SOFTWARE ENGINEERING 2 PROJECT

Code Inspection

Assignment 3

Amos Paribocci (854818); Lorenzo Pinosa (852231)

Version 1 – 03/01/2015

Table of contents

1. Introduction and assigned classes

1.1 Document introduction and structure

This document aims to present the results of our inspection of the assigned code, coming from a release of the Java EE Server Glassfish 4.1, following an issue checklist.

Paragraph 1.2 describes the assigned classes, listing the method names, start lines and locations.

In paragraph 1.3 the functional role of the assigned classes is explained, in the limits of what has been assigned to us.

Section 2 represents the main part of the document: for each issue category, a list of the found problems is given. Other relevant information about the analysis is described in Section 3.

1.2 Assigned classes

The following methods, all part of the class `JDBCStore`, have been assigned to us:

1.2.1 save(Session session)

- Name: `save(Session session)`
- Start line: 747
- Location: *appserver/web/web-core/src/main/java/org/apache/catalina/session/JDBCStore.java*

```
/**
 * Save a session to the Store.
 *
 * @param session the session to be stored
 * @exception IOException if an input/output error occurs
 */
public void save(Session session) throws IOException {
    String saveSql =
        "INSERT INTO " + sessionTable + " (" + sessionIdCol + ", " +
        sessionAppCol + ", " +
        sessionDataCol + ", " +
        sessionValidCol + ", " +
        sessionMaxInactiveCol + ", " +
        sessionLastAccessedCol + ") VALUES (?, ?, ?, ?, ?, ?)";
    ObjectOutputStream oos = null;
    ByteArrayOutputStream bos = null;
    ByteArrayInputStream bis = null;
    InputStream in = null;

    synchronized(this) {
        Connection _conn = getConnection();
        if(_conn == null) {
            return;
        }

        // If sessions already exist in DB, remove and insert again.
        // TODO:
        // * Check if ID exists in database and if so use UPDATE.
        remove(session.getIdInternal());
```

```
        try {
            bos = new ByteArrayOutputStream();
            oos = new ObjectOutputStream(new BufferedOutputStream(bos));

            oos.writeObject(session);
            oos.close();

            byte[] obs = bos.toByteArray();
            int size = obs.length;
            bis = new ByteArrayInputStream(obs, 0, size);
            in = new BufferedInputStream(bis, size);

            if(preparedSaveSql == null) {
                preparedSaveSql = _conn.prepareStatement(saveSql);
            }

            preparedSaveSql.setString(1, session.getIdInternal());
            preparedSaveSql.setString(2, getName());
            preparedSaveSql.setBinaryStream(3, in, size);
            preparedSaveSql.setString(4, session.isValid()?"1":"0");
            preparedSaveSql.setInt(5, session.getMaxInactiveInterval());
            preparedSaveSql.setLong(6, session.getLastAccessedTime());
            preparedSaveSql.execute();
        } catch(SQLException e) {
            String msg = MessageFormat.format(rb.getString(SQL_ERROR),
                                              e);
            log(msg);
        } catch (IOException e) {
            // Ignore
        } finally {
            if (oos != null) {
                oos.close();
            }
            if(bis != null) {
                bis.close();
            }
            if(in != null) {
                in.close();
            }

            release(_conn);
        }
    }

    if (debug > 0) {
        String msg = MessageFormat.format(rb.getString(SAVING_SESSION),
                                          new Object[]
{session.getIdInternal(), sessionTable});
        log(msg);
    }
}
```

1.2.2 getConnection()

- Name: `getConnection()`
- Start line: 831
- Location: *appserver/web/web-core/src/main/java/org/apache/catalina/session/JDBCStore.java*

```
/**
 * Check the connection associated with this store, if it's
 * <code>null</code> or closed try to reopen it.
 * Returns <code>null</code> if the connection could not be established.
 *
 * @return <code>Connection</code> if the connection succeeded
 */
protected Connection getConnection(){
    try {
        if(conn == null || conn.isClosed()) {
            Class.forName(driverName);
            String databaseConnClosedMsg =
rb.getString(DATABASE_CONNECTION_CLOSED);
            log(databaseConnClosedMsg);
            conn = DriverManager.getConnection(connString);
            conn.setAutoCommit(true);

            if(conn == null || conn.isClosed()) {
                String openDatabaseFailedMsg =
rb.getString(RE_OPEN_DATABASE_FAILED);
                log(openDatabaseFailedMsg);
            }
        }
    } catch (SQLException ex){
        String msg = MessageFormat.format(rb.getString(SQL_EXCEPTION),
                                          ex.toString());
        log(msg);
    } catch (ClassNotFoundException ex) {
        String msg =
MessageFormat.format(rb.getString(JDBC_DRIVER_CLASS_NOT_FOUND),
                                          ex.toString());
        log(msg);
    }

    return conn;
}
```

1.2.3 stop()

- Name: `stop()`
- Start line: 883
- Location: *appserver/web/web-core/src/main/java/org/apache/catalina/session/JDBCStore.java*

```
/**
 * Gracefully terminate everything associated with our db.
 * Called once when this Store is stoping.
 *
 */
```

```
public void stop() throws LifecycleException {
    super.stop();

    // Close and release everything associated with our db.
    if(conn != null) {
        try {
            conn.commit();
        } catch (SQLException e) {
            // Ignore
        }

        if( preparedSizeSql != null ) {
            try {
                preparedSizeSql.close();
            } catch (SQLException e) {
                // Ignore
            }
        }

        if( preparedKeysSql != null ) {
            try {
                preparedKeysSql.close();
            } catch (SQLException e) {
                // Ignore
            }
        }

        if( preparedSaveSql != null ) {
            try {
                preparedSaveSql.close();
            } catch (SQLException e) {
                // Ignore
            }
        }

        if( preparedClearSql != null ) {
            try {
                preparedClearSql.close();
            } catch (SQLException e) {
                // Ignore
            }
        }

        if( preparedRemoveSql != null ) {
            try {
                preparedRemoveSql.close();
            } catch (SQLException e) {
                // Ignore
            }
        }

        if( preparedLoadSql != null ) {
            try {
                preparedLoadSql.close();
            } catch (SQLException e) {
                // Ignore
            }
```

```
        }

        try {
            conn.close();
        } catch (SQLException e) {
            // Ignore
        }

        this.preparedSizeSql = null;
        this.preparedKeysSql = null;
        this.preparedSaveSql = null;
        this.preparedClearSql = null;
        this.preparedRemoveSql = null;
        this.preparedLoadSql = null;
        this.conn = null;
    }
}
```

1.3 Functional role of the assigned classes

All methods that were assigned to us are part of a single class: JDBCStore.

This class is an ORM (Object Relational Mapper) for `Session` objects: it is a facility to store, retrieve, edit and delete `Session` objects in the database, connected through JDBC (Java Database Connectivity).

In particular, `getConnection()` sets up the connection with the database by opening it, setting flags and checking if everything went on correctly (and eventually throwing an exception); `save(Session session)` actually saves a session in the database by serializing it; stop() finalizes an usage of the class by committing changes to the database and closing the connection.

To understand this, it was enough to read the code documentation comments and the code itself. The main things that pointed out the class goal were in particular: the comment at class level, the methods and class names, fragments of SQL code in the save method.

1.4 References

- "Assignment 3 – Code Inspection" document, provided in the Software Engineering 2 course context.

2. Found issues

2.1 Naming conventions

2.1.1 1 - All class names, interface names, method names, class variables, method variables, and constants used should have meaningful names and do what the name suggests

Class `JDBCStore`:

a) Line 86: even if the variable name `rb` does not indicate its use, the meaning is clear when reading the variable type, which is called `ResourceBundle`;
b) Line 139: instead of using the name `info` for the variable, a more meaningful name should be used;
c) Line 144: giving a variable the name `name` does not clarify the entity which the variable refers to; instead of explaining the meaning in a comment above the declaration, a more representative name should be used.

Method `save`:

a) Lines 755, 756, 757: the names of the variables `oos`, `bos` and `bis` do not directly suggest their goal, but they represent their variable type (`ObjectOutputStream`, `ByteArrayOutputStream` and `ByteArrayInputStream` respectively) well;
b) Line 778: the name of the variable `obs` does not suggest its meaning.

2.1.2 2 - If one-character variables are used, they are used only for temporary "throwaway" variables, such as those used in for loops

No issues were found: the only single-character variables are used in the method `stop` for exception-handling purposes.

2.1.3 3 - Class names are nouns, in mixed case, with the first letter of each word in capitalized

No issues were found: the correct naming convention is followed.

2.1.4 4 - Interface names should be capitalized like classes

No interface was assigned to our group.

2.1.5 5 - Method names should be verbs, with the first letter of each addition word capitalized

Method `save`:

a) Line 778: the method name `toByteArray` is not a verb, but we still think the name is appropriate since such name is commonly used. That is why we will not report analogous cases again.

Method `getConnection`:

a) Line 834: the method name `forName` is not a verb.

2.1.6 6 - Class variables, also called attributes, are mixed case, but might begin with an underscore ('_') followed by a lowercase first letter. All the remaining words in the variable name have their first letter capitalized

No issues were found. All class variables were named correctly.

2.1.7 7 - Constants are declared using all uppercase with words separated by an underscore

No issues were found: every constant was named in uppercase.

2.2 Indention

2.2.1 8 - Three or four spaces are used for indentation and done so consistently

Method `save`:

a) Line 796: this line's content, which consists in continuing the previous line's statement, is aligned with the previous line's method bracket. That is not consistent with the four spaces usage chosen for the code. The same problem applies to lines 817, 847 and 851.

2.2.2 9 - No tabs are used to indent

No issues were found: indentation is done by using spaces.

2.3 Braces

2.3.1 10 - Consistent bracing style is used, either the preferred "Allman" style (first brace goes underneath the opening block) or the "Kernighan and Ritchie" style (first brace is on the same line of the instruction that opens the new block)

No issues were found: in the assigned code, "Kernighan and Ritchie" style is used.

2.3.2 11 - All if, while, do-while, try-catch, and for statements that have only one statement to execute are surrounded by curly braces

No issues were found: curly braces were put regardless of the number of statements in the blocks.

2.4 File organization

2.4.1 12 - Blank lines and optional comments are used to separate sections (beginning comments, package/import statements, class/interface declarations which include class variable/attributes declarations, constructors, and methods)

Blank lines were used when appropriate: no issues were found.

2.4.2 13 - Where practical, line length does not exceed 80 characters

From a quick scan of the class, it is easy to see that the maximum line length is 106 characters (at line 113). As far as the assigned methods are concerned, the maximum length violation all occurs in the method `getConnection` (at lines 817, 835, 841 and 850).

2.4.3 14 - When line length must exceed 80 characters, it does NOT exceed 120 characters

No issues were found: the maximum line length is 106 characters.

2.5 Wrapping lines

2.5.1 15 - Line break occurs after a comma or an operator

No issues were found: line breaks were inserted after commas or operators only.

2.5.2 16 - Higher-level breaks are used

No issues were found – higher-level breaks are not used in the code.

2.5.3 17 - A new statement is aligned with the beginning of the expression at the same level as the previous line

No issues were found: statements were aligned correctly.

2.6 Comments

2.6.1 18 - Comments are used to adequately explain what the class, interface, methods, and blocks of code are doing

Yes: they were even enough to understand class goal and behavior without any further information or reference.

2.6.2 19 - Commented out code contains a reason for being commented out and a date it can be removed from the source file if determined it is no longer needed

No issues were found: the assigned methods do not contain commented-out code.

2.7 Java source files

2.7.1 20 - Each Java source file contains a single public class or interface

No issues were found: the only file related to our assignment only contains the public class `JDBCStore`.

2.7.2 21 - The public class is the first class or interface in the file

Our file contains one (and only one) public class, thus no issues were found.

2.7.3 22 - Check that the external program interfaces are implemented consistently with what is described in the javadoc

No issues found. All public methods do exactly what the description in the Javadoc-formatted comment says they do.

2.7.4 23 - Check that the javadoc is complete (i.e., it covers all classes and files part of the set of classes assigned to you)

No issues were found: we think the javadoc is complete. All the methods that feature parameters and/or return values use the relative javadoc notation.

2.8 Package and import statements

2.8.1 24 - If any package statements are needed, they should be the first non-comment statements. Import statements follows

No issues were found: the correct structure is followed.

2.9 Class and interface declarations

2.9.1 25 - The class or interface declarations shall follow the correct order (see assignment document for the rules)

a) The correct static variables declaration order, which is described by rule D, is violated: line 86 contains a `private static` variable declaration, line 92 contains a `public static` variable declaration and line 139 contains a `protected static` variable declaration;
b) Not all the instance variables follow the static variables (rules D, E): line 139 contains a `static` variable declaration, line 144 contains an instance variable declaration and line 149 contains a `static` variable declaration;
c) The correct instance variables declaration order, which is described by rule E, is violated: line 144 contains a `private` variable declaration, line 154 contains a `protected` variable declaration, line 164 contains a `private` variable declaration and line 169 contains a `protected` variable declaration.

2.9.2 26 - Methods are grouped by functionality rather than by scope or accessibility

The class features properties methods first (setters and getters), then more complex methods follow.

We consider this a good grouping method.

2.9.3 27 - Check that the code is free of duplicates, long methods, big classes, breaking encapsulation, as well as if coupling and cohesion are adequate

`getConnection()` and `save(Session session)` code is free from duplicates, but `stop()` method does the same thing (closing) for all prepared statements with similar code: maybe it would be better to add all the variables to a list and then close them using a “for” loop.

Methods are not too long and do only what their name states.

JDBCStore is not too big and encapsulates only one purpose; and does not couple strongly with another class.

2.10 Initialization and declarations

2.10.1 28 - Check that variables and class members are of the correct type. Check that they have the right visibility (public/private/protected)

No issues found.

All class variables that should be visible to all classes (subclasses, classes which use JDBCStore) are marked as public; variables that should be visible only to subclasses are declared as protected, and all others are private. In our opinion the subdivision of class variables among these three visibility levels is good, as well as the type chosen for the variables.

2.10.2 29 - Check that variables are declared in the proper scope

No issues found. `stop()` does not declare any variable. `save(Session session)` declares correctly the `Stream` variables out of the try/catch block because it needs them in finally block; they could have been declared inside the synchronized block but in our opinion this choice improves readability. `getConnection()` declares only string variables to hold error messages and does it in the smallest scope as possible (catch blocks).

2.10.3 30 - Check that constructors are called when a new object is desired

Constructors are correctly called when a new object is desired (for example: method `save`, lines 772 and 773). The assigned class, though, does not feature an explicit constructor.

2.10.4 31 - Check that all object references are initialized before use

No issues were found: object references, before being used, are always initialized.

2.10.5 32 - Variables are initialized where they are declared, unless dependent upon a computation

No issues were found in methods getConnection() and stop(), instead in save(Session session) method variables oos, bos, bis, in are declared and set to null. Nevertheless, in our opinion this is not an issue, because these variable are needed out of the try block scope, in particular in the finally block.

2.10.6 33 - Declarations appear at the beginning of blocks. The exception is a variable can be declared in a 'for' loop

Method `save`:

a) Line 778: variable `obs` is not declared at the beginning of a block;
b) Line 779: variable `size` is not declared at the beginning of a block.

Method `getConnection`:

a) Line 835: variable `databaseConnClosedMsg` is not declared at the beginning of a block.

2.11 Method calls

2.11.1 34 - Check that parameters are presented in the correct order

As far as the assigned class is concerned, we could not find any issue: method calls feature the correct parameters order.

2.11.2 35 - Check that the correct method is being called, or should it be a different method with a similar name

No issues were found: no methods having similar names are present in the assigned class.

2.11.3 36 - Check that method returned values are used properly

a) Assigned methods:
   Out of all the assigned methods, `getConnection()` is the only one that actually returns a value. The method is called by several of the class methods, for example in method `save` (at line 761). We think the values returned by the calls are used according to the functional role of the method itself (see section 1.3 for further details).
b) Method calls inside the assigned methods:
   Most of the methods called in the assigned functions are not part of the assignment (or even declared outside of the assigned file): that is why, in this context, our analysis can only be partial. However, regarding the remaining methods (like the ones provided by the Java standard library), we did not find any error in the use of the returned values.

2.12 Arrays

2.12.1 37 - Check that there are no off-by-one errors in array indexing (that is, all required array elements are correctly accessed through the index)

No issues found. Arrays are not used at all in our methods, except for byte arrays used to support object serialization in `save(Session session)` method - but they aren't supposed to be accessed using an index.

2.12.2 38 - Check that all array (or other collection) indexes have been prevented from going out-of-bounds

No issues found for the same reason referred to in the previous point.

2.12.3 39 - Check that constructors are called when a new array item is desired

No issues found. New arrays are never created.

2.13 Object comparison

2.13.1 40 - Check that all objects (including Strings) are compared with "equals" and not with "=="

No issues found. "==" is only used when comparing to null.

2.14 Output format

2.14.1 41 - Check that displayed output is free of spelling and grammatical errors

No issues found. None of the methods produce any output.

2.14.2 42 - Check that error messages are comprehensive and provide guidance as to how to correct the problem

No issues found. None of the methods produce any error output.

2.14.3 43 - Check that the output is formatted correctly in terms of line stepping and spacing

No issues found, for the same reason exposed in 2.14.1.

2.15 Computation, comparisons and assignments

2.15.1 44 - Check that the implementation avoids "brutish programming: (see this link)

No patterns like (or similar to) those described in the "brutish programming" document were found.

2.15.2 45 - Check order of computation/evaluation, operator precedence and parenthesizing

No issues found. No multiple operators are used in a way that might produce ambiguity and parenthesis are used correctly.

2.15.3 46 - Check the liberal use of parenthesis is used to avoid operator precedence problems

No issues found, as described in the previous point.

2.15.4 47 - Check that all denominators of a division are prevented from being zero

No divisions are made at all in the code.

2.15.5 48 - Check that integer arithmetic, especially division, are used appropriately to avoid causing unexpected truncation/rounding

No mathematical operations are done at all in the code.

2.15.6 49 - Check that the comparison and Boolean operators are correct

a) Method `save(Session session)`:
   Many of the comparisons performed here consist in `null` value checking in order to safely access to object methods (lines 801, 804, 807).
b) Method `getConnection()`:
   Similarly to what happens in method `save`, the comparisons here are used to avoid `NullPointerExceptions` when accessing to object methods (lines 833, 840).
   In fact, in this if clause
   "`if(conn == null || conn.isClosed())`",

the second part of the OR is evaluated only when the first one is false, thus ensuring the call is performed safely.

c) Method `stop()`:
Exactly the same situation described for method `save` applies here (lines 887, 894, 902, 910, 918, 926, 934).

2.15.7 50 - Check throw-catch expressions, and check that the error condition is actually legitimate

No issues were found: when try-catch blocks are used in the assigned methods, the exceptions being caught are legit, since the code inside the try blocks could actually might throw them.

2.15.8 51 - Check that the code is free of any implicit type conversions

No issues found.

2.16 Exceptions

2.16.1 52 - Check that the relevant exceptions are caught

No issues found. All possibly thrown exception are caught.

2.16.2 53 - Check that the appropriate action are taken for each catch block

Some exceptions are logged, but some else - in methods `save(Session session)` and `stop()` are completely ignored! Even simple logging, at debug or verbose level, would be better.

2.17 Flow of control

2.17.1 54 - In a switch statement, check that all cases are addressed by break or return

No switch statements are used at all.

2.17.2 55 - Check that all switch statements have a default branch

No switch statements are used at all.

2.17.3 56 - Check that all loops are correctly formed, with the appropriate initialization, increment and termination expressions

No loops are used at all.

2.18 Files

2.18.1 57 - Check that all files are properly declared and opened

No files are used at all in the code. Streams are used, though, and they are always correctly declared (using the correct subclass) and opened. SQL connection is correctly declared and opened, too.

2.18.2 58 - Check that all files are closed properly, even in the case of an error

No files are used at all in the code. Streams are used, though, and they are always correctly closed in the finally part of the try-catch block. Even the SQL connections and statements are correctly closed in the `stop()` method, which is supposed to be called properly.

2.18.3 59 - Check that EOF conditions are detected and handled correctly

No EOF condition could be ever reached, since no file scans are performed.

2.18.4 60 - Check that all file exceptions are caught and dealt with accordingly

No issues were found - this was already checked as part of 2.16.1 point.

3. Additional highlighted problems

We believe that the checklist used in section 2 shows and explains all the highlighted problems.

No supplementary information is needed.

4. Appendix

4.1 Hours of work

This is the time spent in order to redact this document:

- Amos Paribocci: 5 hours;
- Lorenzo Pinosa: 5 hours.