POLITECNICO DI MILANO

Software engineering 2 project

TAXI SERVICE

CODE INSPECTION DOCUMENT

AUTHOR

Pallotta Simone

PROFESSOR

Raffaela Mirandola

Version 1.0

ACADEMIC YEAR 2015/2016

intentionally left blank

CONTENTS

1. Analysed classes and methods

P**ublic Class:** StandardContext

E**xtends:** ContainerBase

I**mplements:** Context, javax.servlet.ServletContext

M**ethods:**

Name:addFilter(String filterName , Filter filter)
Start Line:3009
Location: appserver/web/web-core/src/main/java/org/apache/catalina/core/StandardContext.java

Name:addFilter(String filterName , Class < ? extends Filter > filterClass)
Start Line:3096
Location: appserver/web/web-core/src/main/java/org/apache/catalina/core/StandardContext.java

2. Roles of our class clusters

StandardContext class is a standard implementation of the c**ontext** interface.

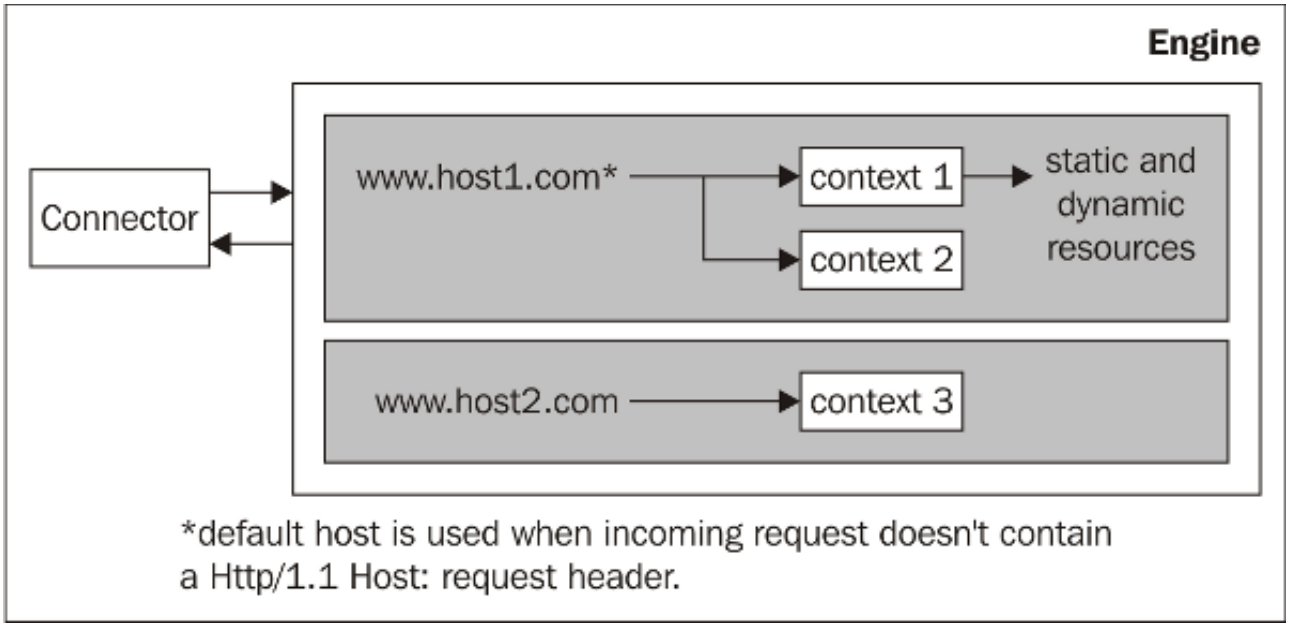
The application server (Tomcat, Glassfish, …) uses contexts to distribute requests to the appropriate servlets.

The context Container represents a single web application running within a given instance of an application server.

Once a Context has been defined, Catalina will attempt to match incoming HTTP requests to its context path.

There is no limit to the number of contexts that can be defined, as long as each context is given its own unique context path

In summary, when an HTTP request is made, Catalina receives it, and passes it to the appropriate Context, which in turn passes the request to the appropriate servlet to serve it.



Hint and screen taken from: https://www.packtpub.com/books/content/overview-tomcat-6-servlet-container-part-1

The method **addFilter(String filterName, Filter filter)** registers the given filter instance with this ServletContext under the given filterName.

The other one, **addFilter(String filterName,Class <? extends Filter> filterClass)** adds the filter with the given name and class type to this servlet.

Both are methods of the interface **javax.servlet.ServletContext.**

The interface defines a set of methods that a servlet uses to communicate with its servlet container.

There is one context container per "web application" (A "web application" is a collection of servlets and content installed under a specific subset of the server's URL namespace such as /**catalog** and possibly installed via a **.war** file.)

A filter is an object that performs filtering tasks on either the request to a resource (a servlet or static content), or on the response from a resource, or both.

A filter is typically used to perform a particular piece of functionality: if a request is made for a particular resource such as a servlet and a filter is used, the filter code may execute and then pass the user on to the servlet.

As a further example, the filter might determine that the user does not have permissions to access a particular servlet, and it might send the user to an error page rather than to the requested resource.

A filter dynamically intercepts requests and responses to transform or use the information contained in the requests or responses.

Filters typically do not themselves create responses, but instead provide universal functions that can be "attached" to any type of servlet or JSP page.

A filters can be used to transform the response from a servlet or a JSP page.

A common task for the web application is to format data sent back to the client.

Increasingly the clients require formats (for example, WML) other than just HTML.

To accommodate these clients, there is usually a strong component of transformation or filtering in featured web application.

3. Discovered Issues

In this section, we report on the quality status of the two methods assigned using the checklist for JAVA code inspection.

3.1 Naming Convention

1. attributes, classes, interface satisfy the JAVA convention
2. no *one-character* variables used
3. ***DynamicFilterRegistrationImpl***, ***FilterDef*** are class and follow the JAVA naming convention
4. This class implements **Context**, **ServletContext** and are capitalized
5. both methods addFilter(….) respect the JAVA convention
6. ***isContextInitializedCalled, filterDefs*** are class attributes and conform with JAVA convention, no attributes begin with underscore (“_”)
7. ***SERVLET_CONTEXT_ALREADY_INIT_EXCEPTION, NULL_EMPTY_FILTER_NAME_EXCEPTION*** are constant and satisfy the JAVA convention.

3.2 Indentation

All line of codes seems to indent with tabs and no space, this do not satisfy the JAVA convention.

3.3 Braces

The piece of code use “Kernighan and Ritchie” style, we show some example taken from assigned methods:

```
if (isContextInitializedCalled) {
        ……
    }
if (filterName == null || filterName.length() == 0) {
        ……
    }
……..
```

All the control statement even with one “execute” statement used braces conform with JAVA convention.

3.4 File Organization

no line exceed 80 character.

no line exceed 120 character.

3.5 Wrapping Lines

wrapping lines rule are satisfied.

3.6 Comments

Comments are used to explain what the methods and blocks of code are doing.

no commented out code present

3.7 Java Source Files

The **StarndardContext.java** contains only one class.

```
124
125 public class StandardContext
126     extends ContainerBase
127     implements Context, ServletContext
128 {
```

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

3.8 Package and Import Statements

```
59 package org.apache.catalina.core;
60
61 import org.glassfish.grizzly.http.server.util.AlternateDocBase;
62 import org.glassfish.grizzly.http.server.util.Mapper;
63 import org.glassfish.grizzly.http.server.util.MappingData;
64 import org.glassfish.logging.annotation.LogMessageInfo;
65 import org.glassfish.pfl.basic.logex.Log;
66 import org.apache.catalina.*;
```

Satisfy JAVA convention: first the packages and then the imports.

3.9 Class and Interface Declarations

• The class or interface declarations shall be in the following order:

points A,B,C follow the JAVA convention, point D “class (static) variables”

```
private static final ClassLoader standardContextClassLoader =
    StandardContext.class.getClassLoader();

private static final Set<SessionTrackingMode> DEFAULT_SESSION_TRACKING_MODES =
    EnumSet.of(SessionTrackingMode.COOKIE);

/**
 * Array containing the safe characters set.
 */
protected static final URLEncoder urlEncoder;

/**
 * The descriptive information string for this implementation.
 */
private static final String info =
    "org.apache.catalina.core.StandardContext/1.0";

private static final RuntimePermission GET_CLASSLOADER_PERMISSION =
    new RuntimePermission("getClassLoader");
```

does not follow, as we can see, the rule

```
// ---------------------------------------------------------- Constructors

/**
 * Create a new StandardContext component with the default basic Valve.
 */
public StandardContext() {
    pipeline.setBasic(new StandardContextValve());
    namingResources.setContainer(this);
    if (Globals.IS_SECURITY_ENABLED) {
        mySecurityManager = AccessController.doPrivileged(
                new PrivilegedCreateSecurityManager());
    }

    //START PWC 6403328
    this.logPrefix = logName() + " ServletContext.log():";
    //END PWC 6403328
}

// ----------------------------------------------------- Instance Variables

/**
```

Constructors come before than instance variables, does not follow the convention.

Instance variables does not respect the order imposed by convention.

3.10 Object Comparison

```
synchronized (filterDefs) {
    for (Map.Entry<String, FilterDef> e : filterDefs.entrySet()) {
        if (filterName.equals(e.getKey()) ||
            filter == e.getValue().getFilter()) {
            return null;
        }
    }
```

Object filter do not use equals().

3.11 Flow of Control

no standard for loop used, is used foreach loop instead.

The loops are formalised correctly.

4. Other problems

Analyzing the class hierarchy we found a naming issue in its ancestor class “ContainerBase” seems to break a rule number 1 as the name suggest is the bar of the container; the correct name should be “BaseContainer”.

Appendix

Code inspection checklist

Naming Conventions

- All class names, interface names, method names, class variables, method variables, and constants used should have meaningful names and do what the name suggests.
- If one-character variables are used, they are used only for temporary "throwaway" variables, such as those used in for loops.
- Class names are nouns, in mixed case, with the first letter of each word in capitalized. Examples: class Raster; class ImageSprite;
- Interface names should be capitalized like classes.
- Method names should be verbs, with the first letter of each addition word capitalized. Examples: getBackground(); computeTemperature().
- 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.
- Constants are declared using all uppercase with words separated by an underscore. Examples: MIN_WIDTH; MAX_HEIGHT;

Indention

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

Braces

- 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).
- 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

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

Wrapping Lines

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

Comments

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

- Each Java source file contains a single public class or interface.
- The public class is the first class or interface in the file.
- Check that the external program interfaces are implemented consistently with what is described in the javadoc.
- 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

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

Class and Interface Declarations

- 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

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

- Check that variables and class members are of the correct type. Check that they have the right visibility (public/private/protected)
- Check that variables are declared in the proper scope
- Check that constructors are called when a new object is desired
- Check that all object references are initialized before use
- Variables are initialized where they are declared, unless dependent upon a computation
- 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

- Check that parameters are presented in the correct order
- Check that the correct method is being called, or should it be a different method with a similar name
- Check that method returned values are used properly

Arrays

- Check that there are no off-by-one errors in array indexing (that is, all required array elements are correctly accessed through the index)
- Check that all array (or other collection) indexes have been prevented from going out-of-bounds
- Check that constructors are called when a new array item is desired

Object Comparison

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

Output Format

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

Computation, Comparisons and Assignments

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

Exceptions

- Check that the relevant exceptions are caught
- Check that the appropriate action are taken for each catch block

Flow of Control

- In a switch statement, check that all cases are addressed by break or return
- Check that all switch statements have a default branch
- Check that all loops are correctly formed, with the appropriate initialisation, increment and termination expressions

Files

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