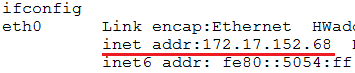
VM Deployment Exercise

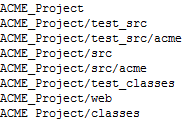
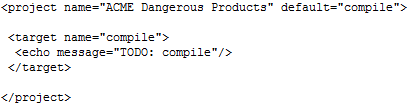
# CSCI 4300

**Introduction**: until now, all of our exercises and assignments have run on our own computer. In order to have our applications visible on the Web, we must deploy them on a Web server. This exercise uses the ANT software development scripting tool on a virtualUnix server, nike.cs.uga.edu, running the Apache Tomcat Java Web server. Other deployment options include Windows servers, Tomcat or JBoss Java Web servers, and the Maven software development scripting tool.

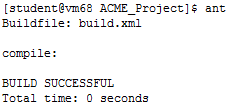
1. Get or install an ssh client program on your computer. If using a Windows machine, I recommend **WinSSH**, available via the magic of UGA site licensing from the friendly folks at Enterprise Information Technology Services, <http://sitesoft.uga.edu>. For Macs, you can use the ssh command on your underlying Unix OS, accessible via the Terminal application.
2. Using your ssh client, log into your CS account at **nike.cs.uga.edu** using the login and password given by the instructor. Note: nike is behind the UGA firewall, so if attempting this from outside the UGA network it will fail, unless you use the UGA VPN Tunnel software (also available at http://sitesoft.uga.edu).
3. Having logged into nike, you must now log into your virtual machine (VM), using the **student** login name. Please see the TA for this course if you don’t yet have your VM.
4. This double hop makes it difficult to upload files from your local computer. Use the **ifconfig** command to check find your IP address, as shown. You can learn about all these other cool configuration values in your networking class!
5. Now go back to WinSSH (or the SSH command from the Mac terminal) and login to the IP address instead of the vmxx hostname. Same username and password. Now you have a direct link to your VM, which is useful when you need to upload files.
6. Do we have the ANT software installed? Type the command

**which ant**

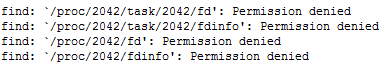
– this will show that ANT is, indeed, available. Let’s go to work!

1. First, let’s set up a development environment on our VM. Traditionally, you will create a separate project directory for each project. Create the following directories in your home directory. If we follow the pattern of providing subdirectories to match package names, the ANT build process will go more smoothly.
2. Also, we create separate directories for Java source code and Java class files, and separate pairs of directories for the implementation code and source code. The missing directories **ACME\_Project/classes/acme** and **ACME\_Project/test\_classes/acme** will be automatically created when we compile the code.
3. Use the File Transfer tool on WinSSH to upload the implementation source files **LineItem.java** and **ShoppingCartBean.java** into **ACME\_Project/src/acme**. Upload the test source files **LineItemTest.java** and **ShoppingCartBeanTest.java** into **ACME\_Project/test\_src/acme**.
4. Now the real fun starts! Create a file named **build.xml** in your ACME\_Project folder with these contents. Note that every XML tag must have a matching closing tag, a rule that is not enforced in HTML. The <echo> tag is allowed to be its own closing tag, since it does not contain any content.

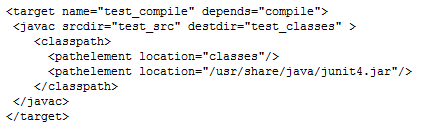
We’ve discussed the format of ANT build files in class; note that setting the default target to “compile” causes the “compile” target to run when you enter the “ant” command. From the shell, make sure you are in the ACME\_PROject folder and type **ant**. You may need to fix a few XML syntax errors, but eventually you should see this (rather Spartan) output telling you that the compile step has been run and the build was successful, but you still need to actually compile the code. Yay!

1. Now go to ant.apache.org and find the manual. Look up the Javac task in the manual and replace the <echo> task in your build file with a <javac> task. [[1]](#footnote-1)You only need to define the srcdir, destdir, and includeAntRuntime attributes. (You don’t need the runtime libraries, so set this to “no.” The <javac> task does not contain any other tags, so make it self-close like the echo command above.
2. Test by typing the “ant” command again. After fixing any XML syntax errors, you should get a message telling you that the compile command has been run, two Java files have been run, and the build was successful. Yippee!
3. Using the Unix ls command or your SCP client, check that the Java class files **LineItem.class** and **ShoppingCartBean.class** have been created and placed in the classes directory. Note that ANT created a subdirectory **acme** of **classes**. Again, Java source and class files go into folders matching their package names.
4. Run ant again. This time, the output tells us that the compile step has been run, but no files have actually been compiled! Just as its older cousin make does, ANT checks whether files that have already been compiled need to be re-compiled. In this case, since the class files are newer than their corresponding source files, the files are *not* recompiled.
5. Edit build.xml to add another target named **test\_compile** that compiles all the test code. Since the test classes call methods of the implementation classes, any changes in the implementation code should be recompiled before we compile the test code. Therefore, make the test\_compile target *depend* on the **compile** target.
6. Inside the **test\_compile** target tag, add a <javac> task to compile the test code. This will look just like the <javac> task in the compile target, except that the srcdir and destdir attributes will be test\_src and test\_classes, respectively. Also change the default target to **test\_compile** in the <project> tag on line 1. Test by running the command **ant test\_compile**.
7. Uh-oh! After correcting any XML errors, you will get a slew of errors about stuff being “not found.” This often means a classpath error. Examining the error messages, you will see that it is the implementations classes that are not found. These are in the *classes* directory, so add an attribute **classpath=”classes”** to the <javac> tag.[[2]](#footnote-2) Run the **ant test\_compile** command again.

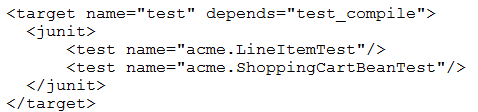
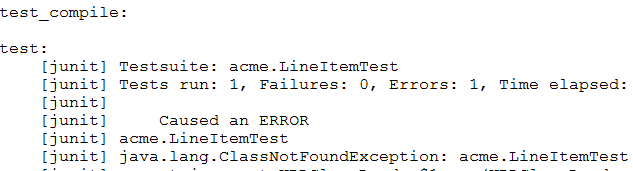
 Yikes! Now ant cannot find the JUnit framework classes themselves. [[3]](#footnote-3)Unlike the implementation classes that we just created, JUnit classes have been compiled already and will be in a jar file. (Java Archive file.) But where?

1. Part of the fun of administering Unix systems is figuring out where stuff is. The most basic tool is the **find** command. Check out the online manual for find by entering **man find**.
2. Now try to actually find the file**: find / -name junit\\*jar**. (We need to escape the wildcard character so the shell does not replace it by a list of all the files in the current directory.) As you can tell, this command will search the entire directory structure of your VM, starting at the root directory /, for files whose names start with **junit** and end with **jar**.
3. Unfortunately, this command generates hundreds of lines of error output , telling us about all the directories we don’t have permission to list. We could override this problem by promoting ourselves to be the root user, but I discourage doing this unless it is really necessary.[[4]](#footnote-4) Instead, alter the find command to discard all the error messages by sending them to /dev/null:

**find / -name junit\\*jar 2>/dev/null[[5]](#footnote-5)**

1. This list is a bit more manageable! All we need to do is pick one of the JUnit 4 jar files, since we probably built our JUnit code on JUnit 4. Since we now have two items to go into our classpath, we get rid of the classpath attribute and replace it with a classpath element:
2. Note that the <javac>element is no longer self-closing, because it now contains the <classpath> element. The <classpath> element is not self-closing either, but the <pathelement> elements are.

Please take a moment to look over the section entitled Path-like Elements in the Apache documentation for the Javac task. In general, the elements of classpaths are either directories or jar files, both of which can contain classes.

1. Run ant test\_compile again. This time the compile should succeed. If you get the error shown at right, make sure you have included org.junit.Test in your includes.
2.  Now it’s time to actually run the JUnit test. In your Web browser, look up the <junit> element. Check the list of attributes – none of them are required! According to the description above the attribute list, “Tests are defined by nested test or batchtest tags (see [nested elements](http://ant.apache.org/manual/Tasks/junit.html#nested)).” Go down to the nested elements list, and find the **<test>** nested element. Note that the only required attribute here is **name**, the name of the test class. So, set up a basic test target as shown at right.
3. Change the default target to **test** in the <project tag> on line 1, and run ant. You’ll see a message telling you that the test has failed, but no error message to tell you what went wrong! What’s up with that?
4. To control the output, use a **<formatter>**nested element. Since <formatter> is a nested element, it can come before or after the <test> element inside <junit>. Choose the “plain” type and set usefile to false, which will dump the output to the console.
5. Run **ant** again. Now you should see the error shown below, which is another classpath problem. This time the task needs two sets of classes: the implementation classes in the classes directory, and the test classes in the test\_classes directory. To add multiple locations to the classpath, use the nested **<classpath>** element, instead of the classpath attribute, as shown above. 
6. Run **ant** again. This time it should succeed!
7. Add **haltonfailure=”on”** to the <junit> tag. This will stop ant from continuing to the deploy step if there are any errors in the test step.

# Deploying the Web project

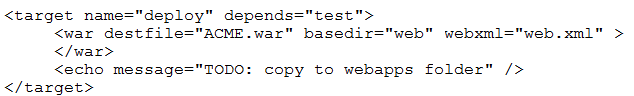
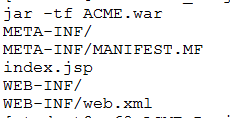
1. Now the real fun begins! We’re going to deploy the actual Web project. This has three steps:
   1. Package the project into a Web Archive (WAR) file;
   2. Copy this WAR file into the webapps directory on your Tomcat server;
   3. Restart the server.

Actually it’s possible to skip step c. by configuring Tomcat to automatically reload any new WAR files that we drop into our webapps directory. However, I can’t resist the chance to try out the Tomcat control tags that we just learned about from our student presentation.

Accordingly, let’s add a new target named deploy to our build.xml file. In the project flow, we shall make “deploy” depend on “test.” This means that any time we change our Java code, we should automatically run all the tests again, and we should not redeploy unless all the tests pass.

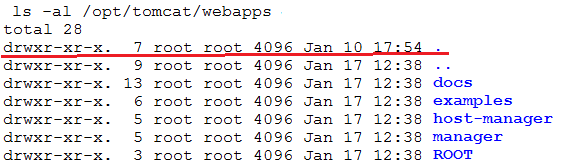


Continuing with our slow-and-steady approach, let’s start by just adding a stub target as shown above. It does not matter in which order the targets are arranged within the build file, but I usually put them in chronological order of execution.

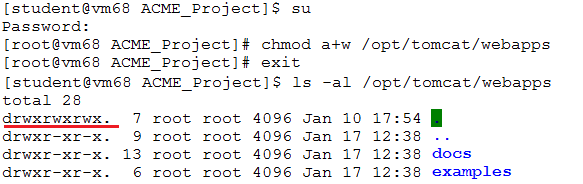
1. Now change the default target to “deploy” in the first line of build.xml. Save build.xml and run ant. You should see all the targets run. No files should be recompiled, but the tests will be re-run anyway.
2. Now it’s time to build the WAR file. These have a (fairly) complex specified layout that must be just so in order to work. Fortunately, ANT has a <war> task that organizes the files into precisely this order, regardless of the arrangement of files within your project directory!
3. First, use SCP to upload your index.jsp file into the web directory inside your project directory. Use SCP to upload web.xml into the project directory itself.
4. Look up the War task in the ant manual. Before we blindly forge ahead, look up the basedir, webxml, and destfile attributes. Let’s write a basic <war> task with just these attributes:
5. Note that we have scaled down the TODO comment!
6. Run ant again. You should see the TODO message, which means that the <war > task has succeeded! To check, examine the WAR file with the command jar –tf ACME.war.[[6]](#footnote-6) See! The WAR task has created the special META-INF and WEB-INF directories, and copied web.xml into the WEB-INF directory.
7. OK, lets copy this file to our webapps directory and see what happens:

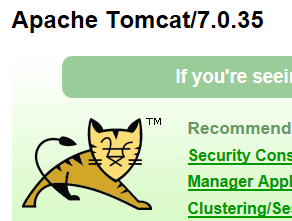
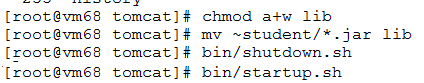
**cp ACME.war /opt/tomcat/webapps**

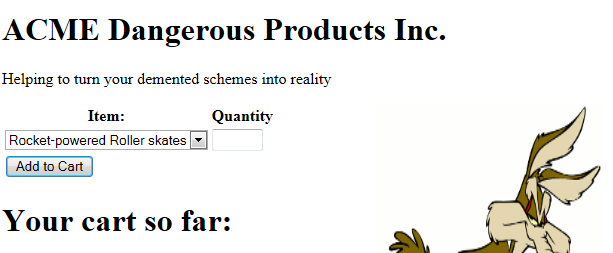
1. Oops! We get a permission denied error! Check this out with the appropriate ls command:



1. If you are not comfortable with Unix file permissions, this is a good time to catch up. The underlined line says that the file “.”, which is actually /opt/tomcat/webapps, is owned by root. Root has read, write, and execute permissions to this file, but others have only read and execute permissions. Please look up the ls command if this is Greek to you – if you get lost here, it is much more difficult to succeed in this course.
2. In order for the copy command to work, we need write permissions to the directory. So let’s give ourself write permission:
3. This is a predictable result, since we don’t own this file! The only recourse is to temporarily become root! Root’s password is the same as your own, original password (you did keep that paper, didn’t you?)



1. Note that the prompt changes to “#” when you are root. Do your job and exit back to being student as soon as possible! Remember that if you mess up your server, we’ll simply wipe the whole VM and you can start over.
2. Now try your copy command again: **!cp[[7]](#footnote-7)**
3. Your ACME application is now loaded into Tomcat. Lets’ start the server from the command line:
4. Launch a Web browser and navigate to <http://XXX:8080>, where “XXX” is the IP address you located earlier on in the exercise. You should see the Tomcat logo. Yippee!
5. Almost done, right? Well, we’ve got a few tweaks left. Now attempt to navigate to your ACME app at <http://XXX:8080/ACME>. You will see a message saying that the files needed to implement the JSTL are not found. Bummer!
6. Download the JAR files javax.servlet.jsp.jstl-1.2.1.jar and javax.servlet.jsp.jstl-api.1.2.1.jar from eLC to your local computer. Upload them to your home directory on the VM, then:Relo
7. Reload the web page one more time. It looks like the same old same old, but on closer examination you have graduated to a different error! We have fixed the JSTL problem; now the problem is that Tomcat can’t find the Bean files.
8. Exit from root, and return to the ACME Project work directory. Repeat the command jar –tf ACME.war. Note that the LineItem and ShoppingCartBean are not included in the war file!
9. Return to the ANT War task manual page and look for the classes nested element. This is where you put in the location of your class files. See if you can compose the necessary <classes> element (*Hint*: it’s a one-liner). Note that because Java maps packages to directories, the root directory specified should be **classes**, not **classes.acme**.
10. Now re-run ant and re-do the jar –tf command. You should now see the classes files. Note that the classes have been placed inside an acme subdirectory.
11. Cross your fingers, stroke your rabbit’s foot, and copy ACME.war to the Tomcat webapps folder. Reload the page and this is what you should see!



TO BE CONTINUED….

1. In general, always remove TODO messags when you begin to actually perform thetask. [↑](#footnote-ref-1)
2. Note that it is class files, not Java source files, that need to be provided in a classpath. [↑](#footnote-ref-2)
3. If you don’t get this error, I don’t know why! Read on… [↑](#footnote-ref-3)
4. Root access allows you to do all kinds of otherwise forbidden actions, some of which can completely trash your system. If you do get into an unrecoverable situation, we can always flush your entire VM and start over – not something you really want to do. [↑](#footnote-ref-4)
5. Standard file handles: 0 = standard input, 1=standard output, 2=standard error. /dev/null is a file handle that goes nowhere, provided for situations like this! [↑](#footnote-ref-5)
6. Check the jar documentation to learn the “t” and “f” options. [↑](#footnote-ref-6)
7. Repeat the last command that started with “cp” [↑](#footnote-ref-7)