GlassFish Appserver Parent Project 4.1.1 API

Code inspection

Belluschi Marco 791878, Cerri Stefano 849945, Di Febbo Francesco 852389

January 10, 2016

Revision

In the following are listed the differences between versions:

1. First version
2. Document's title changed

Contents

Chapter 1

Introduction

1.1 Modules inspected

Location
appserver/web/web-core/src/main/java/org/apache/catalina/session/**StandardManager.java**

Methods

- setContainer(Container container) @ 253
- load() @ 435
- doLoadFromFile() @ 465
- readSessions(InputStream is) @ 519
- unload(boolean doExpire , boolean isShutdown) @ 638
- doUnload(boolean doExpire , boolean isShutdown) @ 665

1.2 Functional role

The class inspected is **StandardManager**.
It belongs to the package *org.apache.catalina.session*
The class inheritance is the following:

```
java.lang.Object
    org.apache.catalina.session.ManagerBase
        org.apache.catalina.session.StandardManager
```

Implemented interfaces are:

```
PropertyChangeListener, EventListener, Lifecycle, Manager
```

```
79  /**
80  * Standard implementation of the <b>Manager</b> interface that provides
81  * simple session persistence across restarts of this component (such as
82  * when the entire server is shut down and restarted, or when a particular
83  * web application is reloaded.
84  * <p>
85  * <b>IMPLEMENTATION NOTE</b>: Correct behavior of session storing and
86  * reloading depends upon external calls to the <code>start()</code> and
87  * <code>stop()</code> methods of this class at the correct times.
88  *
89  * @author Craig R. McClanahan
90  * @author Jean-Francois Arcand
91  * @version $Revision: 1.14.6.2 $ $Date: 2008/04/17 18:37:20 $
92  */
```

Description *StandardManager* inherits from *ManagerBase*, a minimal implementation of the *Manager* interface that supports no session persistence or distributable capabilities. On the contrary, *StandardManager* manages the persistence of sessions when occurs start or stop events on the component. *Lifecycle* interface is responsible for component lifecycle methods. Components may implement this interface in order to provide a consistent mechanism to start and stop the components. *PropertyChangeListener* interface is responsible for *PropertyChange* events handling whenever a bean changes a "bound" property.

setContainer(Container container)

```
245 /**
246 * Set the Container with which this Manager has been associated. If
247 * it is a Context (the usual case), listen for changes to the session
248 * timeout property.
249 *
250 * @param container The associated Container
251 */
252 @Override
253 public void setContainer(Container container)
```

Description This method overrides *ManagersBase*'s *setContainer* method. In particular, it de-registers from any pre-existing, *Context Container* through the *removePropertyChangeListener* method. In any case, the method then proceeds calling the superclass *setContainer* method. Finally, if the *Container* argument that was passed is a non-null *Context*, the method registers with it through the *addPropertyChangeListener* method.

load( )

```
426 /**
427 * Loads any currently active sessions that were previously unloaded
428 * to the appropriate persistence mechanism, if any. If persistence is not
429 * supported, this method returns without doing anything.
430 *
431 * @exception ClassNotFoundException if a serialized class cannot be
432 * found during the reload
433 * @exception IOException if a read error occurs
434 */
435 public void load() throws ClassNotFoundException, IOException
```

Description This method requires persistence support (verifying it through the *SecurityUtil.isPackageProtectionEnabled()* method), otherwise simply calls *doLoadFromFile*. The method looks for and loads all previously unloaded active sessions: it tries the *AccessController.doPrivileged(PrivilegedExceptionAction⟨T⟩)* method, catching exceptions *ClassNotFoundException*, *IOException* and reporting all other unexpected ones.

doLoadFromFile( )

```
457 /**
458 * Loads any currently active sessions that were previously unloaded
459 * to file
460 *
461 * @exception ClassNotFoundException if a serialized class cannot be
462 * found during the reload
463 * @exception IOException if a read error occurs
464 */
465 private void doLoadFromFile() throws ClassNotFoundException, IOException
```

Description This method loads persisted session from an existing file. If the file returned by the *file()* method doesn't exist, is empty or is null the method returns and no operations are performed. Otherwise a *FileInputSession fis* is open with parameter the *absolutePath* of the file. If no exceptions are caught the method invokes the *readSession(fis)* otherwise a error message is shown by the logger. In any case the method closes the *FileInputStream* and if there are no other exceptions the method deletes the file.

readSessions(InputStream is)

```
509 /*
510 * Reads any sessions from the given input stream, and initializes the
511 * cache of active sessions with them.
512 *
513 * @param is the input stream from which to read the sessions
514 *
515 * @exception ClassNotFoundException if a serialized class cannot be
516 * found during the reload
517 * @exception IOException if a read error occurs
518 */
519 public void readSessions(InputStream is)
520 throws ClassNotFoundException, IOException
```

Description This method reads any session from the parameter *is* (*InputStream* object). First of all the method initializes the class variable sessions. Then, the method, creates a *ObjectInputStream ois* and try to initialize it with a *BufferedInputStream bis* from the *InputStream is* (the parameter of the method). If an *IOException* is caught the method shows an error through the logger and then try to close the *ObjectInputStream ois* created. Then the class variable sessions is synchronized and the method try to load the persisted sessions and create a object session from the class *StardardSession* and put it in the sessions class variable and activates it. In any case the method closes the *ObjectInputStream* and terminates.

unload(boolean doExpire, boolean isShutDown)

```
626 /**
627 * Save any currently active sessions in the appropriate persistence
628 * mechanism, if any. If persistence is not supported, this method
629 * returns without doing anything.
630 *
631 * @doExpire true if the unloaded sessions are to be expired, false
632 * otherwise
633 * @param isShutdown true if this manager is being stopped as part of a
634 * domain shutdown (as opposed to an undeployment), and false otherwise
635 *
636 * @exception IOException if an input/output error occurs
637 */
638 protected void unload(boolean doExpire, boolean isShutdown) throws IOException
```

Description This method checks if package protection mechanism is enabled through *SecurityUtil.isPackageProtectionEnabled()* method. If so *doUnload(boolean, boolean)* method is performed with privileges enabled through *AccessController .doPrivileged(PrivilegedExceptionAction⟨T⟩)* method. If the action's method throws an exception, it will propagate through the latter method. Otherwise *doUnload(boolean, boolean)* method is simply executed. If an exception occurs, it will be rethrown without handling.

doUnload(boolean doExpire, boolean isShutdown)

```
657 /**
658 * Saves any currently active sessions to file.
659 *
660 * @doExpire true if the unloaded sessions are to be expired, false
661 * otherwise
662 *
663 * @exception IOException if an input/output error occurs
664 */
665 private void doUnload(boolean doExpire, boolean isShutdown) throws IOException
```

Description This method checks *isShutDown* parameter. If it's true the session is saved. A new file (through *file()* method) and its output stream are opened. On this output stream are written all active sessions through *writeSession(OutputStream, boolean)* method. If *isShutDown* is false or file is not valid or an exception occurs sessions will not be written.

Chapter 2

Issues

setContainer(Container container)

```
245         /**
246         * Set the Container with which this Manager has been associated. If
247         * it is a Context (the usual case), listen for changes to the session
248         * timeout property.
249         *
250         * @param container The associated Container
251         */
252         @Override
253         public void setContainer(Container container) {
254 
255             // De-register from the old Container (if any)
256             if ((this.container != null) && (this.container instanceof Context))
257                 ((Context) this.container).removePropertyChangeListener(this);
258 
259             // Default processing provided by our superclass
260             super.setContainer(container);
261 
262             // Register with the new Container (if any)
263             if ((this.container != null) && (this.container instanceof Context)) {
264                 setMaxInactiveIntervalSeconds
265                     ( ((Context) this.container).getSessionTimeout()*60 );
266                 ((Context) this.container).addPropertyChangeListener(this);
267             }
268 
269         }
```

Problems

1. @ 256: missing curly braces surrounding the *if* statement [11]

2. @ 264: line break does not occur after a comma or an operator [15]

There are no major or critical problems.

load()

```
426     /**
427     * Loads any currently active sessions that were previously unloaded
428     * to the appropriate persistence mechanism, if any. If persistence is not
429     * supported, this method returns without doing anything.
430     *
431     * @exception ClassNotFoundException if a serialized class cannot be
432     * found during the reload
433     * @exception IOException if a read error occurs
434     */
435     public void load() throws ClassNotFoundException, IOException {
436         if (SecurityUtil.isPackageProtectionEnabled()){
437             try{
438                 AccessController.doPrivileged(new PrivilegedDoLoadFromFile());
439             } catch (PrivilegedActionException ex){
440                 Exception exception = ex.getException();
441                 if (exception instanceof ClassNotFoundException){
442                     throw (ClassNotFoundException)exception;
443                 } else if (exception instanceof IOException) {
444                     throw (IOException)exception;
445                 }
446                 if (log.isLoggable(Level.FINE)) {
447                     log.log(Level.FINE, "Unreported exception in load() "
448                             + exception);
449                 }
450             }
451         } else {
452             doLoadFromFile();
453         }
454     }
```

Problems

1. @ 447: line break does not occur after a comma or an operator [15]

There are no major or critical problems.

doLoadFromFile( )

```
457  /**
458  * Loads any currently active sessions that were previously unloaded
```

```
459  * to file
460  *
461  * @exception ClassNotFoundException if a serialized class cannot be
462  * found during the reload
463  * @exception IOException if a read error occurs
464  */
465 private void doLoadFromFile() throws ClassNotFoundException, IOException {
466     if (log.isLoggable(Level.FINE)) {
467         log.log(Level.FINE, "Start: Loading persisted sessions");
468     }
469
470     // Open an input stream to the specified pathname, if any
471     File file = file();
472     if (file == null || !file.exists() || file.length() == 0) {
473         return;
474     }
475     if (log.isLoggable(Level.FINE)) {
476         String msg = MessageFormat.format(rb.getString(LOADING_PERSISTED_SESSION), pathname);
477         log.log(Level.FINE, msg);
478     }
479     FileInputStream fis = null;
480     try {
481         fis = new FileInputStream(file.getAbsolutePath());
482         readSessions(fis);
483         if (log.isLoggable(Level.FINE)) {
484             log.log(Level.FINE, "Finish: Loading persisted sessions");
485         }
486     } catch (FileNotFoundException e) {
487         if (log.isLoggable(Level.FINE)) {
488             log.log(Level.FINE, "No persisted data file found");
489         }
490     } finally {
491     try {
492         if (fis != null) {
493             fis.close();
494         }
495     } catch (IOException f) {
496         // ignore
497     }
498     // Delete the persistent storage file
499     deleteFile(file);
500     }
501 }
```

Problems

1. @ 495 catch not managed [53]

Other Problems There is two main problem that must be fixed in order to have an easier debug:

1. the catch statement @ 495 must be managed
2. @ 473 is better to use the log to inform the caller that the method returns without perform any operation

readSessions(InputStream is)

```
509 /*
510 * Reads any sessions from the given input stream, and initializes the
511 * cache of active sessions with them.
512 *
513 * @param is the input stream from which to read the sessions
514 *
515 * @exception ClassNotFoundException if a serialized class cannot be
516 * found during the reload
517 * @exception IOException if a read error occurs
518 */
519 public void readSessions(InputStream is)
520        throws ClassNotFoundException, IOException {
521
522     // Initialize our internal data structures
523     sessions.clear();
524
525     ObjectInputStream ois = null;
526     try {
527         BufferedInputStream bis = new BufferedInputStream(is);
528         if (container != null) {
529             ois = ((StandardContext)container).createObjectInputStream(bis);
530         } else {
531             ois = new ObjectInputStream(bis);
532         }
533     } catch (IOException ioe) {
534         String msg = MessageFormat.format(rb.getString(LOADING_PERSISTED_SESSION_IO_EXCEPTION),
535                                           ioe);
536
537         log.log(Level.SEVERE, msg, ioe);
538         if (ois != null) {
539             try {
540                 ois.close();
541             } catch (IOException f) {
542                 // Ignore
543             }
544             ois = null;
545         }
546         throw ioe;
```

```
547     }
548
549     synchronized (sessions) {
550         try {
551             Integer count = (Integer) ois.readObject();
552             int n = count.intValue();
553             if (log.isLoggable(Level.FINE))
554                 log.log(Level.FINE, "Loading " + n + " persisted sessions");
555             for (int i = 0; i < n; i++) {
556                 StandardSession session =
557                     StandardSession.deserialize(ois, this);
558                 session.setManager(this);
559                 sessions.put(session.getIdInternal(), session);
560                 session.activate();
561             }
562         } catch (ClassNotFoundException e) {
563             String msg = MessageFormat.format(rb.getString(CLASS_NOT_FOUND_EXCEPTION),
564                                               e);
565             log.log(Level.SEVERE, msg, e);
566             if (ois != null) {
567                 try {
568                     ois.close();
569                 } catch (IOException f) {
570                     // Ignore
571                 }
572                 ois = null;
573             }
574             throw e;
575         } catch (IOException e) {
576             String msg = MessageFormat.format(rb.getString(LOADING_PERSISTED_SESSION_IO_EXCEPTION),
577                                               e);
578           log.log(Level.SEVERE, msg, e);
579             if (ois != null) {
580                 try {
581                     ois.close();
582                 } catch (IOException f) {
583                     // Ignore
584                 }
585                 ois = null;
586             }
587             throw e;
588         } finally {
589             // Close the input stream
590             try {
591                 if (ois != null) {
592                     ois.close();
593             }
594             } catch (IOException f) {
595                 // ignore
596             }
```

```
597         }
598     }
599 }
```

Problems

1. @ 578 two spaces instead of four are used for indentation [8]
2. @ 553-4 the if statement is not surronded by curly braces [11]
3. @ 566-573, @ 579-586, @590-596 duplicated code [27]
4. @ 541, @ 569, @ 582, @ 594 catch not managed [53]

Other Problems There are three main problems that must be fixed in order to avoid some bugs:

1. the duplicated code @ 566-573, @ 579-586 can be deleted because the finally statement does the same thing
2. the catch statement @ 541, @ 569, @ 582, @ 594 must be managed
3. there is a dead code @ 538-545; the variable ois, @ 538, is null so the rest of the code is never executed

unload(boolean doExpire, boolean isShutdown)

```
626 /**
627 * Save any currently active sessions in the appropriate persistence
628 * mechanism, if any. If persistence is not supported, this method
629 * returns without doing anything.
630 *
631 * @doExpire true if the unloaded sessions are to be expired, false
632 * otherwise
633 * @param isShutdown true if this manager is being stopped as part of a
634 * domain shutdown (as opposed to an undeployment), and false otherwise
635 *
636 * @exception IOException if an input/output error occurs
637 */
638 protected void unload(boolean doExpire, boolean isShutdown) throws IOException {
639     if (SecurityUtil.isPackageProtectionEnabled()){
640         try {
641             AccessController.doPrivileged(
642                 new PrivilegedDoUnload(doExpire, isShutdown));
643         } catch (PrivilegedActionException ex){
644             Exception exception = ex.getException();
645             if (exception instanceof IOException){
```

```
646                 throw (IOException)exception;
647             }
648             if (log.isLoggable(Level.FINE))
649                 log.log(Level.FINE, "Unreported exception in unLoad() " + exception);
650         }
651     } else {
652         doUnload(doExpire, isShutdown);
653     }
654 }
```

Problems

1. @ 631: the description of first parameter is badly written. Instead it has to be written like this: **@param doExpire** true if the unloaded sessions are to be expired, false otherwise [23]

2. @ 649: missing curly braces surrounding the *if* statement [11]

doUnload(boolean doExpire, boolean isShutdown)

```
657 /**
658 * Saves any currently active sessions to file.
659 *
660 * @doExpire true if the unloaded sessions are to be expired, false
661 * otherwise
662 *
663 * @exception IOException if an input/output error occurs
664 */
665 private void doUnload(boolean doExpire, boolean isShutdown) throws IOException {
666     if(isShutdown) {
667         if(log.isLoggable(Level.FINE)) {
668             log.log(Level.FINE, "Unloading persisted sessions");
669         }
670         // Open an output stream to the specified pathname, if any
671         File file = file();
672         if(file == null || !isDirectoryValidFor(file.getAbsolutePath())) {
673             return;
674         }
675         if(log.isLoggable(Level.FINE)) {
676             log.log(Level.FINE, SAVING_PERSISTED_SESSION, pathname);
677         }
678         FileOutputStream fos = null;
679         try {
680             fos = new FileOutputStream(file.getAbsolutePath());
681             writeSessions(fos, doExpire);
682             if(log.isLoggable(Level.FINE)) {
683                 log.log(Level.FINE, "Unloading complete");
```

```
684             }
685         } catch(IOException ioe) {
686             if(fos != null) {
687                 try {
688                     fos.close();
689                 } catch(IOException f) {
690                     ;
691                 }
692                 fos = null;
693             }
694             throw ioe;
695         } finally {
696             try {
697                 if(fos != null) {
698                     fos.close();
699                 }
700             } catch(IOException f) {
701                 // ignore
702             }
703         }
704     }
705 }
```

Problems

1. @ 660: the description of first parameter is badly written. Instead it should be written like this: **@param doExpire** true if the unloaded sessions are to be expired, false otherwise [23]
2. Javadoc is not complete. It misses of second parameter description [23]
3. @ 689, @ 700: exception not handled. The catch block is empty. The exception should be either logged or rethrown [53]

Other problems

1. @ 678: resources 'fos' should be managed by try-with-resource
2. @ 690: useless empty statement

Other problems

- The methods inherited from class and interfaces lack of @Override annotations
- The instance variables are in mixed visibility order [25]

Appendix A

Appendix

A.1 References

- Software Engineering 2 Project AA 2015/2016: Project Description And Rules
- Software Engineering 2 Project AA 2015/2016: Assignments 3 - Code Inspection
- CodeInspectionChecklist https://github.com/AntoniniP/CodeInspectionChecklist

A.2 Checklist

Naming Conventions

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.
2. If one-character variables are used, they are used only for temporary "throwaway" variables, such as those used in for loops.
3. Class names are nouns, in mixed case, with the first letter of each word in capitalized. Examples: `class Raster`; `class ImageSprite`;
4. Interface names should be capitalized like classes.
5. Method names should be verbs, with the first letter of each addition word capitalized. Examples: `getBackground()`; `computeTemperature()`.
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. Examples: `_windowHeight`, `timeSeriesData`.

7. Constants are declared using all uppercase with words separated by an underscore. Examples: `MIN_WIDTH`; `MAX_HEIGHT`.

Indention

8. Three or four spaces are used for indentation and done so consistently.
9. No tabs are used to indent.

Braces

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).
11. All `if`, `while`, `do-while`, `try-catch`, and `for` statements that have only one statement to execute are surrounded by curly braces. Example: avoid this:

```
if ( condition )
    doThis();
```

instead do this:

```
if ( condition )
{
    doThis();
}
```

File Organization

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).
13. Where practical, line length does not exceed 80 characters.
14. When line length must exceed 80 characters, it does NOT exceed 120 characters.

Wrapping Lines

15. Line break occurs after a comma or an operator.
16. Higher-level breaks are used.
17. A new statement is aligned with the beginning of the expression at the same level as the previous line.

Comments

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

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.

Java Source Files

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

21. The public class is the first class or interface in the file.

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

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

Package and Import Statements

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

Class and Interface Declarations

25. The class or interface declarations shall be in the following order:
    (a) class/interface documentation comment;
    (b) class or interface statement;
    (c) class/interface implementation comment, if necessary;
    (d) class (static) variables;
        i. first public class variables;
        ii. next protected class variables;
        iii. next package level (no access modifier);
        iv. last private class variables.
    (e) instance variables;
        i. first public instance variables;
        ii. next protected instance variables;
        iii. next package level (no access modifier);
        iv. last private instance variables.
    (f) constructors;
    (g) methods.

26. Methods are grouped by functionality rather than by scope or accessibility.

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

Initialization and Declarations

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

29. Check that variables are declared in the proper scope.

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

31. Check that all object references are initialized before use.

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

33. Declarations appear at the beginning of blocks (A block is any code surrounded by curly braces '{' and '}'). The exception is a variable can be declared in a `for` loop.

Method Calls

34. Check that parameters are presented in the correct order.

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

36. Check that method returned values are used properly.

Arrays

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).

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

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

Object Comparison

40. Check that all objects (including Strings) are compared with `equals` and not with `==`.

Output Format

41. Check that displayed output is free of spelling and grammatical errors.
42. Check that error messages are comprehensive and provide guidance as to how to correct the problem.
43. Check that the output is formatted correctly in terms of line stepping and spacing.

Computation, Comparisons and Assignments

44. Check that the implementation avoids "brutish programming": (see `http://users.csc.calpoly.edu/~jdalbey/SWE/CodeSmells/bonehead.html`).
45. Check order of computation/evaluation, operator precedence and parenthesizing.
46. Check the liberal use of parenthesis is used to avoid operator precedence problems.
47. Check that all denominators of a division are prevented from being zero.
48. Check that integer arithmetic, especially division, are used appropriately to avoid causing unexpected truncation/rounding.
49. Check that the comparison and Boolean operators are correct.
50. Check throw-catch expressions, and check that the error condition is actually legitimate.
51. Check that the code is free of any implicit type conversions.

Exceptions

52. Check that the relevant exceptions are caught.
53. Check that the appropriate action are taken for each catch block.

Flow of Control

54. In a `switch` statement, check that all cases are addressed by break or return.
55. Check that all switch statements have a default branch.
56. Check that all loops are correctly formed, with the appropriate initialization, increment and termination expressions.

Files

57. Check that all files are properly declared and opened.
58. Check that all files are closed properly, even in the case of an error.
59. Check that EOF conditions are detected and handled correctly.
60. Check that all file exceptions are caught and dealt with accordingly.

A.3 Software and tool used

- LaTeX http://www.latex-project.org/: to redact and to format this document
- eclipse https://eclipse.org/: to analyse the code

A.4 Working hours

This is the time spent for redact the document

- Belluschi Marco: 10 hours
- Cerri Stefano: 10 hours
- Di Febbo Francesco: 10 hours