TaxiService

Code Inspection

Authors: Jacopo Silvestri 773082 Simone Penati 850448
Reference Professor: Mirandola Raffaela
Release Date February 2016

POLITECNICO
DI MILANO

0. Table of Content

1. Classes Assigned

1.1 replaceSpecial (Object val)

appserver/admingui/common/src/main/java/org/glassfish/admingui/common/util/JSONUtil.java
Starting line 710

1.2 reloadApplication (String appName, List<String> targets, HandlerContext handlerCtx)

appserver/admingui/common/src/main/java/org/glassfish/admingui/common/util/DeployUtil.java
Starting line 67

1.3 getApplicationTarget (String appName, String ref)

appserver/admingui/common/src/main/java/org/glassfish/admingui/common/util/DeployUtil.java
Starting line 98

1.4 getRefEndpoints (String name, String ref)

appserver/admingui/common/src/main/java/org/glassfish/admingui/common/util/DeployUtil.java
Starting line 128

1.5 getTargetEnableInfo (String appName, boolean useImage, boolean isApp)

appserver/admingui/common/src/main/java/org/glassfish/admingui/common/util/DeployUtil.java
Starting line 164

2. Functional Roles

DeployUtil is a class that have to guarantee the correct functioning of the generation and reload of the Applications and Endpoints. While replaceSpecial belongs to an entirely different class and it is never called in any of the DeployUtil's methods, all the others manage directly various aspect in which the applications are runned by Glassfish on different servers and clusters, and the way they reload. The getTargetEnableInfo method counts on how many among clusters and servers, if any in the application's target list, the given application is enabled; it uses the getApplicationTarget method to generate the targets list. The getApplicationTarget method generates a list of targets to which it adds every server or cluster that run the given application. The reloadApplication method asks for a list of target; then it will force the shutdown of the application by changing its status to 'disabled' and then enabling it again. The getRefEndpoints method returns an hash map where every cluster or server in the target list is paired with the encoded URL of the application contextualized to the cluster or server itself.
All these methods focus on managing a particular aspect of application deployment: this was the main "clue" that brought us to understand the functional role of the class.

3. Issues Found

3.1 JSONUtil

3.1.0 General Issues

- Sometimes indentation is not correct (in methods other than the ones assigned to us).
- Javadoc on Glassfish.pompel.me only concerns javaToJSON and jsonToJava methods. Therefore we cannot check eventual external program interface consistency with the assigned class. However replaceSpecial, does what the javadoc inside its code states it should.
- No Constructor method.
- jsonToJava (JsonChars json) is not grouped with the other methods that have similar functionalities.

3.1.1 replaceSpecial (Object val)

- Incomplete Javadoc!
- Indentation made by mixed tabs and spaces!
- Name not completely clear ("replaceSpecialStrings" could be a logical solution to this problem).

3.2 DeployUtil

3.2.0 General Issues

- No Class Javadoc! And therefore we cannot check that the class interface is coherent to what the javadoc states.
- File organization is thoroughly consistent and correct, but there are some imprecisions such as blank lines inside methods and scattered comments throughout the code.
- The 120 character limits is exceeded at line 104, 113, 135, 138, 146, 149, 177, 178, 179, 182, 185, 199, 201.
- Comments are not adequate. They're scattered. without apparent logic or a coherent style.
- No Constructor method.
- Order of methods' visibility and options is not consistent (es: static public, public static).
- Bad exception handling!

3.2.1 reloadApplication (String appName, List<String> targets, HandlerContext handlerCtx)

- Incomplete Javadoc!

3.2.2 getApplicationTarget (String appName, String ref)

- Incomplete Javadoc!
- Try/Catch block with generic exception on exceptions handling!

3.2.3 getRefEndpoints (String name, String ref)

- No Javadoc!
- Not enough comments!
- The "aMap" variable has not a clear name!
- Try/Catch block with generic exception on exceptions handling!

3.2.4 getTargetEnableInfo (String appName, boolean useImage, boolean isApp)

- No Javadoc!
- Try/catch block with ignores on exceptions handling!
- Not enough comments!
- Method's name not clear enough!
- Commented Code!

- Variable declaration should be at the beginning of the method or in the proper scope.
- Variable enable should be boolean.

4. Point-by-Point Check

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.
The method getTargetEnableInfo could have a different name, specifying more clearly its function.
The method replaceSpecial has a vague name.
The variable HashMap aMap has a generic name.
2. If one-character variables are used, they are used only for temporary "throwaway" variables, such as those used in for loops.
None one-character variables are used.
3. Class names are nouns, in mixed case, with the first letter of each word in capitalized.
Considering that JSON is an acronym, and is therefore understandable to have JSONUtil named as it is, there are no problems with this statement.
4. Interface names should be capitalized like classes.
No interfaces declared.
5. Method names should be verbs, with the first letter of each addition word capitalized.
No problems with this statement.
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 problems with this statement.
7. Constants are declared using all uppercase with words separated by an underscore.
No problems with this statement.

Indention

8. Three or four spaces are used for indentation and done so consistently.
Indentation Style not consistent in JSONUtil: mixed use of tabs and spaces in the replaceSpecial method.
9. No tabs are used to indent
JSONUtil: the replaceSpecial method uses Tabs.

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).
Both assigned classes use consistently the "Kernighan and Ritchie" style. No problems with this statement.
11. All if, while, do-while, try-catch, and for statements that have only one statement to execute are surrounded by curly braces.
If statements mainly contain only one statement in the assigned code, always surrounded by curly braces.
No problems with this statement.

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).
Blank lines inside DeployUtil methods and scattered comments throughout the code. That must be changed. Little to none beginning comments and comments before method declaration explaining their usage, making code difficult to read.
13. Where practical, line length does not exceed 80 characters.
The majority of lines in DeployUtil which exceeds this limit could easily avoid that, since it is mainly string concatenation causing this. The code could easily use a line break after the concatenation “+” before the subsequent string fragment.
14. When line length must exceed 80 characters, it does NOT exceed 120 characters.
The 120 character limits is exceeded at line 104, 113, 135, 138, 146, 149, 177, 178, 179, 182, 185, 199, 201.

Wrapping Lines

15. Line break occurs after a comma or an operator.
Line breaks always occurs after a comma or an operator; DeployUtil could use more line breaks, however.
16. Higher-level breaks are used.
No breaks used.
17. A new statement is aligned with the beginning of the expression at the same level as the previous line.
No problems with this statement.

Comments

18. Comments are used to adequately explain what the class, interface, methods, and blocks of code are doing.
Comments do not adequately in explain methods usage and function. They also lack coherence and uniformity of style in all methods of Deploy Util.
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.
There is commented code without nor reason nor date, lines 211-218 of DeployUtil, inside the getTargetEnableInfo method.

Java Source Files

20. Each Java source file contains a single public class or interface.
No problems with this statement: JSONUtil contains another static class but it is not a public class.
21. The public class is the first class or interface in the file.
No problems with this statement.
22. Check that the external program interfaces are implemented consistently with what is described in the javadoc.
With regard to JSONUtil, the javadoc on glassfish.pompel.me only concerns jsonToJava and javaToJSON methods, therefore we cannot check eventual external program interface consistency with the assigned class. However, the method replaceSpecial does what the javadoc inside its code states it should.
In DeployUtil, on the other hand, the javadoc is completely missing, except an author declaration.
23. Check that the javadoc is complete (i.e., it covers all classes and files part of the set of classes assigned to you).
Javadoc is not complete in the following methods: replaceSpecial, ReloadApplication and getTargetApplication.
Javadoc is completely absent in the getTargetEnableInfo and getRefEndpoints methods.
Class javadoc is thorough in JSONUtil but completely absent in DeployUtil.

Package and Import Statements

24. If any package statements are needed, they should be the first non-comment statements. Import statements follow.
No problems with this statement.

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*
  - *a. first public class variables*
  - *b. next protected class variables*
  - *c. next package level (no access modifier)*
  - *d. last private class variables*
- *E. instance variables*
  - *a. first public instance variables*
  - *e. next protected instance variables*
  - *f. next package level (no access modifier)*
  - *g. last private instance variables*
- *F. constructors*
- *G. methods*

No problems with this statement.

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

jsonToJava (JsonChars json) is not grouped with the other methods that have similar functionalities.

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

No problems with this statement.

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).

The variable enable in getTargetEnableInfo should be a boolean variable.

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

In targetEnableInfo not all variable declaration is at the beginning of the method or in the proper scope: line 210, numTargets.

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

No problems with this statement.

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

No problems with this statement.

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

No problems with this statement.

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.

No problems with this statement.

Method Calls

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

No problems with this statement.

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

No problems with this statement.

36. Check that method returned values are used properly

No problems with this statement.

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).
No problems with this statement: arrays are parsed as collections, without using incremental indexes.
38. Check that all array (or other collection) indexes have been prevented from going out-of-bounds.
No problems with this statement.
39. Check that constructors are called when a new array item is desired.
No problems with this statement.

Object Comparison

40. Check that all objects (including Strings) are compared with "equals" and not with "==".
No object comparisons. No problems with this statement.

Output Format

41. Check that displayed output is free of spelling and grammatical errors.
No Output Text.
42. Check that error messages are comprehensive and provide guidance as to how to correct the problem.
During exception handling there are no error messages, aside from the logger information. No guidance whatsoever is provided.
43. Check that the output is formatted correctly in terms of line stepping and spacing.
No Output Text.

Computation, Comparisons and Assignments

44. Check that the implementation avoids “brutish programming: (see http://users.csc.calpoly.edu/~jdalbey/SWE/CodeSmells/bonehead.html).
The variable enable in getTargetEnableInfo should be a boolean variable.
45. Check order of computation/evaluation, operator precedence and parenthesizing
No problems with this statement.
46. Check the liberal use of parenthesis is used to avoid operator precedence problems.
No problems with this statement.
47. Check that all denominators of a division are prevented from being zero.
No problems with this statement.
48. Check that integer arithmetic, especially division, are used appropriately to avoid causing unexpected truncation/rounding.
No problems with this statement.
49. Check that the comparison and Boolean operators are correct.
No problems with this statement.
50. Check throw-catch expressions, and check that the error condition is actually legitimate.
No problems with this statement.
51. Check that the code is free of any implicit type conversions.
No problems with this statement.

Exceptions
52. Check that the relevant exceptions are caught
Generic exception in every try catch block used.
53. Check that the appropriate action are taken for each catch block
Bad exception handling in all methods of DeployUtil.

Flow of Control
54. In a switch statement, check that all cases are addressed by break or return
No switch statements in the assigned code.
55. Check that all switch statements have a default branch
No switch statements in the assigned code.
56. Check that all loops are correctly formed, with the appropriate initialization, increment and termination expressions
No switch statements in the assigned code.

Files
57. Check that all files are properly declared and opened
No file are used.
58. Check that all files are closed properly, even in the case of an error
No file are used.
59. Check that EOF conditions are detected and handled correctly
No file are used.
60. Check that all file exceptions are caught and dealt with accordingly
No file are used.

4.1 Other Issues

We did not find other relevant code issues in the code portion assigned to our team that were not already included in the checklist.