Code Inspection

Alessandro Comodi, Davide Coficconi, Stefano Longari

v1.0

January 4, 2016

Contents

Chapter 1

Class, methods assigned

The block of code assigned is included in only one class and regards in particular three methods, which will be shown below.

- Name of the class: WebdavServlet
- Location: appserver/web/web-core/src/main/java/org/apache/catalina/servlets/WebdavServlet.java
- Methods:
  1. doUnlock(HttpServletRequest req , HttpServletResponse resp), starting at line 1420
  2. isLocked(String path , String ifHeader), starting at line 1538
  3. copyResource(HttpServletRequest req , HttpServletResponse resp), starting at line 1596

Chapter 2

Functional role of assigned class, methods

The assigned class is a servlet, which is a Java applet and helps the interaction with the server. It offers many services in order to let the application work via the web and handles the http requests and responses.

2.1 Role of assigned methods

Below will be presented a short explanation of the functional role of the assigned pieces of code.

- doUnlock(HttpServletRequest req , HttpServletResponse resp), starting at line 1420:
  This method, as says its name, unlocks a resource that was previously locked with the use of the "doLock" method, which is situated just before the "doUnlock". If the request is locked or if the resource is a readOnly, the doUnlock returns without doing any change. Otherwise it starts to remove all the resource locks and inheritable collection locks, sending, in the end, a Status Code which informs the success of the operation.

- isLocked(String path , String ifHeader), starting at line 1538:
  This method checks whether a resource in a certain path is currently "write locked" and, if so, it returns true.

- copyResource(HttpServletRequest req , HttpServletResponse resp), starting at line 1596:
  This method offers the possibility to copy a resource from a source to a destination. In case the copy fails the method returns "false" and "true" otherwise.

Chapter 3

List of issues found by applying the checklist

In this chapter will be anaylized the various issues of the methods and the class previously described.

3.1 Code to be inspected

Below there is the code present in the three assigned methods that have been inspected.

3.1.1 doUnlock

```java
    /**
     * UNLOCK Method.
     */
    protected void doUnlock(HttpServletRequest req, HttpServletResponse resp)
        throws ServletException, IOException {

        if (readOnly) {
            resp.sendError(WebdavStatus.SC_FORBIDDEN);
            return;
        }

        if (isLocked(req)) {
            resp.sendError(WebdavStatus.SC_LOCKED);
            return;
        }

        String path = getRelativePath(req);

        String lockTokenHeader = req.getHeader("Lock-Token");
        if (lockTokenHeader == null)
            lockTokenHeader = "";

        // Checking resource locks

        LockInfo lock = resourceLocks.get(path);
        Enumeration<String> tokenList = null;
        if (lock != null) {

            // At least one of the tokens of the locks must have been given

            tokenList = lock.tokens.elements();
            while (tokenList.hasMoreElements()) {
                String token = tokenList.nextElement();
                if (lockTokenHeader.indexOf(token) != -1) {
                    lock.tokens.removeElement(token);
                }
            }

            if (lock.tokens.isEmpty()) {
                resourceLocks.remove(path);
                // Removing any lock-null resource which would be present
                lockNullResources.remove(path);
            }

        }

        // Checking inheritable collection locks

        Enumeration<LockInfo> collectionLocksList = collectionLocks.elements();
        while (collectionLocksList.hasMoreElements()) {
            lock = collectionLocksList.nextElement();
            if (path.equals(lock.path)) {

                tokenList = lock.tokens.elements();
                while (tokenList.hasMoreElements()) {
                    String token = tokenList.nextElement();
                    if (lockTokenHeader.indexOf(token) != -1) {
                        lock.tokens.removeElement(token);
                        break;
                    }
                }

                if (lock.tokens.isEmpty()) {
                    collectionLocks.removeElement(lock);
                    // Removing any lock-null resource which would be present
                    lockNullResources.remove(path);
                }

            }
        }

        resp.setStatus(WebdavStatus.SC_NO_CONTENT);

    }
```

3.1.2 isLocked

```
    /**
     * Check to see if a resource is currently write locked.
     *
     * @param path Path of the resource
     * @param ifHeader "If" HTTP header which was included in the request
     * @return boolean true if the resource is locked (and no appropriate
     * lock token has been found for at least one of the non-shared locks which
     * are present on the resource).
     */
    private boolean isLocked(String path, String ifHeader) {

        // Checking resource locks

        LockInfo lock = resourceLocks.get(path);
        Enumeration<String> tokenList = null;
        if (lock != null && lock.hasExpired()) {
            resourceLocks.remove(path);
        } else if (lock != null) {

            // At least one of the tokens of the locks must have been given

            tokenList = lock.tokens.elements();
            boolean tokenMatch = false;
            while (tokenList.hasMoreElements()) {
                String token = tokenList.nextElement();
                if (ifHeader.indexOf(token) != -1)
                    tokenMatch = true;
            }
            if (!tokenMatch)
                return true;

        }

        // Checking inheritable collection locks

        Enumeration<LockInfo> collectionLocksList = collectionLocks.elements();
        while (collectionLocksList.hasMoreElements()) {
            lock = collectionLocksList.nextElement();
            if (lock.hasExpired()) {
                collectionLocks.removeElement(lock);
            } else if (path.startsWith(lock.path)) {

                tokenList = lock.tokens.elements();
                boolean tokenMatch = false;
                while (tokenList.hasMoreElements()) {
                    String token = tokenList.nextElement();
                    if (ifHeader.indexOf(token) != -1)
                        tokenMatch = true;
                }
                if (!tokenMatch)
                    return true;

            }
        }

        return false;

    }
```

3.1.3 copyResource

```
    /**
     * Copy a resource.
     *
     * @param req Servlet request
     * @param resp Servlet response
     * @return boolean true if the copy is successful
     */
    private boolean copyResource(HttpServletRequest req,
                                 HttpServletResponse resp)
        throws ServletException, IOException {

        // Parsing destination header

        String destinationPath = req.getHeader("Destination");

        if (destinationPath == null) {
            resp.sendError(WebdavStatus.SC_BAD_REQUEST);
            return false;
        }

        // Remove url encoding from destination
        destinationPath = RequestUtil.urlDecode(destinationPath, "UTF8");

        int protocolIndex = destinationPath.indexOf("://");
        if (protocolIndex >= 0) {
            // if the Destination URL contains the protocol, we can safely
            // trim everything upto the first "/" character after "://"
            int firstSeparator =
                destinationPath.indexOf("/", protocolIndex + 4);
            if (firstSeparator < 0) {
                destinationPath = "/";
            } else {
                destinationPath = destinationPath.substring(firstSeparator);
            }
        } else {
            String hostName = req.getServerName();
            if (hostName != null && destinationPath.startsWith(hostName)) {
                destinationPath = destinationPath.substring(hostName.length());
            }

            int portIndex = destinationPath.indexOf(":");
            if (portIndex >= 0) {
                destinationPath = destinationPath.substring(portIndex);
            }

            if (destinationPath.startsWith(":")) {
                int firstSeparator = destinationPath.indexOf("/");
                if (firstSeparator < 0) {
                    destinationPath = "/";
                } else {
                    destinationPath =
                        destinationPath.substring(firstSeparator);
                }
            }
        }

        // Normalise destination path (remove '.' and '..')
        destinationPath = RequestUtil.normalize(destinationPath);

        String contextPath = req.getContextPath();
        if (contextPath != null &&
            destinationPath.startsWith(contextPath)) {
            destinationPath = destinationPath.substring(contextPath.length());
        }

        String pathInfo = req.getPathInfo();
        if (pathInfo != null) {
            String servletPath = req.getServletPath();
            if (servletPath != null &&
                destinationPath.startsWith(servletPath)) {
                destinationPath = destinationPath
                    .substring(servletPath.length());
            }
        }

        if (debug > 0)
            log("Dest path :" + destinationPath);

        if (destinationPath.toUpperCase(Locale.ENGLISH).startsWith("/WEB-INF") ||
            destinationPath.toUpperCase(Locale.ENGLISH).startsWith("/META-INF")) {
            resp.sendError(WebdavStatus.SC_FORBIDDEN);
            return false;
        }

        String path = getRelativePath(req);

        if (path.toUpperCase(Locale.ENGLISH).startsWith("/WEB-INF") ||
            path.toUpperCase(Locale.ENGLISH).startsWith("/META-INF")) {
            resp.sendError(WebdavStatus.SC_FORBIDDEN);
            return false;
        }

        if (destinationPath.equals(path)) {
            resp.sendError(WebdavStatus.SC_FORBIDDEN);
            return false;
        }

        // Parsing overwrite header

        boolean overwrite = true;
        String overwriteHeader = req.getHeader("Overwrite");

        if (overwriteHeader != null) {
            if ("T".equalsIgnoreCase(overwriteHeader)) {
                overwrite = true;
            } else {
                overwrite = false;
            }
        }

        // Overwriting the destination

        boolean exists = true;
        try {
            resources.lookup(destinationPath);
        } catch (NamingException e) {
            exists = false;
        }

        if (overwrite) {

            // Delete destination resource, if it exists
            if (exists) {
                if (!deleteResource(destinationPath, req, resp, true)) {
                    return false;
                }
            } else {
                resp.setStatus(WebdavStatus.SC_CREATED);
            }

        } else {

            // If the destination exists, then it's a conflict
            if (exists) {
                resp.sendError(WebdavStatus.SC_PRECONDITION_FAILED);
                return false;
            }

        }

        // Copying source to destination

        Hashtable<String,Integer> errorList = new Hashtable<String,Integer>();

        boolean result = copyResource(resources, errorList,
                                      path, destinationPath);

        if (!result || !errorList.isEmpty()) {

            sendReport(req, resp, errorList);
            return false;

        }

        // Copy was successful
        resp.setStatus(WebdavStatus.SC_CREATED);

        // Removing any lock-null resource which would be present at
        // the destination path
        lockNullResources.remove(destinationPath);

        return true;

    }
```

3.2 Checklist

In this section will be presented the application of the checklist.

3.2.1 Naming Conventions

1. *Meaningful variable, constant, class and methods names:* All the names of variables, methods and classes have meaningful names. Often are used some abbreviations (like "resp" or "req") which are used locally in each method, but it does not influence the readability and understanding of the code.
2. *One-character variables:* In the given methods there are no single-character variables. They are present though in the class, but they are used as temporary variables.
3. *Class names:* All the class names present in the file are written in the correct format.
4. *Interface names:* There are no interfaces used in the given methods.
5. *Method names:* All the methods present in the class are correctly named, except the method "service" at line 365, which is not a verb. It would be better if it is called "getService()".
6. *Class attributes:* All class variables follow the naming conventions.
7. *Constant names:* All the constants follow the naming conventions.

3.2.2 Indention

8. *Spaces for indention:* All the given methods use the indention correctly with the constant use of four spaces.
9. *Use of tabs:* No tabs are used for indention purposes.

3.2.3 Braces

10. *Consistent use of braces style:* In the given code there is a consistent use of the "Kernighan and Ritchie" style.
11. *11. All if, do-while, try-catch have braces even with only one statement:*
    (a) In method "isLocked" there is a violation of the rule at line 1554, 1557, 1575 and 1578. The four if statements are not surrounded with braces.
    (b) In method "doUnlock" there is a violation of the rule at line 1436. The if statement is not surrounded with braces.

   (c) In method “copyResource” there is a violation of the rule at line 1664. The if statement is not surrounded with braces.

3.2.4 File organization

12. *Separation using comments and Blank lines:* There is a good use of blank lines and comments in order to highlight important sections of the code making it more readable.

13. *Line length:*

    (a) In method “copyResource” lines 1667, 1668 exceed the maximum length of 80 columns because of the long condition of the if. These lines do arrive at 83 columns of length, which is still acceptable.

14. *Line length exceeds (>=120):* All the previous lines that exceed the 80 columns limit do not exceed the 120 columns length.

3.2.5 Wrapping lines

15. *Line breaks after comma or operator:* All the line breaks that occur follow the rule.

16. *Higher-level breaks:* No issues found.

17. *Statements alignment:* All the statements are correctly aligned.

3.2.6 Comments

18. *Adequate use of comments:* All the methods include comments which are useful in the understanding of the code.

    (a) Method “doUnlock” has meaningless JavaDoc comment before the declaration which gives no clues on how the method works (line 1418)

    (b) Method “copyResource” has a JavaDoc comment which is too generic and gives no hint on how the method works (line 1590)

19. *Commented code:* There is no commented code.

Chapter 4

Other problem highlighted

Chapter 5

Working hours & other info