartTag()`, calling `PermissionTag`'s `checkPermission()`, and then displaying the tag's body accordingly.

The relationship between these 3 classes is diagramed below:



The code for each tag is listed in the following sections.

PermissionTag

The primary work done by the `PermissionTag` is done in the `checkPermission()` method. This method attempts to reflectively instantiate the permission specified by the type, name, and optional actions tags. Once the permission instance is created, it uses that instance to call `AccessController.checkPermission()`.

```
package chp10;

import java.lang.reflect.Constructor;
import java.lang.reflect.InvocationTargetException;
import java.security.AccessController;
```



This work is licensed under a Creative Commons Attribution-NonCommercial 2.5 License: <http://creativecommons.org/licenses/by-nc/2.5/>


```
import java.security.Permission;

import javax.security.auth.Subject;
import javax.servlet.jsp.JspException;
import javax.servlet.jsp.tagext.TagSupport;

public abstract class PermissionTag
    extends TagSupport {

    protected boolean checkPermission() throws JspException {
        Subject ctxSubject = Subject.getSubject(AccessController
            .getContext());
        String type = getType();
        if (type == null) {
            throw new NullPointerException("type is null.");
        }

        String name = getName();
        String actions = getActions();
        Permission perm = null;

        Class clazz = null;
        try {
            clazz = Class.forName(type);
        } catch (ClassNotFoundException e) {
            throw new JspException(type + " was not found.", e);
        }

        if (!Permission.class.isAssignableFrom(clazz)) {
            throw new IllegalArgumentException(type
                + " is not a java.security.Permission.");
        }

        try {
            if (name != null && actions == null) {
                Constructor c = clazz
                    .getConstructor(new Class[] { String.class });
                perm = (Permission) c.newInstance(new Object[] { name });
            } else if (name != null && actions != null) {
                // name and actions
                Constructor c = clazz.getConstructor(new Class[] {
                    String.class, String.class });
                perm = (Permission) c.newInstance(new Object[] { name,
                    actions });
            } else {
                throw new NullPointerException(
                    "Permission name must be specified.");
            }
        } catch (SecurityException e) {
            throw new JspException(e);
        } catch (NoSuchMethodException e) {
            throw new JspException("Could not instantiate " + type
```



```
        + " instance.", e);
    } catch (IllegalArgumentException e) {
        throw new JspException(e);
    } catch (InstantiationException e) {
        throw new JspException(e);
    } catch (IllegalAccessException e) {
        throw new JspException(e);
    } catch (InvocationTargetException e) {
        throw new JspException(e);
    }

    boolean granted = true;

    try {
        AccessController.checkPermission(perm);
    } catch (SecurityException e) {
        granted = false;
    }
    return granted;
}

public String getActions() {
    return actions_;
}

public void setActions(String actions) {
    actions_ = actions;
}

public String getName() {
    return name_;
}

public void setName(String name) {
    name_ = name;
}

public String getType() {
    return type_;
}

public void setType(String type) {
    type_ = type;
}

private String type_;
private String name_;
private String actions_;
}
```

GrantedPermissionTag



This work is licensed under a Creative Commons Attribution-NonCommercial 2.5

License: <http://creativecommons.org/licenses/by-nc/2.5/>

The `GrantedPermissionTag` displays the tag body if the permission has been granted, or omits the tag's body if the Permission has not been granted:

```
package chp10;

import javax.servlet.jsp.JspException;

public class GrantedPermissionTag
    extends PermissionTag {
    public int doStartTag() throws JspException {
        boolean granted = checkPermission();

        if (granted) {
            return EVAL_BODY_INCLUDE;
        } else {
            return SKIP_BODY;
        }
    }
}
```

NotGrantedPermissionTag

The `NotGrantedPermissionTag` displays the tag body if the permission is not granted, or omits the body if the permission has been granted:

```
package chp10;

import javax.servlet.jsp.JspException;

public class GrantedPermissionTag
    extends PermissionTag {
    public int doStartTag() throws JspException {
        boolean granted = checkPermission();

        if (granted) {
            return EVAL_BODY_INCLUDE;
        } else {
            return SKIP_BODY;
        }
    }
}
```

Tag Library Descriptor

The following TLD is used to specify the usage of the two tags:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE taglib PUBLIC "-//Sun Microsystems, Inc.//DTD JSP Tag Library
1.1//EN" "http://java.sun.com/j2ee/dtds/web-jsptaglibrary_1_1.dtd">
<taglib>
    <tlibversion>1.0</tlibversion>
```



This work is licensed under a Creative Commons Attribution-NonCommercial 2.5
License: <http://creativecommons.org/licenses/by-nc/2.5/>

```
<jspversion>1.1</jspversion>
<shortname>auth</shortname>
<uri>/WEB-INF/auth-tags.tld</uri>
<tag>
  <name>granted</name>
  <tagclass>chp10.GrantedPermissionTag</tagclass>
  <bodycontent>JSP</bodycontent>
  <attribute>
    <name>type</name>
    <required>true</required>
    <rtexprvalue>true</rtexprvalue>
  </attribute>
  <attribute>
    <name>name</name>
    <required>true</required>
    <rtexprvalue>true</rtexprvalue>
  </attribute>
  <attribute>
    <name>actions</name>
    <required>false</required>
    <rtexprvalue>true</rtexprvalue>
  </attribute>
</tag>
<tag>
  <name>notGranted</name>
  <tagclass>chp10.NotGrantedPermissionTag</tagclass>
  <bodycontent>JSP</bodycontent>
  <attribute>
    <name>type</name>
    <required>true</required>
    <rtexprvalue>true</rtexprvalue>
  </attribute>
  <attribute>
    <name>name</name>
    <required>true</required>
    <rtexprvalue>true</rtexprvalue>
  </attribute>
  <attribute>
    <name>actions</name>
    <required>false</required>
    <rtexprvalue>true</rtexprvalue>
  </attribute>
</tag>
</taglib>
```

10.4 Pulling it all Together

To demonstrate the filters and tags introduced in this chapter, we'll expand on the web application from chapter 9. As in chapter 9, the database is populated with two users, and admin and a customer, each belonging to two different roles. The admin user is granted the



This work is licensed under a Creative Commons Attribution-NonCommercial 2.5

License: <http://creativecommons.org/licenses/by-nc/2.5/>

permission to read and write to the file /tmp/test.txt, while the customer's is *not* granted that permission.

Deploying the Example Web Application

To deploy the example web application, change to the source code directory and type `ant deploy-chp10`. This will seed the test database with the appropriate users and permissions, configure the web application, and deploy it to the Tomcat install.

Once you start Tomcat, you'll be able to load the example web application in your browser with the URL `http://localhost:8080/jass-book-chp10/`.

Adding Principals to the Users

The example web applications first uses the the SQL inserts used in chapter 9 to setup up the database. Additionally, the following SQL is used to grant `java.io.FilePermission` to all users in the admin user group access to our test file:

```
INSERT INTO permission VALUES
('file-perm-id','java.io.FilePermission','/tmp/test.txt','read,write')

INSERT INTO principal_permission VALUES
('admin-principal-id','file-perm-id')
```

Adding in the Filters

First, in chapter 10's `web.xml`, we add in the `AuthenticationFilter` and the `DoAsPrivilegedFilter`, making sure to map them to all URLs:

```
<!--except from src/webapp/chp10/WEB-INF/web.xml -->
<filter>
  <filter-name>authentication-filter</filter-name>
  <filter-class>chp10.AuthenticationFilter</filter-class>
  <init-param>
    <param-name>app-name</param-name>
    <param-value>chp09</param-value>
  </init-param>
  <init-param>
    <param-name>subject-key</param-name>
    <param-value>subject</param-value>
  </init-param>
</filter>

<filter>
  <filter-name>privileged-filter</filter-name>
  <filter-class>chp10.DoAsPrivilegedFilter</filter-class>
  <init-param>
    <param-name>subject-key</param-name>
    <param-value>subject</param-value>
  </init-param>
</filter>

<filter-mapping>
```



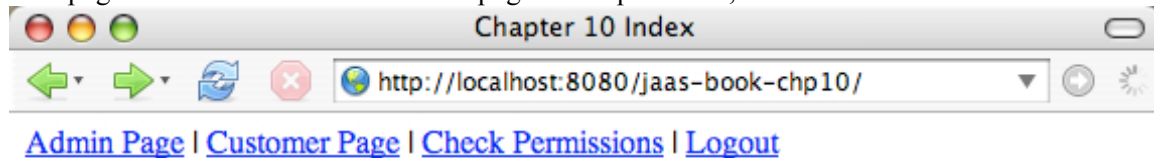
```
<filter-name>authentication-filter</filter-name>
<url-pattern>/*</url-pattern>
</filter-mapping>

<filter-mapping>
  <filter-name>privileged-filter</filter-name>
  <url-pattern>/*</url-pattern>
</filter-mapping>
```

With these filters in place, we'll be able to get the currently logged in Subject from the session, and all requests will execute in the security context of that Subject.

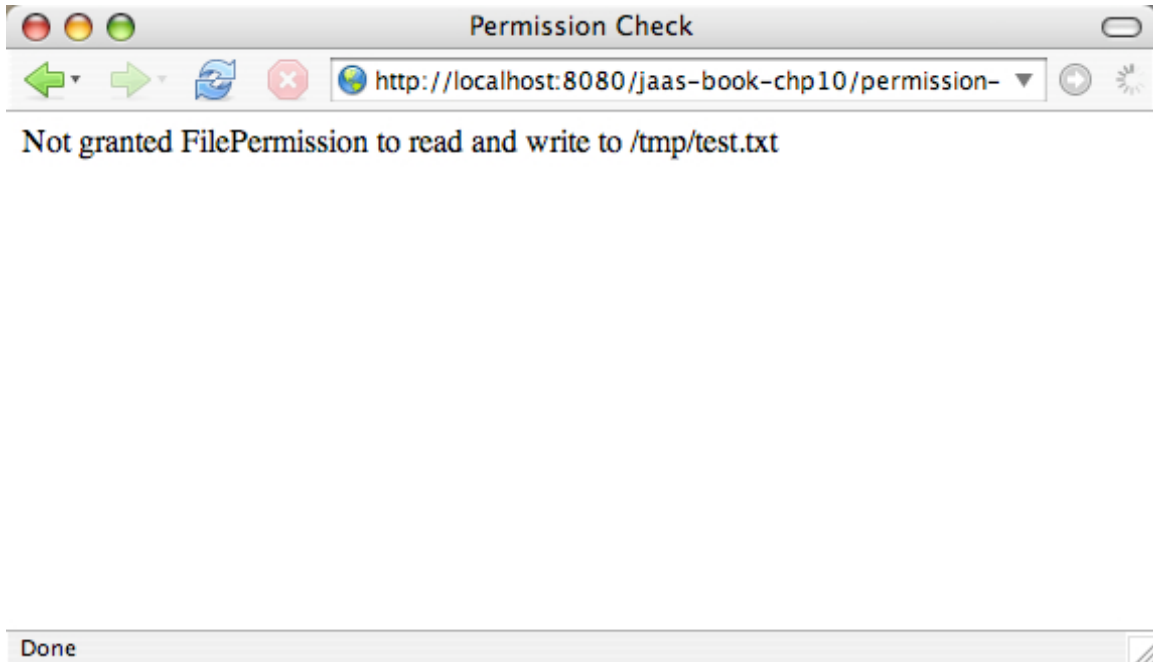
Using the Taglibs

We first add in a link to the `permission-check.jsp` from above to the index page. The first page we see looks familiar to the page in chapter nine, but has a Check Permissions link:

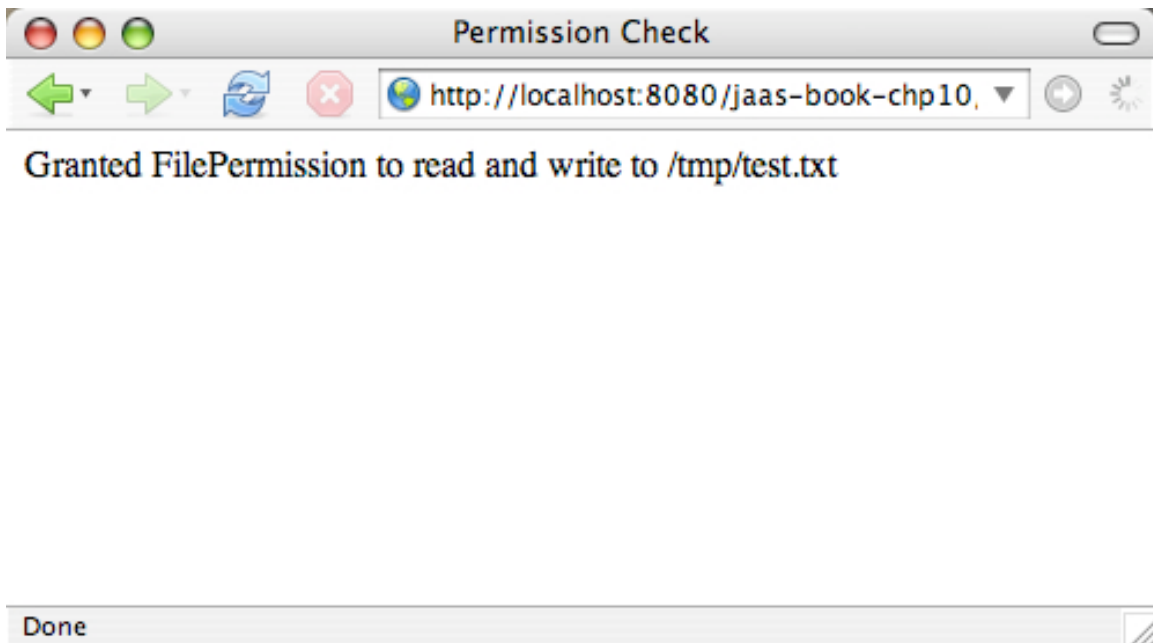


The page `permission-check.jsp`, listed above in section XXX, uses the permission tags to display different messages when the authenticated user is granted the correct `java.io.FilePermission` and when the user isn't granted the `java.io.FilePermission`. When a user that has't been granted permission to access the file clicks on Check Permissions, they see this message:





Once the user is granted permission, they'll see this message:



To test this out yourself, first login as the customer with the username/password customer/secret. You'll see the first, "Not granted" page. Then, logout, and login as the admin with the username "admin" and the password "secret." This time, you'll see the "Granted" page.



This work is licensed under a Creative Commons Attribution-NonCommercial 2.5 License: <http://creativecommons.org/licenses/by-nc/2.5/>

Summary

This chapter introduced several ways to use JAAS to secure web applications. First, we used a `ServletFilter` to get the authenticated `Subject` from the web container. Once we had the `Subject`, we could wrap an entire request in a privileged filter, allowing the code to execute as the requesting `Subject`. Finally, we implemented two tags that conditionally show their JSP bodies if the appropriate `Permission` has been granted to the logged in `Subject`. With each of the above, you can easily use JAAS to secure any web application.

