EKSTAZI INTEGRATION WITH MAVEN PROJECTS

Technical Documentation

The project has been implemented using Java and Bash scripts. The following sections give an overview of the working of various sections of the code.

1. JAVA Code Overview

The java code has been written to modify pom.xml files with Ekstazi related settings/configuration. It consists of the following files:

ListDir.java :

This class contains the following function:

ListDirRecursively ( ): This function takes a Project Folder path as input, and recursively finds out the paths of all the pom.xml files present inside the folder. It is initially called with a path of the project folder and a depth parameter with value 1. The function will look at each file or folder and if the file is pom.xml or pom-main.xml it stores its path. If it is a folder, then, the function is called recursively by increasing the depth parameter by 1. We also maintain a max depth parameter so that recursion can be stopped at some depth i.e level of subfolders inside the root. By default this parameter is -1 which means recurse through all the subfolders to find out all the pom.xml files.

PomParser.java :

This class contains the code for modifying pom.xml file with Ekstazi related settings.
Input Parameters -> The code is called with command line arguments to be specified in the following order:

- Absolute/Relative path of project folder (Required)
- Depth parameter with value 1 or -1 (Optional)
- Ekstazi version (Optional)
- Surefire version (Optional)

Main ( ): This function is the entry point of our parser. This class is called with the input parameters defined above. It calls the 'ListDirRecursively' function from the ListDir class to get all the paths of all the pom.xml files in the project folder. Then, for each pom.xml in the project folder it calls the queryPom ( ) function described below along with the input parameters.

queryPom ( ): This function contains the main implementation of the parser. It is called with the path of the pom.xml file to be modified with Ekstazi related configuration. Here is the overall implementation of the code:

- First it tries to parse the xml file using the xpath library builder function. If the library is not able to parse the file, then the file is either not readable or properly xml formatted. In that case parser terminates run for that pom file.

- Then, the presence of <build> element is checked in the file. If it is not present we create a new one.

- Then, we check for the presence of <plugins> element in xml file. If it is not present we create a new one inside of <build> element.

- Then, we check for all the maven-surefire plugin entries present inside of the xml file.

- For each such occurrence, we trace the exact XSLT path of the surefire plugin entry. All these paths are stored. If no surefire plugin in the file, then a new plugin is created inside the <plugins> element.

- For each surefire plugin entry we check its version and upgrade it if necessary, either when its less than 2.13 or in the case where the user has specifically passed a surefire version as a command line argument to the code.

- Then the <argline> element is searched inside each surefire entry. The text of that tag is then prefixed by ${argLine}.

- Then, for each surefire plugin node, we add ekstazi plugin over it if its already not present. Plugin checks for the passed Ekstazi revision and changes b/w selection or select goal accordingly. If no Ekstazi version is passed, then 4.2.0 is assumed by default.

- Also, ekstazi dependency node is create inside dependencies node or dependenyManagement/dependencies node. If the dependency of ekstazi is already there, its version is checked with current version to see whether change in the version is required or not.

- <excludesFile> tag is created under surefire configuration. If the <configuration> element is not present then first <configuration> element is created followed by <excludesFile>.

- Then, the changes in the dom structures are written to a temporary file with certain indentation settings. Then, minimum diff code is called on that temporary file which writes the minimal diff on to the original pom and the temporary file is removed.

- The steps from 1-12 is called for each pom.xml file in that project folder.

MinimizeDiff.java:

This class contains the code for obtaining only the minimal difference from the temporary pom.xml file modified by PomParser.java and writing it the the original pom.xml file of the project. Here is how the code works:

First, both the files i.e. original pom.xml and modified pom.xml (modified by our code PomParser.java) are read line-by-line and their data is converted into ArrayList<String> format. Then these data structure is passed to the library which calculates the diff between 2 files. This delta information also contains the type of delta for each block that is different from the other file. Then, we check for the delta information and its content, and if we find it related to Ekstazi then we keep the delta otherwise we delete it. Then, we would have the pure delta that is only related to ekstazi modification. Using this delta information we patch the original file to get the minimal diff file.

2. **Bash Scripts Overview**

The scripts contain the code for fetching the projects and running maven tests on the project by integrating the project with Ekstazi. It internally calls the JAVA code described above to modify the pom.xml files inside the project, post which it runs the tests for the project.

run_ekstazi.sh:

The script contains the overall code for downloading and running maven tests on the project with Ekstazi. It runs the tests twice on the same version of the project, to check if the tests are getting reduced.

Input parameter: The script takes various input parameters which can be provided using command line switches

- Project Url (using switch –u)
- Project Folder Name to be created (using switch –p) - Optional
- Ekstazi version (using switch –v) – Optional. Currently we support 3.4.2 and 4.2.0.
- Modules to run (using switch –m) – Optional. Specify comma separated list of modules to run on.
- Revisions to run (using switch –r) – Optional. Specify comma separated list of revisions to run on.
- Depth parameter (using switch –d) – Optional. Specify depth 1 for modifying only root pom.xml, else -1 for all.

EKSTAZI INTEGRATION WITH MAVEN PROJECTS

Here is the overall working of the code:

- User provide different parameters to the script using switches. For example, -u “URL” -p “Folder” -v “ekstazi_version” -r “r1,r2,r3” -m “m1,m2” etc.
- Script then checks whether the url is of a git project or svn project by pinging each one of those one-by-one.
- Uses appropriate git/svn commands to clone into local folder.
- mvn install -DskipTests runs on this folder to install additional dependencies of the project.
- Then, this folder is copied into a .${project}_clone folder to keep a pure (w/o modifications) copy of it, so that it is not getting downloaded again using n/w resources. Also, in the next run, if it finds the cloned folder, it directly copies from that folder instead of downloading the whole project again.
- Then, the scripts loop over each revision, for each revision it first removes the changes of previous run if its there and checkout that specific revision otherwise use the latest revision.
- If the modules are specified, script loops over each module, otherwise just run on main module.
- Script then calls our pom_parser.jar file to modify pom files of the project using the parameters given by the user.
- It then calls the script ‘test_ekstazi.sh’ which is created separately to test the changes made for integrating Ekstazi by running the maven tests on the project.

test_ekstazi.sh:

- Script first deletes the .ekstazi folder present inside folder/subfolders.
- Then, script calls mvn test 2 times and it catches the output of those commands so as to fetch number of tests run. Also, it finds the time taken for each mvn test run.
- Script also looks for content in .ekstazi folders of the project and if the content inside of these folders is > 3 (for each), it will write module passed otherwise failed.
- In the end, it prints out the time taken and test reduced info on the screen as well as in logs.

Output: The scripts also outputs two log files described below:
ek_{$project_name}.log – This file contains the log ouput of the result of the test run. The results recorded by the script include: number of tests that ran each time, difference of test cases, time taken to run each time etc.

ekstazi_parser_log_{$project_name}.log – This file contains the output generated by the JAVA code, listing the modifications made to each of the pom.xm files in the project.

3. JAVA Unit Tests:

External Dependencies:

- Common-IO : Used for using its FileUtils module which gives the simple functions of file/folder copying/creating/deleting.

- Diff-Utils : Used to get the diff between 2 files same as we use diff command on linux CLI. It first requires file to be read in arraylist of string and then those 2 files are compared and the delta of 2 files are iterated to check what changes are required. Only those changes are kept and then, patched to the input file to get the output.

Junit Test Cases:

ListDir TestCases

- *testPomFIleCount()* : Our goal was to check whether the ListDirRecursively function in ListDir class was able to correctly tell all the pom.xml files in the passed directory. First, we created a test_lstdir folder and in that folder we created random number of folders(1-5) and again in those random folders we created random number of folders(1-5). Then, we just placed empty pom.xml in each of the folder and then, called the ListDir function to check whether the count of pom.xml on that test_listdir folder is correct or not.

- *testPomFilePresence()* : Apart from knowing the count of number of pom.xml in a folder, we would also like to know whether the path of pom returned are actually present in that folder. So, we created this testcase that checks whether the pom files path are actually path that are present and accessible.

PomParser TestCases

- *testOutputPOM()* → This is our main functionality. Here, we need to check whether each of the pom file in the folder are getting integrated with ekstazi as intended. For this, we used some demo files and created expected pom files manually by editing the pom.xml files as our tool should have done. This demo folder contains pom of different structure, in some cases there is no build, in some there is no surefire or plugin or dependencies. So, this give us very different types of pom on which our parser should give correct output results. So, we maintained some expected output files in the same format that our tool is expected to output. Then, we copied this demo folder into a demo_test folder and ran our parser on it. If the diff between resulting files and the expected files is clost to zero, we pass the testcase otherwise we fail it.

- *testMalformedException()* → There are some case where the xpath library that we are using is unable to parse the xml files. These are the cases where either the pom file is not readable or not formatted properly as xml, or it is empty. So, in those cases a SAXParserException is thrown. To check that we created a dummy pom.xml having empty content and called our parser function on it. If the SAXParserExeption is raised we pass the testcase otherwise we fail it.

MinimalDiff TestCases

- *testMinimal()* → We also need to check whether the final output of the parser is the best possible minimal diff containing only the ekstazi specific changes and none others. As we described earlier that xpath library do some optimization while creating internal dom tree structure so we see many other changes in the output but to eliminate them we used diff utils to minimize the diff. In this testcase, we again use the demo folder having pom.xml files, and run our parser to create minimal diff. Then, we check whether the final files produced containing only those extra/changed lines that are related to ekstazi. If we find some other changes we fail the test otherwise the test passes.

We also moved our project to be a part of maven framework because of few reasons:

- Handles the external dependencies well, no need of maintaining extra jar files manually.
- Provides elegant way of pacakging the source code in jar file along with the necessary dependencies.
- Also, all the testcases of project can be run easily using mvn test
- Provided better understanding of integration of maven with java projects and how to use plugins and build in pom files.