WIT: A Framework for In-Container Testing of Web-Portal Applications

Wenliang Xiong, Harpreet Bajwa & Frank Maurer
University of Calgary, Department of Computer Science
Calgary, Alberta, Canada T2N 1N4
+1 (403) 220-7140

{xiongw,bajwa,maurer}@cpsc.ucalgary.ca

Abstract: In this paper we describe a novel approach that allows for in-container testing of web portal applications. Concretely, our approach helps in locating and debugging (a) Deployment environment related problems, (b) Security: role based testing of resource access and (c) Problems arising from the interaction between the container and the application code in the form of request and response objects and other application environment objects. Our approach allows developers to write automated in-container test cases for web portal applications. Using Aspect technology, the test code is injected into the application code allowing the tests to run in the same environment as the portal application. WIT, our testing framework, provides the developers the ability to control the portal server environment by setting up an initial environment state before the execution of the application code. After the application code is executed, the environment state can be validated and cleaned up to prevent any traces or side effects. A test failure is reported if the results of executing the original code are incorrect. In this paper, we present the overall testing approach, design & implementation of WIT as well as a usage scenario.

1 Introduction

Container-based web technologies ease the burden on developers by providing underlying services such as persistence, security etc so that developers can concentrate on implementing the business logic. By providing robust and fine-tuned services to the application code the reliability, maintainability and performance of websites is improved considerably. The container further provides added value by managing the life cycle of the application code. While the advantages of container-based technologies are obvious, a container acts as a black box from the application developer's point of view and is only accessible via the API. Thus, automated testing of container-based application is challenging.

When an error occurs on the client side, it is difficult to predict the precise origin of the error. One of the reasons of the error may come from the container interacting incorrectly with the application code. Also, unpredictable changes in the container environment are often caused when the application code is deployed in the container. Although the application code runs correctly in the testing development container

environment, developers cannot be guaranteed success of the application in the production environment.

Testing an application for such errors that surface only at deployment time requires an approach for executing the test code inside the container environment and the ability to access and control the environment specific objects. We refer to this approach henceforth as in-container testing (ICT). Existing tools for front-end GUI or back-end business logic testing cannot test the deployment-related problems such as those mentioned above because the tests run outside of the container. A high-level report of the problems provided by them cannot be used to narrow down the scope of problems.

The motivation for our work comes from one of our industry partners that are building enterprise java based web portal applications [1]. The company reported 1) unknown deployment related errors and 2) lack of an automated way to test access to sensitive portal resources[1]. This paper addresses the problems discussed above by proposing a novel approach for performing automated ICT using the WIT framework for JSR [2] compliant portals.

The rest of the paper is organized as follows. Section 2 explains in detail the deployment related problems needed to be addressed by ICT. A detailed explanation of the architecture of web portal applications is provided in Section3. Section 4 provides a description of the design of WIT. Then some example usage scenarios and details on how tests can be implemented and run using WIT is provided in Section 5. Section 6 compares related work and approaches that currently exist for in-container testing of web-applications. Finally, section 7 discusses the future work and concludes our paper.

2 Problems Needed to Be Addressed by ICT

Unpredictable changes in the container environment are often caused when the application code is deployed in the production environment. Testing an application for such errors that surface only at deployment time requires an approach for executing the test code inside the container environment. In the following section we briefly discuss some of the problems encountered when a web-application is deployed in the production environment.

a) Deployment related problems: Certain environment attributes are set within the container at deployment time for e.g. descriptor files are read at deployment time and the environment is configured accordingly. That might, for example, mean that certain database resources are different in the test and the deployment environment or that some security roles do not match. Another possible difference between the test and the production container environment may be due to the fact that a different version of a library file is being referenced by the application code. All these subtle differences may introduce an error. For example, a portlet configured with the connection string to database A in the testing environment, for some reason, is assigned a connection string to database B when deployed to the production environment. Portlet code executing

[1] A more comprehensive analysis of portal test practices and the results of our case study are published concurrently in ICWE 2005 [16].

successfully in the test environment may fail because of the changed connection string. Another example: The version of a specific jar library is different in the test and production environment, e.g. a newer version is deployed in the test environment and referred by the portlet code directly or indirectly. The above examples highlight that the successful execution of the portlet code in test environment cannot guarantee its success in the production environment.
b) Security: role based testing of resource access. Access to sensitive resources for e.g. portlets [3] is controlled by assigning permissions to individual users or user groups granting the appropriate access. Without automated testing tool support the administrator setting the permissions must log in as a user with a specific role and test manually each time the applications are deployed in the production environment to verify whether the permissions have been correctly assigned.
c) Problems arising from the interaction between the container and the application code in the form of request, response objects and other application environment objects. In container-based web application, data submitted by the browser is assembled by the container as a request object. The data is then forwarded to the application code through access to certain environment objects. After the execution of the application code, results are sent back to the browser as a response object assembled by the container. The request and response objects are primarily responsible for carrying the data exchanged between the container and the application code. The application code can use all accessible objects as part of its business logic. Changing the values of some environment objects might create side effects on other parts of the application. Automatically testing the application code that relies on these objects requires a mechanism that allows developers to manipulate all these objects.

3 Portlet-based Web-Portal Application Architecture

Web Portals are an example of container-based web application providing a single integrated point of access to information by aggregating multiple streams of dynamic content rendered as portlet windows. Technically, a portlet is a piece of code that runs within the portlet container [3] and provides content fragments to be embedded into the portal pages.

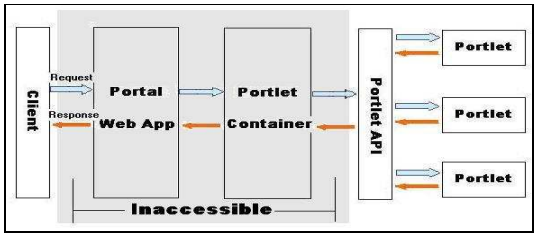

Fig. 1. Portal Server Component-Interactions

A client request as shown in Fig. 1 for a portal page interacts with multiple interfaces defined by the portal server components. The portal server completes the client request for the portal page by retrieving the portlets written by the developer for the current page. Thereafter the portal server invokes the portlet container for each portlet. The final portal page presented to the client represents the aggregated content generated by several portlets. With commercial portal servers, the source code of the portal server components as highlighted in Fig. 1 is inaccessible to the developer.

Because of the complexity of web portals, automated in-container testing presents four unique challenges. Firstly, the portlet API layer depicted in Fig. 1 is the only way that portlets can 'talk' to the inaccessible components. Thus, we need to find a way to intercept calls from the container to the portlets and vice versa so that testers can access and manipulate the calls generated by the container. Secondly, testing portlets involves invoking a series of inaccessible interactions in the portal server as seen in the Fig. 1. Thirdly, since the tests run in the container we need to collect individual test results of executing each portlet and then send back the aggregated results to the test client. Lastly, while the test code runs with the original application code, portal clients still should receive the correct response from the portlets. Thus, minimizing the side effects of the test code on the original portlet code becomes imperative. In the next section, we describe how our approach addresses these issues.

4 WIT: Web Portal Application In-container Testing Framework

4.1 Design Overview

The WIT system consists of following modules: Converter, Weaver, Invoker, Controller, and Repository.

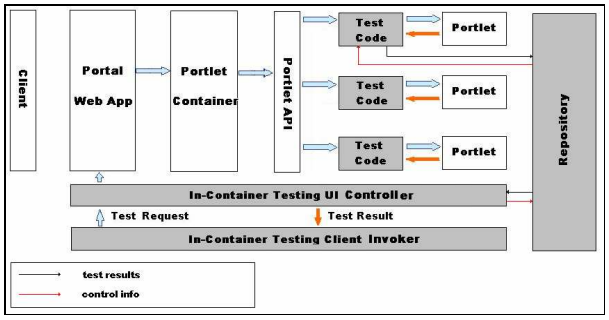

Fig. 2. In-Container Portlet Test Request Invocation

The tests are initiated by the testing client Invoker depicted in Fig.2. This starts a process whereby the test Controller assembles and sends the request for the portlet under test and simultaneously, writes the test control instruct to the Repository. Before the request reaches the portlet under test, a check is made to ensure that the test control instruct allows the test code to execute. If the check is successful, the test code 1) intercepts the calls between the application code and the portlet API and 2) sets up the initial state to execute the portlet code. After the portlet code is executed, the results of the tests get stored in the Repository and then reported to developers by the Controller.

4.2 Invoker & Controller

The Invoker is the starting point of the in-container testing process. The responsibilities of the Invoker are twofold. First, it calls the Converter & Weaver as explained in section 4.3 to generate portlet code together with the test code and then deploys the generated code into the target portal server. Secondly, it sends a test request to the Controller and reports the test results returned by the Controller. The Controller is a servlet that accepts the test request from the Invoker. It then simulates the invocation of the portlet from a browser and assembles the portlet request. Meanwhile, it writes a control instruct into the Repository to indicate which portlet is going to be tested. The Controller is also responsible for querying the test results saved in the Repository and then sending them back to the Invoker.

4.3 Converter & Weaver

We utilized AspectJ technology [4, 5] in order to intercept calls to the portlet code by injecting the test code into the portlet code.

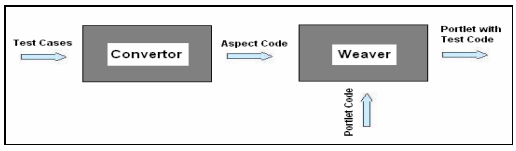

Fig. 3. Injecting the test code into Portlets

As shown in Fig. 3 test cases written for in-container testing are fed into the Converter first to generate Aspect code. This code is in turn compiled with the original portlet code by the AspectJ Weaver. As a result of this, the portlet binary class files are weaved in with the test code. During this phase, information like the location of the Repository is compiled into the portlet code as well. The final output of this converting & weaving phase is deployable portlet code together with the testing code.

4.4 Repository

Multiple Portlets run simultaneously within the portlet container and so does the test code. Writing the test results into a central location makes it possible to collect all the test results asynchronously. In order to provide better performance by avoiding I/O disk operations, we have chosen an in-memory database as our repository. Besides, the test results Repository also contains control information indicating which portlets are going to be tested. Only test code in Portlets indicated by the control information is executed. In this way, we avoid the side-effect from our tests on other portlets. Furthermore, if we clear such control information in Repository, no test code will be executed and thus, portlets are restored to the normal state to accept requests from users.

5 Usage Scenario of WIT

After providing an overview on the WIT architecture, we will now describe how a tester can write the in-container test cases for WIT. Further, three main usage scenarios of WIT will be discussed in detail with reference to an Accounts portlet example. Fig. 4 shows an example of an Accounts portlet class containing a method called doView[2] [2] which is invoked by the container. The portlet accesses the database connection string to connect to the backend database. The corresponding account id is retrieved from the PortletSession object [2], which is sent to the back-end database system to get the detail account information, which is in turn returned to client.

```
1)  public class AccountsPortlet{
2)  public void doView(PortletRequest request, PortletResponse re-
    sponse){
3)  try {
4)  PortletSettings portletSettings = request.getPortletSettings();
5)  String dbConnStr = portletSettings.getAttribute("AccountDB");
6)  //Now the AccountsPortlet can persist information to the back-
    end Account database
     ……
7)  String acctId =
8)  (String)request.getPortletSession().getAttribute("acctId");
9)  AccountDetail ad = AccountDB.getAccountDetail(acctId);
10) Request.setAttribute("AcctDtl", ad);
11) PrintWriter out = response.getWriter();
12) //following pseudo code prints out the AccountDetail object
13) response.setContentType("text/html");
14) out.println(……);
15) }
```

Fig. 4. doView Method – AccountsPortlet Class

[2] doView is the core method in which a portlet developer implements the business logic

5.1 In-container Test Case Naming Conventions

Our in-container test case classes follow a specific naming convention.
Name of test case class: = Name of portlet class + "Test"
The name of each test case starts exactly with the name of the portlet being tested and ends with the string "*Test*". For each portlet method being tested, there is a pair of test methods in the test case. The access modifier of these methods must be public, and the return type must be void. The name of these methods consists of three parts. The first part is either *"before"* or *"after"*, and the second part is the name of methods being tested, and the third part is any valid string to make the test methods more meaningful.
Name of test methods := (before | after) +
"_" + name of methods under testing +
"_" + additional string

5.2 Testing Deployment Related Problems

The test scenario presented in this section allows testing for deployment related problems (see Section 2 -- (a)). The PortletSettings [2] object contains configuration parameters accessed by the portlet code at runtime. These parameters are initially defined in the portlet descriptor file called portlet.xml. The portal administrator uses the administrative interface to configure individual portlet by editing the configuration parameters before deploying the application into the production environment.
For instance the accounts portlet as shown in Fig. 4 (line 4, 5, 6) accesses the database connection string by reading the configuration parameter from the portlet descriptor file (refer Fig. 5).

```
<concrete-portlet href="#Accounts">
  <portlet-name>Accounts</portlet-name>
   ……
  <config-param>
    <param-name>AccountDB</param-name>
    <param-value>jdbc:db2://localhost:50000/AccountDB</param-value>
  </config-param>
</concrete-portlet>
```

Fig. 5. A snippet of portlet.xml showing configuration parameters

The AccountsPortletTest code in Fig. 6 checks for the valid database connection string in the production environment. An incorrect value read by the portlet at runtime on the production environment will cause the AccountsPortletTest to fail.

```
public class AccountsPortletTest extends TestCase {
  private final String AccountDBConnStr =
                              "jdbc:db2://DB2BOX:50000/AccountDB";

  public void after_doView_ testGetAcctDBConnStr
  (PortletRequest request, PortletResponse response) {
       PortletSettings portletSettings = request.getPortletSettings();
       String dbConnStr =  portletSettings.getAttribute("AccountDB");
       assertEquals("AccountDB Connection String is incorrect",
                dbConnStr, AccountDBConnStr);
  }
}
```

Fig. 6. doView() – AccountsPortlet Test Case For Database Connection String

5.3 Automated Testing Security: Role Based Testing of Resource Access.

The test scenario presented in this section tests security privileges. We first highlight how the In-container security test case classes differ in naming convention from other test classes. A specific naming convention described below is used.

Name of security test case class: = Name of portlet class + "SecurityTest"

Name of security test case: ="test"+ (View|Edit|Config) + "Security"

Next, we discuss an example scenario below whereby a portal user called David is trying to access a sensitive resource which ideally he should not have access to.

WIT, will first weave the security test case testViewSecurity () code in Fig. 7 into the doView() of the account portlet class, and then login to the portal application with the specified user name and password, and then send a request to view AccountsPortlet. If the request is successful for some reason the doView() method in AccountsPortlet will be executed – which should have been prevented by the security system. Thus, the execution of the doView method means that the security test has failed and, thus, the testViewSecurity method triggers a "fail". This in turn reports a test failure to the developer.

```
public class AccountsPortletSecurityTest
              extends SecurityTestCase {
   public String getAuthenUrl() { return "http://ict5/login"; }
   public String getAuthenUser() {return "david"; }
   public String getAuthenPwd() { return "pass"; }
   public String getPortletInvokeUrl() {
        return "http://ict5/Acct";
   }
   public String testViewSecurity() {
        fail("the user:"+getAuthenUser()+" should not be
                        able to view the AccountPortlet");
```

Fig. 7. doView() – AccountsPortlet Test Case For Security

5.4 Testing Problems Arising from the Interaction Between the Container and the Application Code

The test scenario presented in this section tests problems arising from the interaction between the container and the application code in the form of request, response objects and other application environment objects.
The AccountsPortlet depicted in Fig. 4 displays account detail information according to the account id number submitted by the user (line 7-14). In Fig. 8, developers set up the initial testing environment in the **before_doView**`test_GetAcctDetail` method by adding an account id into session object, and check the environment in the **after_doView**`test_GetAcctDetail` method by comparing the outBalance with the expected number 10. If the account outbalance is not what we expected, then a failure is fired.

```
public class AccountsPortletTest extends TestCase {
  public void before_doView_testGetAcctDetail
    (PortletRequest request, PortletResponse response) {
       session.setAttribute("acctId", "123");
  }

  public void after_doView_testGetAcctDetail
     (PortletRequest request, PortletResponse response) {
     AccountDetail ad =
                   (AccountDetail) request.getAttribute("AcctDtl");
   double outBalance = ad.getOutstandingBalance();
   assertEquals("outstanding balance is incorrect", outBalance, 10);
 }
}
```

Fig. 8. doView () Test Case – AccountsPortletTest Class

5.5 Using WIT to Run Tests

After a tester has written automated portlet tests, he/she is able to compile, deploy and invoke all of them using a custom ANT [6] command that we developed. At the end of the script run, test results are displayed in the script window and the browser. The Ant based script can be integrated with the regular build process and promotes regression testing. The ICT accounts portlet tests for demo usage scenario were deployed and run in the IBM Websphere Portal Server [11] environment.

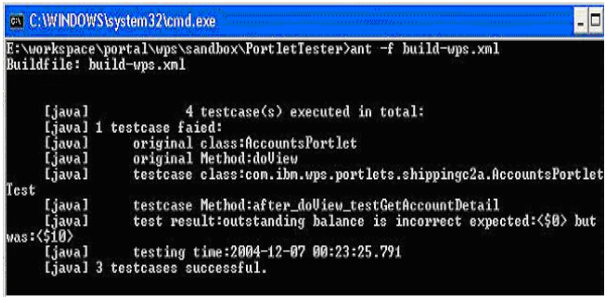

Fig. 9. Results of Test Execution of AccountsPortletTest Cases

6 Related Work

To our best knowledge, WIT is the only framework at this time that supports in-container testing of portlet-based applications. Other alternate approaches such as those provided by Cactus [7] can test Servlets [12], EJBs [13] and JSPs [14], etc. Portlets cannot be tested using the Cactus framework.

Further, the in-container testing approach used by the Cactus framework is more restricted than ours. Components, like Servlets, tested using Cactus are instantiated as normal classes in the test code versus using the real container to manage the component's lifecycle. Thus, they actually do now run in the same environment as when they are deployed. The limitation of this approach is that although the tests run in a real container, some in-container methods and its interactions with the real container cannot be completely tested as the services provided by the real container are being incompletely used. Thus Cactus tests may not be able to adequately detect deployment related errors as well as security issues (refer Section2 – (a), (b)) which can be tested effectively using our approach.

Client side testing frameworks such as httpUnit [8] and jWebUnit [9] support is more geared towards black box testing in a web environment. It can easily query the server externally and analyze the responses received. The frameworks, however, do not give a detailed control over the environment and constructing an initial state for the test is time consuming and often involves multiple http requests.

The Mock Objects approach [10] is another complimentary strategy to in-container testing of methods. In essence, it fakes implementation of the services provided by the container by using simulated objects. The main goal of mock objects is to unit test a method in isolation of domain objects by using simulated copies instead of real objects. Mock Objects suffer from the drawback that they do not assure that the in-container methods will run correctly when deployed on the chosen container. They only allow for a fine grained testing of business logic of in-container methods independent of the real context in which they run.

7 Conclusions and Future Work

Within this paper, we first presented the need for automated testing support in areas where portal applications currently cannot be tested automatically. We then elaborated on the various problems encountered at deployment time. This established the requirement that the tests must execute in the real container in order to test application code using the services provided by the container. To address these problems we have provided automated testing support through the WIT framework. The paper discusses the design and implementation of the WIT framework followed by examples of its usage scenarios. The framework allows in-container testing of web portal applications and provides a way of detecting and debugging deployment and security related problems associated with portlets. Using WIT, some manual testing can be replaced by automated tests.

We developed the ICT testing approach using WIT for portlets due to the scale of problem reported by our industry partner. Future versions of the WIT framework will be provided with configurable settings to perform in-container testing of other container-based web components such as servlets, EJBs, Struts [15] etc. The results of our ongoing empirical studies validating the usability and usefulness of WIT shall be presented in the future.

References

1. Christian Wege, DaimerChrysler. Portal Server Technology. IEEE Internet Computing 2002.
2. JSR-000168 Portlet Specification. http://www.jcp.org/aboutJava/communityprocess/review/jsr168/ (Last Visited: February 11, 2005).
3. Portal Introduction-IBM. http://www-106.ibm.com/developerworks/ibm/library/i-portletintro (Last Visited: February 7, 2005).
4. Aspect Oriented Programming Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier and John Irwin Xerox Palo Alto Research Center; European Conference on Object-Oriented Programming(ECOOP),Finland June 1997.
5. AspectJ Eclipse Project http://eclipse.org/aspectj/ (Last Visited: February 7, 2005).
6. Apache ANT http://ant.apache.org/ (Last Visited: February 7, 2005).
7. Cactus Apache Jakarta Project http://jakarta.apache.org/cactus/ (Last Visited: January 29,2005)
8. Client Side Testing using HttpUnit. http://httpunit.sourceforge.net/ (Last Visited: February 7,2005)
9. Client Side Testing of web-applications using jWebUnit http://jwebunit.sourceforge.net/ (Last Visited: February 7,2005).
10. Mocks Objects. http://c2.com/cgi/wiki?MockObject (Last Visited: January 25,2005).
11. IBM Websphere Portal Zone http://www7b.software.ibm.com/wsdd/zones/portal/ (Last Visited: February 7, 2005).

12. JSR-000154 Java Servlet 2.4 Specification. http://www.jcp.org/aboutJava/communityprocess/final/jsr154/ (Last Visited: February 11, 2005)
13. EJB Specification. http://java.sun.com/products/ejb/docs.html (Last Visited: February 11, 2005)
14. JSR-000152 JavaServer Pages 2.0 Specification. http://www.jcp.org/aboutJava/communityprocess/final/jsr152/ (Last Visited: February 11, 2005)
15. The Apache Struts Web Application Framework. http://struts.apache.org/ (Last Visited: February 11, 2005)
16. Harpreet Bajwa , Wenliang Xiong, Frank Maurer: Evaluating Current Testing Processes of Web-Portal Applications. Proc of ICWE 2005.