Code Inspection

Alessandro Comodi, Davide Coficconi, Stefano Longari
v1.0

January 5, 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 class assigned, as says its own name, is a WebDAV servlet.

A Web Distributed Authoring and Versioning (WebDAV) is an extension of the Hypertext Transfer Protocol (HTTP) that allows clients to perform remote Web content authoring operations. This servlet provides some functionalities thanks to which a user can create, move and change documents on a server. The most important features of the WebDAV protocol include the maintenance of properties about an author or modification date, namespace management, collections, and overwrite protection.

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

```
1417      /**
1418       * UNLOCK Method.
1419       *FROM HERE DAVE
1420      * @param req
1421       *@param resp
1422       *@exception servlet and IO
1423       */
1424      protected void doUnlock(HttpServletRequest req, HttpServletResponse resp)
1425          throws ServletException, IOException { // how it is handled the exception DAVE
1426
1427          if (readOnly) { //Checking the lock as readOnly send error message forbidden DAVE
1428              resp.sendError(WebdavStatus.SC_FORBIDDEN);
1429              return;
1430          }
1431
1432          if (isLocked(req)) { //Checking the lock as the resource is locked DAVE
1433              resp.sendError(WebdavStatus.SC_LOCKED);
1434              return;
1435          }
1436
1437          String path = getRelativePath(req);
1438
1439          String lockTokenHeader = req.getHeader("Lock-Token");
1440          if (lockTokenHeader == null) //Checking the refernces DAVE
1441              lockTokenHeader = "";
1442
1443          // Checking resource locks
1444
1445          LockInfo lock = resourceLocks.get(path);
1446          Enumeration<String> tokenList = null;
1447          if (lock != null) {
1448
1449              // At least one of the tokens of the locks must have been given
1450
1451              tokenList = lock.tokens.elements();
```

```
1452                while (tokenList.hasMoreElements()) {
1453                    String token = tokenList.nextElement();
1454                    if (lockTokenHeader.indexOf(token) != -1) { //find the locked resource then remove from the
                            hash map DAVE
1455                        lock.tokens.removeElement(token);
1456                    }
1457                }
1458
1459                if (lock.tokens.isEmpty()) {
1460                    resourceLocks.remove(path);
1461                    // Removing any lock-null resource which would be present
1462                    lockNullResources.remove(path);
1463                }
1464
1465            }
1466
1467            // Checking inheritable collection locks
1468
1469            Enumeration<LockInfo> collectionLocksList = collectionLocks.elements();
1470            while (collectionLocksList.hasMoreElements()) {
1471                lock = collectionLocksList.nextElement();
1472                if (path.equals(lock.path)) {
1473
1474                    tokenList = lock.tokens.elements();
1475                    while (tokenList.hasMoreElements()) {
1476                        String token = tokenList.nextElement();
1477                        if (lockTokenHeader.indexOf(token) != -1) {
1478                            lock.tokens.removeElement(token);
1479                            break;
1480                        }
1481                    }
1482
1483                    if (lock.tokens.isEmpty()) {
1484                        collectionLocks.removeElement(lock);
1485                        // Removing any lock-null resource which would be present
1486                        lockNullResources.remove(path);
1487                    }
1488
1489                }
1490            }
1491
1492            resp.setStatus(WebdavStatus.SC_NO_CONTENT);
1493
1494        }
```

3.1.2 isLocked

```
1529        /**
1530         * Check to see if a resource is currently write locked.
1531         *
1532         * @param path Path of the resource
1533         * @param ifHeader "If" HTTP header which was included in the request
1534         * @return boolean true if the resource is locked (and no appropriate
1535         * lock token has been found for at least one of the non-shared locks which
1536         * are present on the resource).
1537         */
1538        private boolean isLocked(String path, String ifHeader) {
1539
1540            // Checking resource locks
1541
1542            LockInfo lock = resourceLocks.get(path);
1543            Enumeration<String> tokenList = null;
1544            if (lock != null && lock.hasExpired()) {
1545                resourceLocks.remove(path);
1546            } else if (lock != null) {
1547
1548                // At least one of the tokens of the locks must have been given
1549
1550                tokenList = lock.tokens.elements();
1551                boolean tokenMatch = false;
1552                while (tokenList.hasMoreElements()) {
1553                    String token = tokenList.nextElement();
1554                    if (ifHeader.indexOf(token) != -1)
1555                        tokenMatch = true;
1556                }
1557                if (!tokenMatch)
1558                    return true;
1559
1560            }
1561
1562            // Checking inheritable collection locks
1563
1564            Enumeration<LockInfo> collectionLocksList = collectionLocks.elements();
1565            while (collectionLocksList.hasMoreElements()) {
1566                lock = collectionLocksList.nextElement();
1567                if (lock.hasExpired()) {
1568                    collectionLocks.removeElement(lock);
1569                } else if (path.startsWith(lock.path)) {
```

```
1570
1571                tokenList = lock.tokens.elements();
1572                boolean tokenMatch = false;
1573                while (tokenList.hasMoreElements()) {
1574                    String token = tokenList.nextElement();
1575                    if (ifHeader.indexOf(token) != -1)
1576                        tokenMatch = true;
1577                }
1578                if (!tokenMatch)
1579                    return true;
1580
1581            }
1582        }
1583
1584        return false;
1585
1586    }
```

3.1.3 copyResource

```
1589    /**
1590     * Copy a resource.
1591     *
1592     * @param req Servlet request
1593     * @param resp Servlet response
1594     * @return boolean true if the copy is successful
1595     */
1596    private boolean copyResource(HttpServletRequest req,
1597                                 HttpServletResponse resp)
1598        throws ServletException, IOException {
1599
1600        // Parsing destination header
1601
1602        String destinationPath = req.getHeader("Destination"); //check no conversion DAVE
1603
1604        if (destinationPath == null) { //error handling DAVE
1605            resp.sendError(WebdavStatus.SC_BAD_REQUEST);
1606            return false;
1607        }
1608
1609        // Remove url encoding from destination
1610        destinationPath = RequestUtil.urlDecode(destinationPath, "UTF8"); //check no conversion DAVE
1611
1612        int protocolIndex = destinationPath.indexOf("://");
1613        if (protocolIndex >= 0) {
1614            // if the Destination URL contains the protocol, we can safely
1615            // trim everything upto the first "/" character after "://"
1616            int firstSeparator =
1617                destinationPath.indexOf("/", protocolIndex + 4);
1618            if (firstSeparator < 0) {
1619                destinationPath = "/";
1620            } else {
1621                destinationPath = destinationPath.substring(firstSeparator);
1622            }
1623        } else { //destination url not contains protocol DAVE
1624            String hostName = req.getServerName();
1625            if (hostName != null && destinationPath.startsWith(hostName)) {
1626                destinationPath = destinationPath.substring(hostName.length());
1627            }
1628
1629            int portIndex = destinationPath.indexOf(":");
1630            if (portIndex >= 0) {
1631                destinationPath = destinationPath.substring(portIndex);
1632            }
1633
1634            if (destinationPath.startsWith(":")) {
1635                int firstSeparator = destinationPath.indexOf("/");
1636                if (firstSeparator < 0) {
1637                    destinationPath = "/";
1638                } else {
1639                    destinationPath =
1640                        destinationPath.substring(firstSeparator);
1641                }
1642            }
1643        }
1644
1645        // Normalise destination path (remove '.' and '..')
1646        destinationPath = RequestUtil.normalize(destinationPath);
1647        //get the context path and if the destinationPath starts with the context path replace the
              destinationPath with a substring of context path. length DAVE
1648        String contextPath = req.getContextPath();
1649        if (contextPath != null &&
1650            destinationPath.startsWith(contextPath)) {
1651            destinationPath = destinationPath.substring(contextPath.length());
1652        }
1653        //destinationPath work with a substring of servletPath.length DAVE
1654        String pathInfo = req.getPathInfo();
1655        if (pathInfo != null) {
```

```
1656                String servletPath = req.getServletPath();
1657                if (servletPath != null &&
1658                    destinationPath.startsWith(servletPath)) {
1659                    destinationPath = destinationPath
1660                        .substring(servletPath.length());
1661                }
1662            }
1663            //logger DAVE how it works up to here the normalization of the destination path DAVE
1664            if (debug > 0)
1665                log("Dest path :" + destinationPath);
1666            //error case   for the destination path and the path of the request web inf or meta inf DAVE
1667            if (destinationPath.toUpperCase(Locale.ENGLISH).startsWith("/WEB-INF") ||
1668                destinationPath.toUpperCase(Locale.ENGLISH).startsWith("/META-INF")) {
1669                resp.sendError(WebdavStatus.SC_FORBIDDEN);
1670                return false;
1671            }
1672
1673            String path = getRelativePath(req);
1674
1675            if (path.toUpperCase(Locale.ENGLISH).startsWith("/WEB-INF") ||
1676                path.toUpperCase(Locale.ENGLISH).startsWith("/META-INF")) {
1677                resp.sendError(WebdavStatus.SC_FORBIDDEN);
1678                return false;
1679            }
1680            //error handling in case of equality of the request's path and the destinationPath DAVE
1681            if (destinationPath.equals(path)) {
1682                resp.sendError(WebdavStatus.SC_FORBIDDEN);
1683                return false;
1684            }
1685
1686            // Parsing overwrite header
1687
1688            boolean overwrite = true;
1689            String overwriteHeader = req.getHeader("Overwrite");
1690
1691            if (overwriteHeader != null) {
1692                if ("T".equalsIgnoreCase(overwriteHeader)) {
1693                    overwrite = true;
1694                } else {
1695                    overwrite = false;
1696                }
1697            }
1698
1699            // Overwriting the destination
1700            //try to lookup for the resource in the destinationPath if it was to overwrite and already exists
                  try to delete DAVE
1701            boolean exists = true;
1702            try {
1703                resources.lookup(destinationPath);
1704            } catch (NamingException e) {
1705                exists = false;
1706            }
1707
1708            if (overwrite) {
1709
1710                // Delete destination resource, if it exists
1711                if (exists) {
1712                    if (!deleteResource(destinationPath, req, resp, true)) {
1713                        return false;
1714                    }
1715                } else {
1716                    resp.setStatus(WebdavStatus.SC_CREATED);
1717                }
1718
1719            } else {
1720
1721                // If the destination exists, then it's a conflict
1722                if (exists) {
1723                    resp.sendError(WebdavStatus.SC_PRECONDITION_FAILED);
1724                    return false;
1725                }
1726
1727            }
1728
1729            // Copying source to destination
1730
1731            Hashtable<String,Integer> errorList = new Hashtable<String,Integer>();
1732
1733            boolean result = copyResource(resources, errorList,
1734                                          path, destinationPath);
1735            //if something goes wrong DAVE
1736            if (!result || !errorList.isEmpty()) {
1737
1738                sendReport(req, resp, errorList);
1739                return false;
1740
1741            }
1742
1743            // Copy was successful
1744            resp.setStatus(WebdavStatus.SC_CREATED);
1745
```

```
1746            // Removing any lock-null resource which would be present at
1747            // the destination path
1748            lockNullResources.remove(destinationPath);
1749
1750            return true;
1751
1752        }
```

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. *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.

3.2.7 Object Comparison

20. *Check that all objects (including Strings) are compared with "equals" and not with "==":* All comparison occur in the right way.

    (a) Method “doUnlock” has a comparison with “==”, but it is a check if the object is instantiated (line 1436)

    (b) Method “copyResource” has a comparison with “==”, but it is a check if the object is instantiated (line 1604)

3.2.8 Output Format

21. *Check that displayed output is free of spelling and grammatical errors:* There is not output displayed in the methods assigned

22. *Check that error messages are comprehensive and provide guidance as to how to correct the problem:* There is no error messages in the methods assigned

23. *Check that the output is formatted correctly in terms of line stepping and spacing:* There is no output and neither error on formatting the output in the methods assigned

3.2.9 Computation, Comparisons and Assignments

24. *Check that the implementation avoids “brutish programming: (see http://users.csc.calpoly.edu/~jdalbey/S*
    There are no “brutish programming” in the implementation

25. *Check order of computation/evaluation, operator precedence and parenthesizing:* The order of computation, or evaluation, and operator precedence and parethesizing happen in the right way

26. *Check the liberal use of parenthesis is used to avoid operator precedence problems:* The use of parethesis occurs in a right manner in the methods assigned

27. *Check that all denominators of a division are prevented from being zero:* There are neither denominators possibly equal to zero, neither division in the methods assigned

28. *Check that integer arithmetic, especially division, are used appropriately to avoid causing unexpected truncation/rounding:* There are no integer arithmetic in the methods assigned

29. *Check that the comparison and Boolean operators are correct:* All comparisons and Boolean operators are correct in the methods assigned

30. *Check throw-catch expressions, and check that the error condition is actually legitimate:* There is only one throw catch expression and the error condition is legitimate

    (a) In method "copyResource" the catch expression works because if there is no resource in the destination path, it hasn't to delete nothing (line 1701)

31. *Check that the code is free of any implicit type conversions:* There aren't conversions in the code, only decodification of something or getter

3.2.10 Exceptions

32. *Check that the relevant exceptions are caught:* All relevant exception in the code are caught

33. *Check that the appropriate action are taken for each catch block:* For each catch block an appropriate action is taken

    (a) In method "copyResource" if a resource doesn't exist, the catch allows to take into account that there isn't need of deletion (line 1701)

3.2.11 Flow of Control

34. I*n a switch statement, check that all cases are addressed by break or return:* There aren't switch in the methods assigned

35. *Check that all switch statements have a default branch:* There aren't switch in the methods assigned

36. *Check that all loops are correctly formed, with the appropriate initialization, increment and termination expressions:* All loops are correctly formed

    (a) In method "doUnlock" there are three loops and all are correctly formed (lines 1447,1466,1471)

    (b) In method "isLocked" there are three loops and all are correctly formed (lines 1552,1565,1573)

3.2.12 Files

37. *Check that all files are properly declared and opened:* There are no files in the methods assigned

38. *Check that all files are closed properly, even in the case of an error:* There are no files in the methods assigned

39. *Check that EOF conditions are detected and handled correctly:* There are no files in the methods assigned

40. *Check that all file exceptions are caught and dealt with accordingly:* There are no files in the methods assigned

Chapter 4

Other problem highlighted

Chapter 5

Working hours & other info