



UNIVERSITY of LIMERICK
OLLSCOIL LUIMNIGH

Department of Electronic & Computer Engineering

DISTRIBUTED SYSTEMS GOURP PROJECT

Group ID: Group 08

Students ID:

Module Code: EE6052

**Module Name: WEB-BASED APPLICATION
DESIGN**

Academic Year: 2016/17

April 2017

1	Introduction.....	1
2	Protection from OWASP Top 10 vulnerabilities.....	2
2.1	Injection.....	2
2.1.1	Used Techniques.....	2
2.1.2	Injection Test.....	3
2.2	Cross-Site Scripting(XSS).....	5
2.2.1	Used Techniques.....	5
2.2.2	XSS Test.....	7
2.3	Missing Function Level Access Control.....	9
2.3.1	Used Techniques.....	9
2.3.2	Test.....	10
2.4	Cross Site Request Forgery (CSRF).....	11
2.4.1	Used Techniques.....	12
2.4.2	CSRF Test.....	12
3	Application deployment:.....	14
3.1	Base Environment.....	14
3.1.1	Install a virtual machine.....	14
3.1.2	Oracle's JDK.....	14
3.2	Glassfish server.....	14
3.3	Derby database.....	15
3.4	JDBC.....	16
3.5	JMS.....	18
3.6	Web Application Deployment.....	19
3.7	Log file.....	20
4	APPENDIX.....	21
5	REFERENCE.....	22
	Figure 2–1 Input Verification of Login.....	2
	Figure 2–2 Injection Input.....	4
	Figure 2–3 Injection Result.....	4
	Figure 2–4 XSS Protection Header.....	6
	Figure 2–5 Replacement of String Code.....	7
	Figure 2–6 XSS Test Input.....	7

Figure 2–7 XSS Test Result.....	8
Figure 2–8 Web Browser Protection Detection.....	8
Figure 2–9 Web Browser Protection Detection.....	9
Figure 2–10 Missing Function Level Access Control Test Input.....	11
Figure 2–11 Missing Function Level Access Control Test Result	11
Figure 2–12 CSRF Test - Login Legal Page.....	12
Figure 2–13 CSRF Test - Visit Malicious Page.....	13
Figure 3–1 Glassfish Server Admin Login	17
Figure 3–2 Setting JDBS Connection Pool.....	17
Figure 3–3 Setting the JDBC Resources.....	18
Figure 3–4 Setting JMS Destination Resources.....	18
Figure 3–5 Setting JMS MessageBean Factories.....	19
Figure 3–6 Deploying the Web Application.....	19
Figure 3–7 Testing the Web Application.....	20
Figure 3–8 Log File	20

1 Introduction

This project is Web-Based Application Design. The goal of the project is to learn how to design a web application with HTML, JSF and EJB, and how to protect the Web Application from OWASP Top 10 vulnerabilities.

This report contains countermeasures and testing regarding the A1 Injection, A3 Cross-Site Scripting, A7 Missing Function Level Access Control, and A8 Cross Site Request Forgery in the 2013 OWASP top 10 list, the web application deployment procedure, and some information about how to access the web page on the virtual machine in ECE server.

- **Environment of Development:**

J2EE:	J2EE 7 Web
IDE:	NetBeans 8.2
Database:	Derby
Server:	Glassfish 4.1.1

- **Environment of Deployment:**

VM OS:	Ubuntu 16.04
Linux Kernel	4.4.0
Database:	Derby 10.12.1.1
Server:	Glassfish 4.1.1

2 Protection from OWASP Top 10 vulnerabilities

2.1 Injection

Injection is the top 1 vulnerabilities according to OWASP report.

“Injection flaws, such as SQL, OS, XXE, and LDAP injection occur when untrusted data is sent to an interpreter as part of a command or query. The attacker’s hostile data can trick the interpreter into executing unintended commands or accessing data without proper authorization.”(owasp, p.7)

SQL Injection is a common attack from a web application. Attackers usually use the feature of login/register, search, etc. to embed some malicious SQL code.

2.1.1 Used Techniques

There are two techniques that are used in the project to prevent SQL injection. They are Input Validate and PreparedStatement.

- **Input Validate**

Users’ input must be checked via validate component of JSF. This component can check users’ input. If there are any illegal characters in the input field, the JSF will give some error message to users. For example, username input validator is <f:validateRegex pattern="[A-Za-z0-9]+"/>. It only allows letters and number to be input.

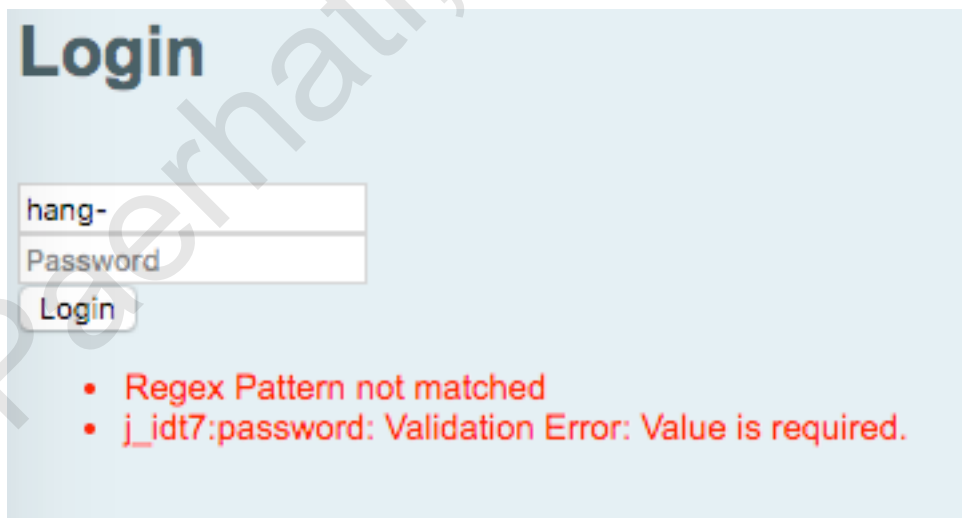
The image shows a web application login page with a light blue background. At the top left, the word "Login" is displayed in a large, bold, dark blue font. Below it, there are two input fields: the first contains the text "hang-" and the second is labeled "Password". A "Login" button is positioned below the password field. At the bottom of the form area, there are two red error messages: "Regex Pattern not matched" and "j_idt7:password: Validation Error: Value is required.".

Figure 2–1 Input Verification of Login

- **PreparedStatement**

PreparedStatement is a persistence API which executes SQL statement. Usually, the PreparedStatement is auto-generated from tables of database by IDE and many entities are generated at the same time.

```
@Entity
@Table(name = "USERS")
@XmlRootElement
@NamedQueries({
    @NamedQuery(name = "Users.findAll", query = "SELECT u FROM Users u")
    , @NamedQuery(name = "Users.findById", query = "SELECT u FROM Users u WHERE u.userId = :userId")
    , @NamedQuery(name = "Users.findByUsername", query = "SELECT u FROM Users u WHERE u.username = :username")
    , @NamedQuery(name = "Users.findByPassword", query = "SELECT u FROM Users u WHERE u.password = :password")}
    public class Users implements Serializable {...}
```

PreparedStatement can prevent SQL injection, because it can rewrite parameters when they are prepared for compile. For example, users input some malicious code when they want to search products. The PreparedStatement will rewrite ' to ', which prevent the possible of injection.

Malicious code: a' or 1=1

Rewrite result: a'' or 1=1

- **Replacement of String Value**

Please see Protection of XSS Replacement of String Value.

2.1.2 Injection Test

Using above techniques, SQL injection is successfully prevented.

- **Write an SQL injection code to search field.**

a' or 2=2

List

1..9/9



Picture	ProductId	ProductName	Price	Description	Quantity in Stock	
	1	mouse	12.0	mouse	1	Add
	2	keyboard	12.0	keyboard	1	Add

Figure 2-2 Injection Input

- Execute the malicious code.

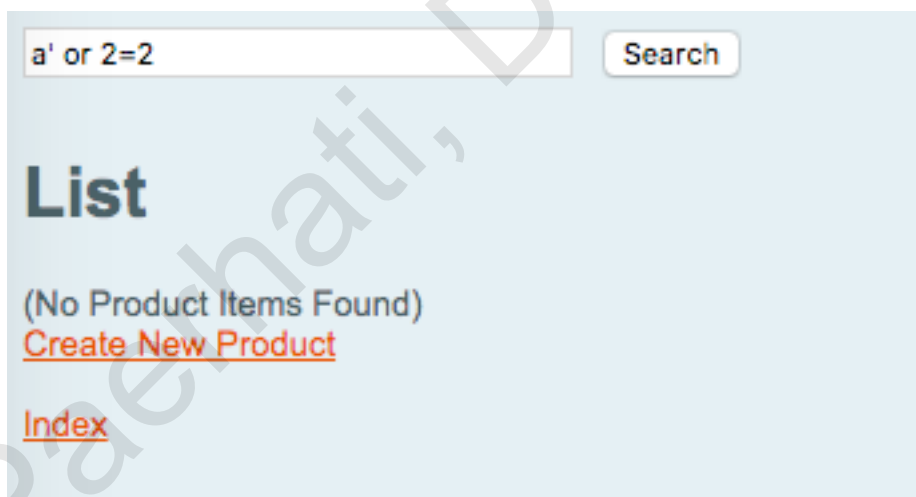


Figure 2-3 Injection Result

- Test result

The malicious code can not be executed, injection failed. If it is executed successfully, the page would display all products.

2.2 Cross-Site Scripting(XSS)

Cross-Site Scripting (XSS) has two main attack ways. One is like a special injection, but it different from SQL injection. The other is embed malicious code into URL request.

For the first attack mechanism, attackers usually write some html tags or javascript into input field or area to attack the web application. The second is similar to the first one, but attackers will write some malicious code as parameters of URL request.

“XSS flaws occur whenever an application includes untrusted data in a new web page without proper validation or escaping, or updates an existing web page with user supplied data using a browser API that can create JavaScript. XSS allows attackers to execute scripts in the victim’s browser which can hijack user sessions, deface web sites, or redirect the user to malicious sites.” (owasp, p.7)

XSS commonly uses privilege of application to load malicious code or access resources in the web application server.

2.2.1 Used Techniques

- **Using Post Not Get**

Hide request parameters is a solution that avoid XSS attack. Post request puts the parameters in http package, but Get append the parameters into the URL. So Post is safer way to pass the parameters than Get. In this case, all parameters used Post to transmit.

- **Response Header (X-XSS-Protection)**

X-XSS-Protection response header is a feature of major web browsers that can effectively prevent XSS attack. Adding a filter of servlet to reset http response header.

Java Code:

```
public class HeaderFilter implements Filter {
    @Override
    public void init(FilterConfig fc) throws ServletException {}

    @Override
    public void doFilter(ServletRequest req, ServletResponse res,
        FilterChain fc) throws IOException, ServletException {
        HttpServletResponse response = (HttpServletResponse) res;

        // 1, is enable xss protection,
        // mode=block, The browser will prevent rendering of the page
        // if an attack is detected.
        response.setHeader("X-XSS-Protection", "1; mode=block");
        fc.doFilter(req, res);
    }
}
```



```

@Override
public void destroy() {}
}

```

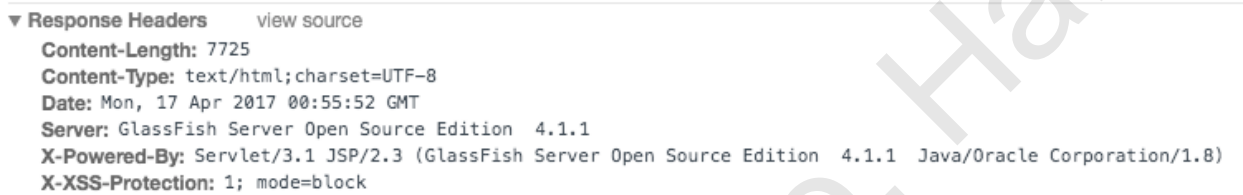
web.xml:

```

<filter>
  <filter-name>HeaderFilter</filter-name>
  <filter-class>com.shop.JavanBean.HeaderFilter</filter-class>
</filter>
<filter-mapping>
  <filter-name>HeaderFilter</filter-name>
  <url-pattern>/*</url-pattern>
</filter-mapping>

```

When login page is requested, the browser gets the response header.



The screenshot shows the 'Response Headers' section of a browser's developer tools. The headers listed are: Content-Length: 7725, Content-Type: text/html; charset=UTF-8, Date: Mon, 17 Apr 2017 00:55:52 GMT, Server: GlassFish Server Open Source Edition 4.1.1, X-Powered-By: Servlet/3.1 JSP/2.3 (GlassFish Server Open Source Edition 4.1.1 Java/Oracle Corporation/1.8), and X-XSS-Protection: 1; mode=block.

Figure 2–4 XSS Protection Header

- **Input Validate**

It is the same as preventing Injection.

- **JSF escape**

JSF is a view framework of MVC of J2EE. It includes some security mechanism. Preventing XSS is included in its security mechanism.

The JSF html tag includes an attribute which is named “escape”. Its default value is true, which means this tag does not execute web code.

- **Replacement of String Value**

Because of validator and JSF, there is no requirement of other frameworks or javascript to prevent XSS from web application, but some users’ input still needs to be processed then stored into database. For example, the message of user is included in the processing range, because there is not enough validator to control the content of user’s input.

When a user edits his/her or others’ profile, he/she can type some sensitive characters to message area to attempt to attack the web application. So these characters have to be replaced with harmless characters. Finally, the harmless characters are stored in database or be used.

```

public static String getCleanString(String value) {
    if (value==null || value.equals("")) return "";
    value = value.replaceAll("<", "&lt;").replaceAll(">", "&gt;");
    value = value.replaceAll("\\(", "(").replaceAll("\\)", ")");
    value = value.replaceAll("'", "'");
    value = value.replaceAll("eval\\((.*)\\)", "");
    value = value.replaceAll("[\\\"\\\\\\\\']\\\\s*javascript:(.*)[\\\"\\\\\\\\']", "");
    value = value.replaceAll("script", "");
    return value;
}

public static String getOriginalString(String value) {
    if (value==null || value.equals("")) return "";
    value = value.replaceAll("&lt;", "<").replaceAll("&gt;", ">");
    value = value.replaceAll "(", "\\(").replaceAll ")", "\\)";
    value = value.replaceAll "'", "'";
    return value;
}

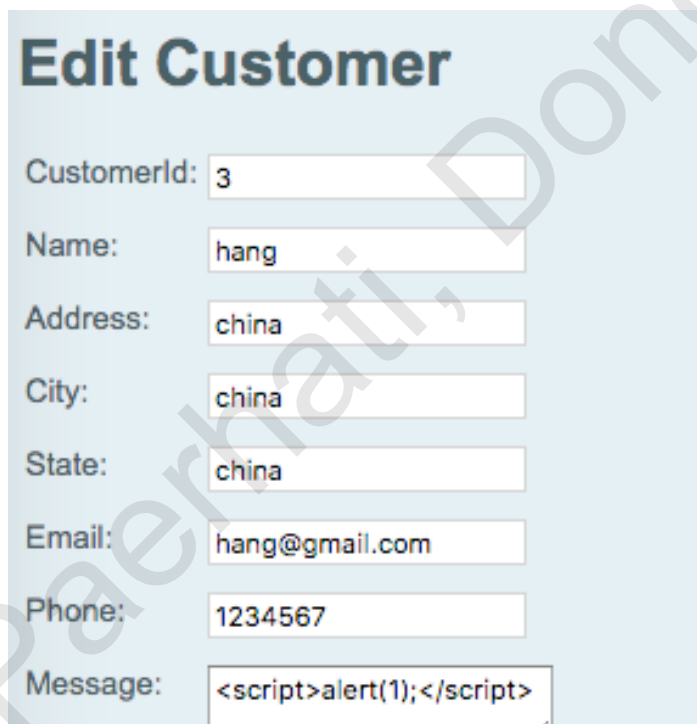
```

Figure 2–5 Replacement of String Code

2.2.2 XSS Test

Because of Post transmitted parameters, there is no need to test XSS attack via URL.

- Write a javascript into message area and save.



Edit Customer

CustomerId:

Name:

Address:

City:

State:

Email:

Phone:

Message:

Figure 2–6 XSS Test Input

- View the saved result.

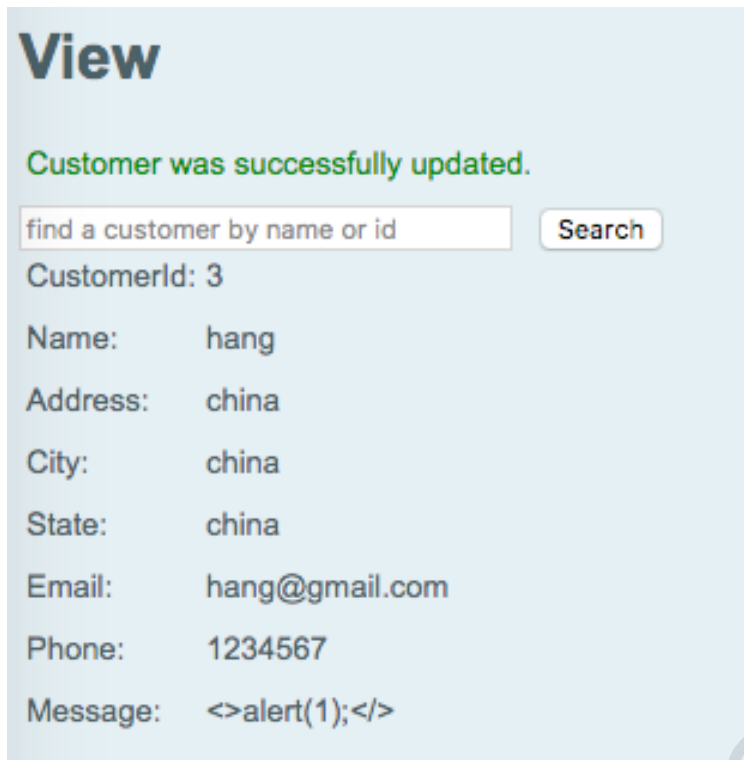


Figure 2-7 XSS Test Result

- Using OWASP-ZAP to discover XSS vulnerability.

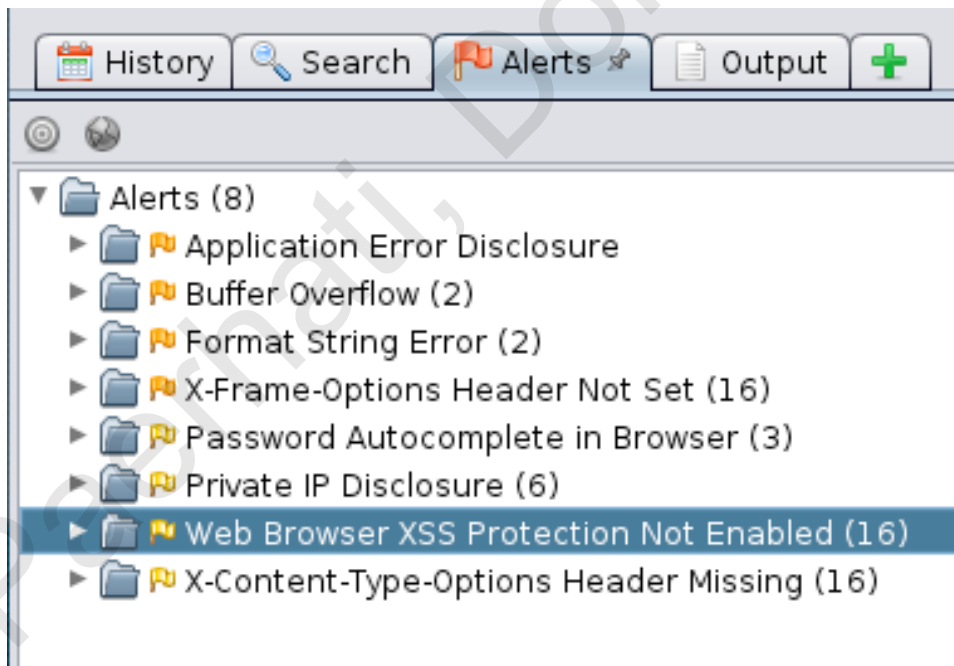


Figure 2-8 Web Browser Protection Detection

There are 16 URI that can not be protected by browsers.

- After setup the filter.

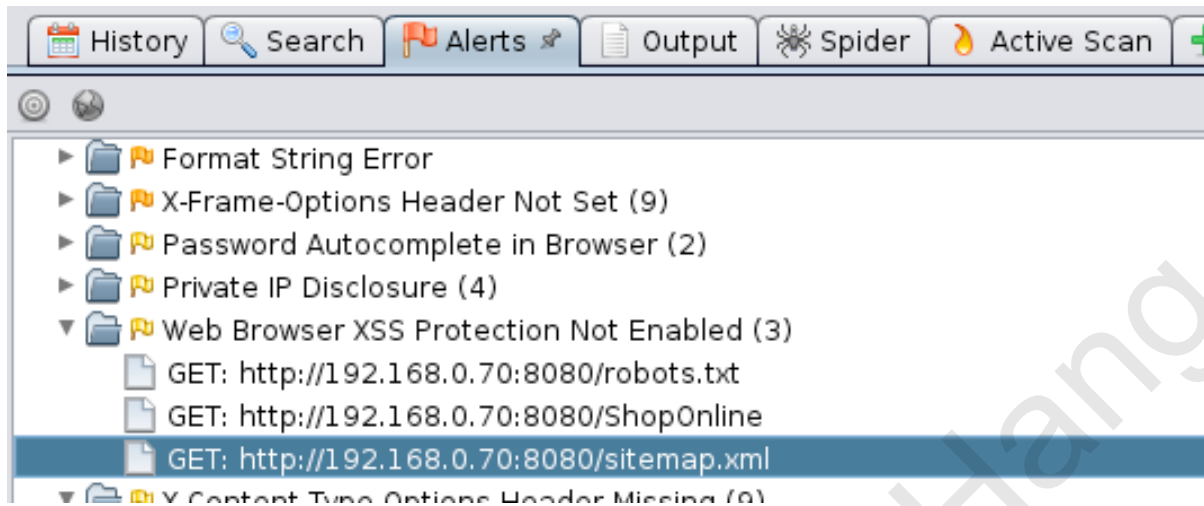


Figure 2-9 Web Browser Protection Detection

There are only three URIs detected and they have low risk level.

- Test result

It is safe to store and display the message. There is not any XSS attack which is executed.

Although there are three warnings about XSS Protection not enabled, the risk is reduced. The risk URIs only leave resource files.

2.3 Missing Function Level Access Control

Anyone can access the resource or web page by URIs without authentication or authorization. This threat is easy and dangerous. For preventing this attack, web application needs to protect URIs or check the authorization of visitors.

2.3.1 Used Techniques

- Configure the navigation rule in *faces-config.xml*.

Navigation rules are some rules which allow pages to redirect in each other via an action. If there is no rule, the redirect action will be intercepted.

```
<from-view-id>/index.xhtml</from-view-id>
<navigation-case>
  <from-action>#{productController.prepareCreate}</from-action>
  <from-outcome>success</from-outcome>
  <to-view-id>/product/Create.xhtml</to-view-id>
</navigation-case>
</navigation-case>
```

```

<from-action>#{productController.prepareCreate}</from-action>
<from-outcome>failure</from-outcome>
<to-view-id>/index.xhtml</to-view-id>
</navigation-case>

```

This is a sample which allow *index.xhtml* to navigate to */product/Create.xhtml* via “productController.prepareCreate” action.

- **Add phase listener to intercept illegal request.**

PhaseListener is a class of “javax.faces.event”. It is used to process something in JSF lifecycle.

When a user wants to visit a web page, “beforePhase” method will be called before the JSF page rendering. Adding an interceptor here can filter illegal request.

```

// limit anonymous only visit Login & LoginFailure
if (userType < 0) {
    if (!viewid.equals("/users/Login.xhtml") && !viewid.equals("/users/LoginFailure.xhtml"))
    {
        try {
            fc.getExternalContext().redirect(getURI("/users/Login.xhtml"));
        } catch (IOException ex) {
            Logger.getLogger(ShopPhaseEvent.class.getName()).log(Level.SEVERE, null, ex);
        }
    }
}

```

This is a sample which redirects anonymous visit to login page.

- **Render the content for users who have correct authorization.**

“render” is an attribute of JSF component tags. It is useful to process controller of authorization. “render=true” means render the UI component.

```

<h:form style="float:right;padding-right: 10px;" rendered="#{sessionScope.user != null}">
    <h:commandLink action="#{customerController.goHomePage}">
        <h:outputLabel value="Hi,#{usersController.username}"/>
    </h:commandLink>
</h:form>

```

This is a sample which renders the page when a user has logged in.

2.3.2 Test

- **Visit the user’s edit page without login.**

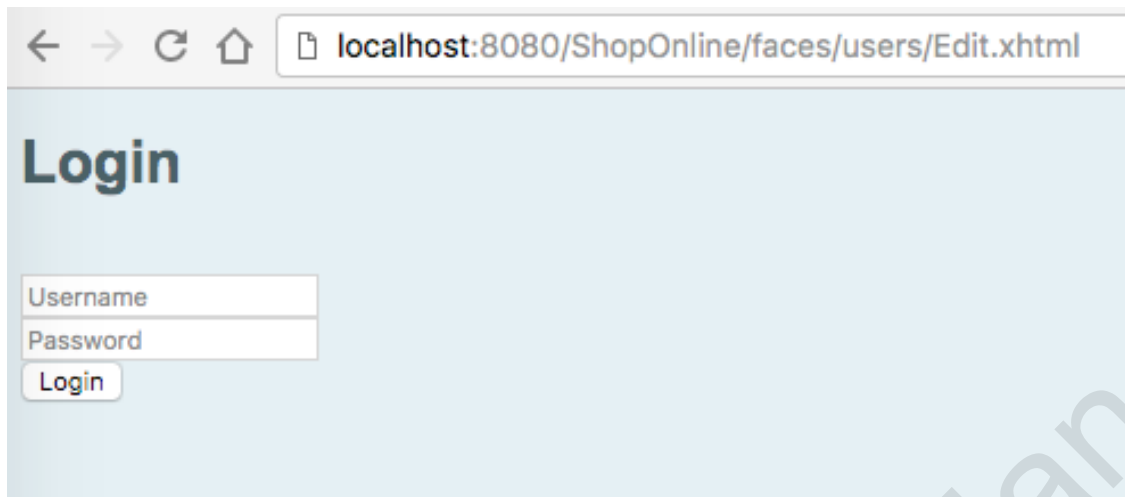


Figure 2–10 Missing Function Level Access Control Test Input

- Redirect to login page.

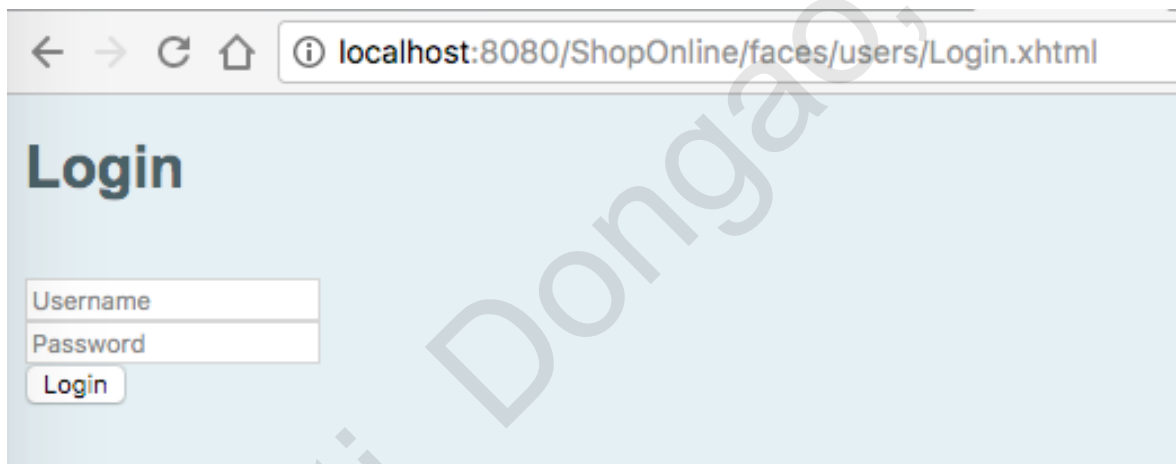


Figure 2–11 Missing Function Level Access Control Test Result

- Test result

The user's edit page is a sample to test the vulnerability. All pages must be given authorization to different role of users. Users only visit their own pages.

2.4 Cross Site Request Forgery (CSRF)

CSRF is an attack which forces users to do some actions that they do not want to. Usually, attackers offer a malicious web-site to users.

“A CSRF attack forces a logged-on victim's browser to send a forged HTTP request, including the victim's session cookie and any other automatically included authentication information, to a vulnerable web application. Such an attack allows the attacker to force a victim's browser to generate requests the vulnerable application

thinks are legitimate requests from the victim.” (owasp, p.7)

2.4.1 Used Techniques

- JSF 2.2 (oracle)

After JSF 2.0, the JSF has support to prevent CSRF attack in the framework. Developers only need to configure *faces-config.xml* file. The web application can prevent CSRF attack.

```
<protected-views>
  <url-pattern>/*</url-pattern>
</protected-views>
```

2.4.2 CSRF Test

- Write a malicious web page using PHP, which tries to visit */users/Edit.xhtml*.

```
<?php
$url = 'http://localhost:8080/ShopOnline/faces/users/Edit.xhtml';
$ch = curl_init();
curl_setopt($ch, CURLOPT_URL,$url);
$result= curl_exec ($ch);
curl_close ($ch);
echo $result;
?>
```

- Login with a customer id.

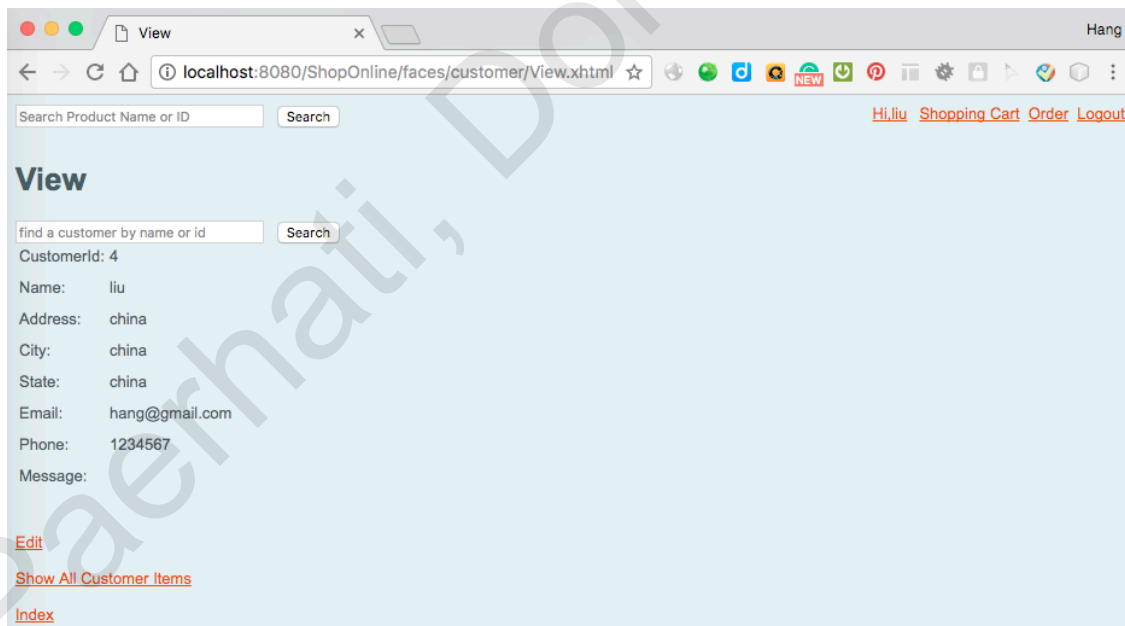


Figure 2–12 CSRF Test - Login Legal Page

- Load malicious PHP page.

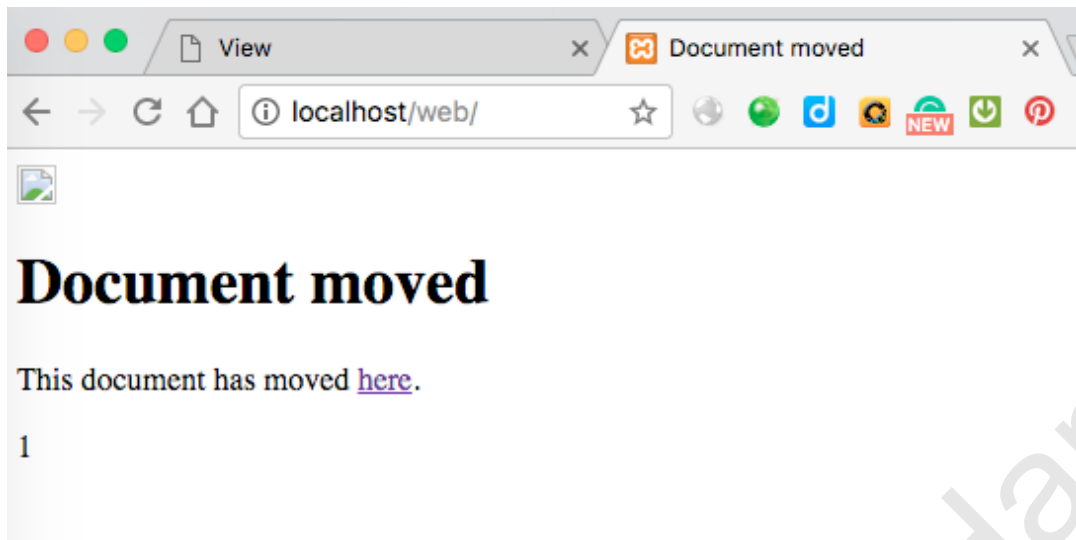


Figure 2–13 CSRF Test - Visit Malicious Page

The request page has moved, means that the attack has been intercepted. If click the link 'here', the page will be redirected to login page. The attack has failed, because the malicious web-site can not get the session which saves user's information.

- **Test result**

The web application has been protected by JSF from CSRF.

3 Application deployment:

3.1 Base Environment

3.1.1 Install a virtual machine.

System: Ubuntu 16.04 LTS

Linux Kernel: 4.4.0

3.1.2 Oracle's JDK.

1. Download and install, version: 1.8.0, location /usr/lib/jvm/jdk1.8.0.
2. Configure system environment variables for JDK and Glassfish.

- Append following line into file /etc/bash.bashrc:

```
export GLASSFISH_HOME=/home/student
export JAVA_HOME=/usr/lib/jvm/jdk1.8.0
export PATH=$PATH:$JAVA_HOME/bin:$GLASSFISH_HOME/bin
```

- Append following line into file /etc/environment

```
JAVA_HOME=/usr/lib/jvm/jdk1.8.0
AS_JAVA=/usr/lib/jvm/jdk1.8.0
```

3.2 Glassfish server

1. Download and install, version: 4.1.1, location /home/student/glassfish.
2. Setup an init script for Glassfish to startup on system boot.

- Create file /etc/init.d/glassfish with following content:

```
#!/bin/sh
### BEGIN INIT INFO
# Provides:          glassfish
# Required-Start:    $local_fs $network
# Required-Stop:     $local_fs
# Default-Start:     2 3 4 5
# Default-Stop:      0 1 6
# Short-Description: glassfish
# Description:       glassfish server
### END INIT INFO

#to prevent some possible problems
export AS_JAVA=/usr/lib/jvm/jdk1.8.0

GLASSFISHPATH=/home/student/bin

case "$1" in
start)
echo "starting glassfish from $GLASSFISHPATH"
sudo -u student $GLASSFISHPATH/asadmin start-domain domain1
;;
restart)
```

```

$0 stop
$0 start
;;
stop)
echo "stopping glassfish from $GLASSFISHPATH"
sudo -u student $GLASSFISHPATH/asadmin stop-domain domain1
;;
*)
echo $"usage: $0 {start|stop|restart}"
exit 3
;;
esac
:

```

- Add the above file into system's default runlevel with following command:

```

sudo chmod a+x /etc/init.d/glassfish
sudo update-rc.d glassfish defaults

```

3.3 Derby database

1. Download and install Derby database, version 10.12.1.1, location /home/student/derby.
2. Configure system environment variables for Derby database.

- Append following line into file /etc/bash.bashrc:

```

export DERBY_HOME=/home/student/derby/db-derby-10.12.1.1-bin
export PATH=$DERBY_HOME/bin:$PATH
export
CLASSPATH=$DERBY_HOME/lib/derbyclient.jar:$DERBY_HOME/lib/derbytools.jar:$CLASSPATH

```

3. Setup an init script for Derby database to startup on boot.

- Create file /etc/init.d/derby with following content:

```

#!/bin/sh
### BEGIN INIT INFO
# Provides:      derby
# Required-Start: $local_fs $network
# Required-Stop:  $local_fs
# Default-Start:  2 3 4 5
# Default-Stop:   0 1 6
# Short-Description: derby
# Description:    derby server
### END INIT INFO

DERBYPATH=/home/student/derby/db-derby-10.12.1.1-bin

case "$1" in
start)
echo "starting Derby Database from $DERBYPATH"
sudo -u student $DERBYPATH/bin/startNetworkServer -h 0.0.0.0
;;
restart)
$0 stop
$0 start
;;

```

```

stop)
echo "stopping Derby Database from $DERBYPATH"
sudo -u student $DERBYPATH/bin/stopNetworkServer -h 0.0.0.0
;;
ij)
echo "entering Derby Database command prompt from $DERBYPATH"
sudo -u student $DERBYPATH/bin/ij
;;
*)
echo $"usage: $0 {start|stop|restart}"
exit 3
;;
esac
:

```

- Add the above file into system's default runlevel with following command:

```

sudo chmod a+x /etc/init.d/derby
sudo update-rc.d derby defaults

```

4. Add database 'shop' for the web application using Derby ij command line tool.

- Open Derby ij command line tool with command

```
/etc/init.d/derby ij
```

- Create database 'shop' with command

```
connect 'jdbc:derby:shop;create=true';
```

5. Import data into database shop.

A file named export.sql contains all the SQL statements for creating the tables and inserting data into tables, the file can be used directly in the derby ij command line:

```
run '/home/student/6052web/web/export.sql';
```

3.4 JDBC

1. Startup the derby database and restart glassfish server:

```

/etc/init.d/derby start
/etc/init.d/glassfish restart

```

2. Open the Glassfish server administration web page.

Login the Glassfish server (<http://localhost:4848>) with username and password 'admin', 'admin'.

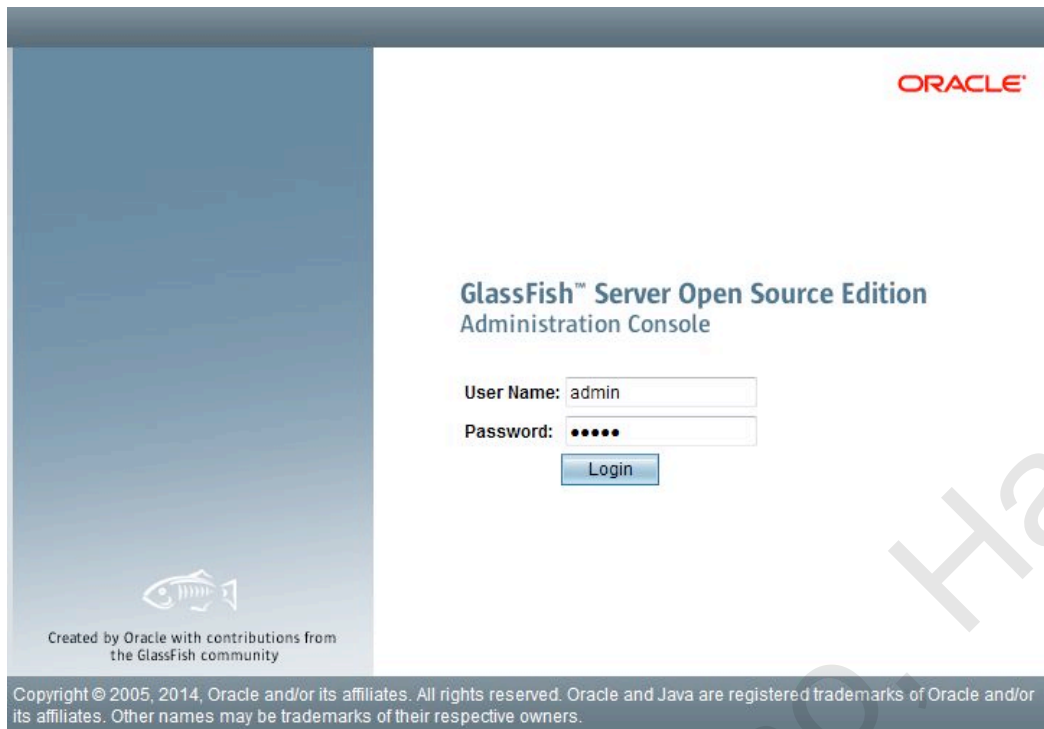


Figure 3–1 Glassfish Server Admin Login

3. Set the JDBC Connection Pool.

Click the new button to add a JDBC connection pool called DerbyPool.

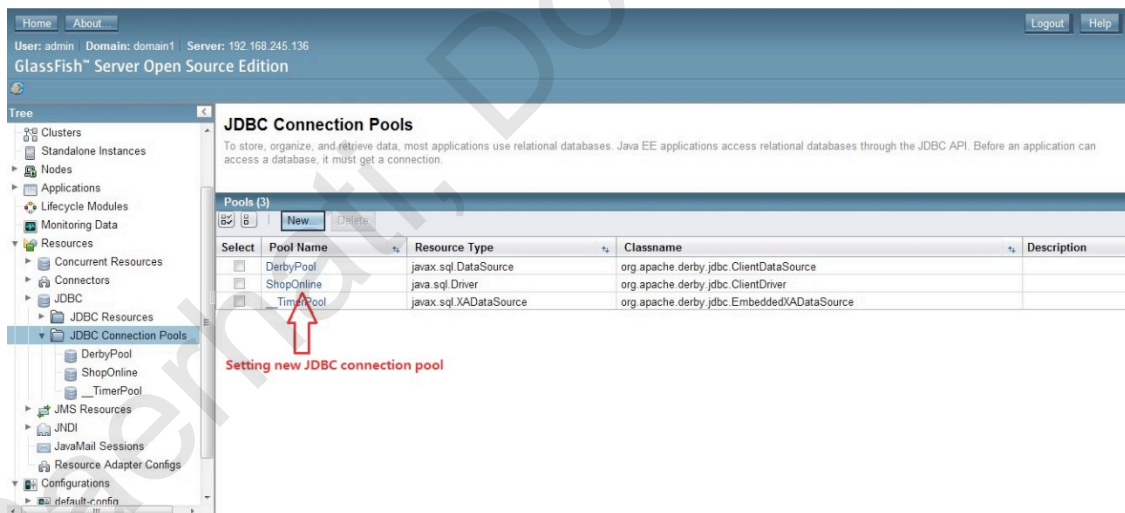


Figure 3–2 Setting JDBS Connection Pool

4. Set the JDBC Resources.

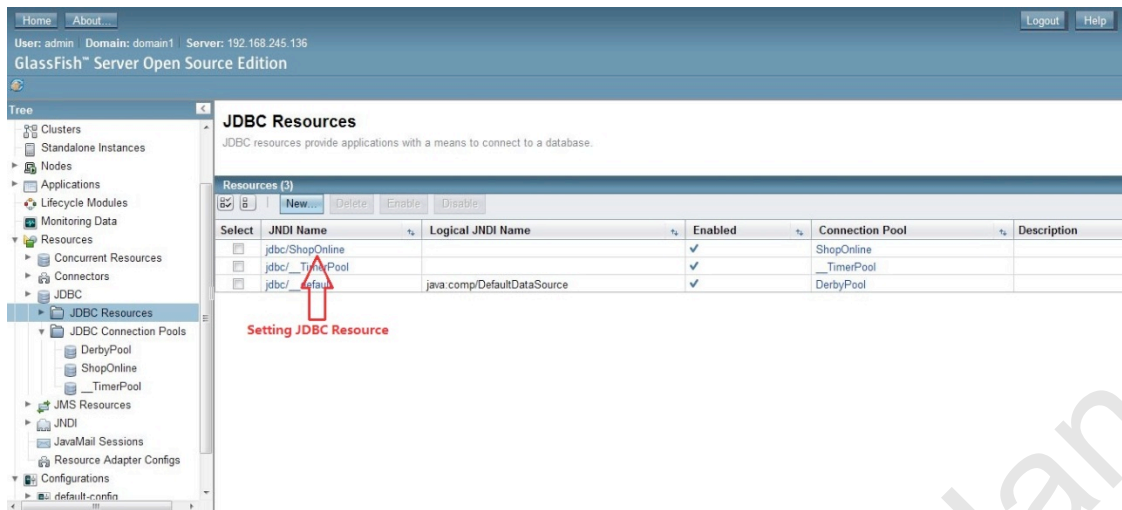


Figure 3–3 Setting the JDBC Resources

3.5 JMS

1. Setting JMS Destination Resources

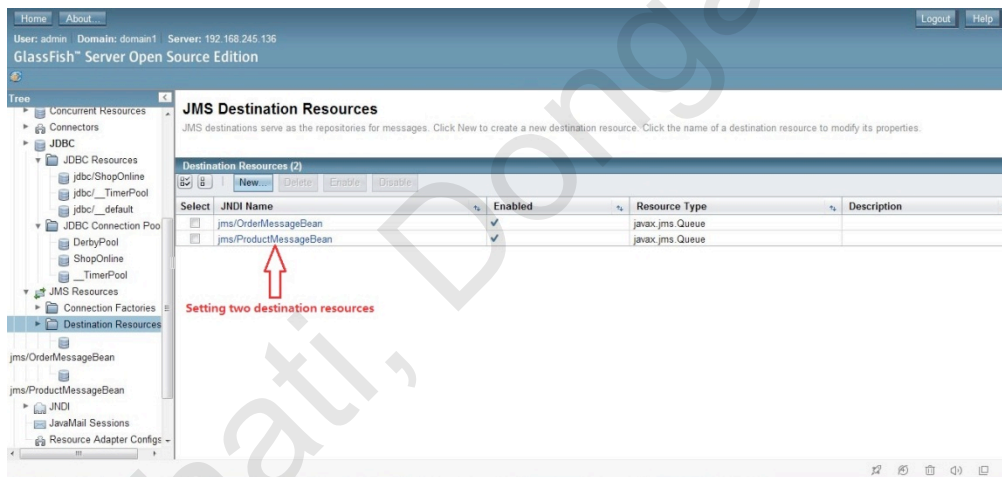


Figure 3–4 Setting JMS Destination Resources

2. Setting JMS connection factories

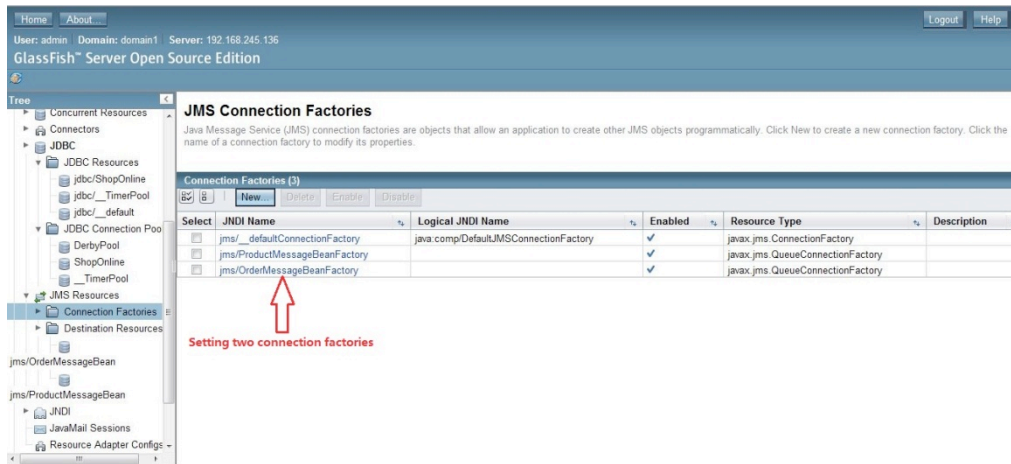


Figure 3–5 Setting JMS MessageBean Factories

3.6 Web Application Deployment

Click the deploy button and choose the .war file to deploy the application, and then click the launch link to test the application, or visit the <http://localhost:8080/ShopOnline>.

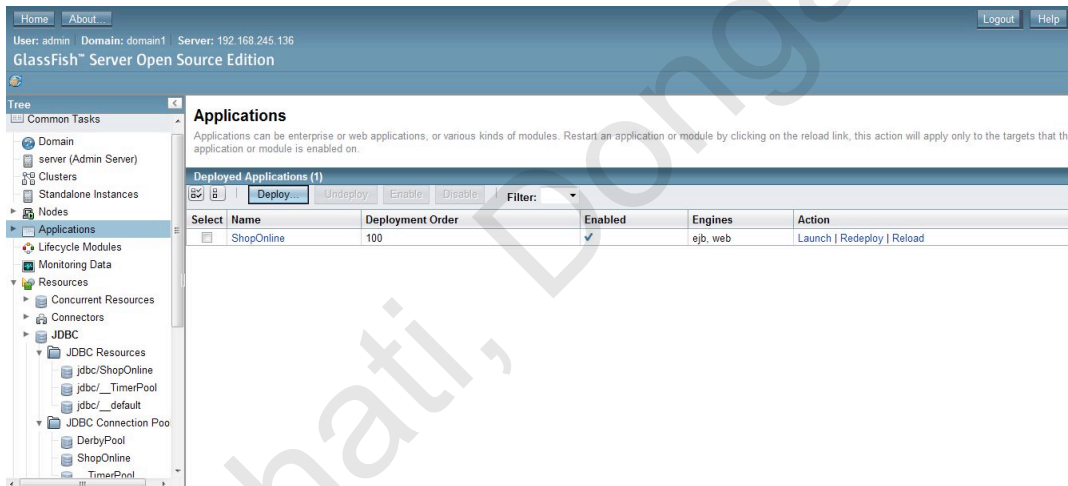


Figure 3–6 Deploying the Web Application

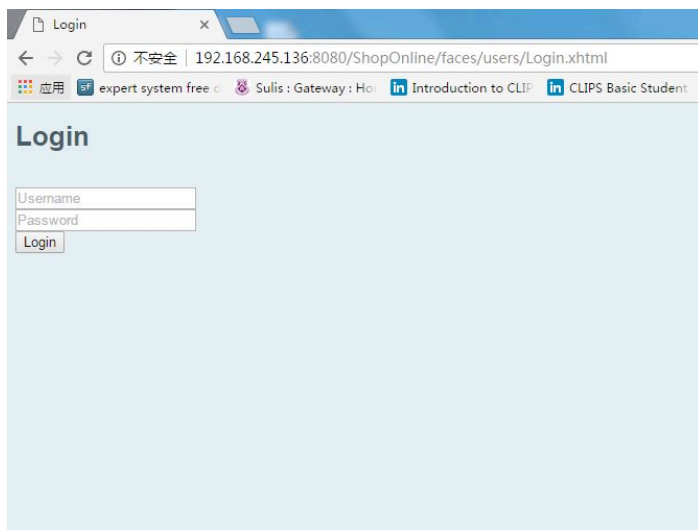


Figure 3-7 Testing the Web Application

3.7 Log file

The log messages are saved in the file package: /home/student/glassfish/domains/domain1/config, the log file is shop-app.log.

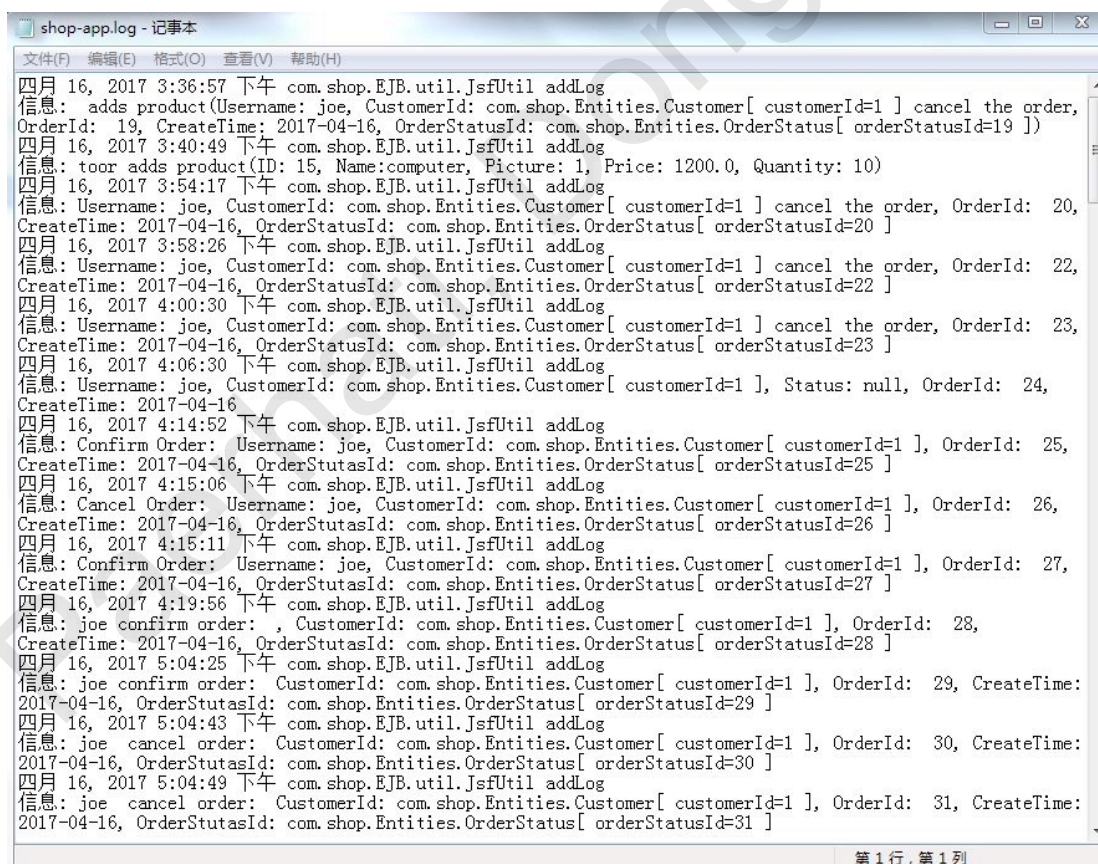


Figure 3-8 Log File

4 APPENDIX

Accessing the web page

1. The login password for the virtual machine is 'student' without the single quote.
2. The web page can be accessed with URL <http://localhost:8080/ShopOnline>.
3. If the login page returns 'password incorrect', it is because of the database startup issues.
Open a terminal and insert command `/etc/init.d/derby restart &` would restart the database and keep it running at the background.
4. The command for restarting the server is `/etc/init.d/glassfish restart`.

5 REFERENCE

1. oracle [online], available:
<http://www.oracle.com/webfolder/technetwork/tutorials/obe/java/JSF-CSRF-Demo/JSF2.2CsrfDemo.html> [accessed 18 April].
2. owasp [online], available:
[https://raw.githubusercontent.com/OWASP/Top10/master/2017/OWASP Top 10 - 2017 RC1-English.pdf](https://raw.githubusercontent.com/OWASP/Top10/master/2017/OWASP%20Top%2010%20-%202017%20RC1-English.pdf) [accessed 18 April 2017].