File Test Rule

Terry Brady edited this page on Aug 18, 2014 · 21 revisions

The File Analyzer Tool walks a directory tree and performs a "File Test" on each file that is encountered. The application framework allows new File Tests to be quickly developed and deployed into the application. The results of each File Test are compiled into a table that summarizes the results of the analysis.

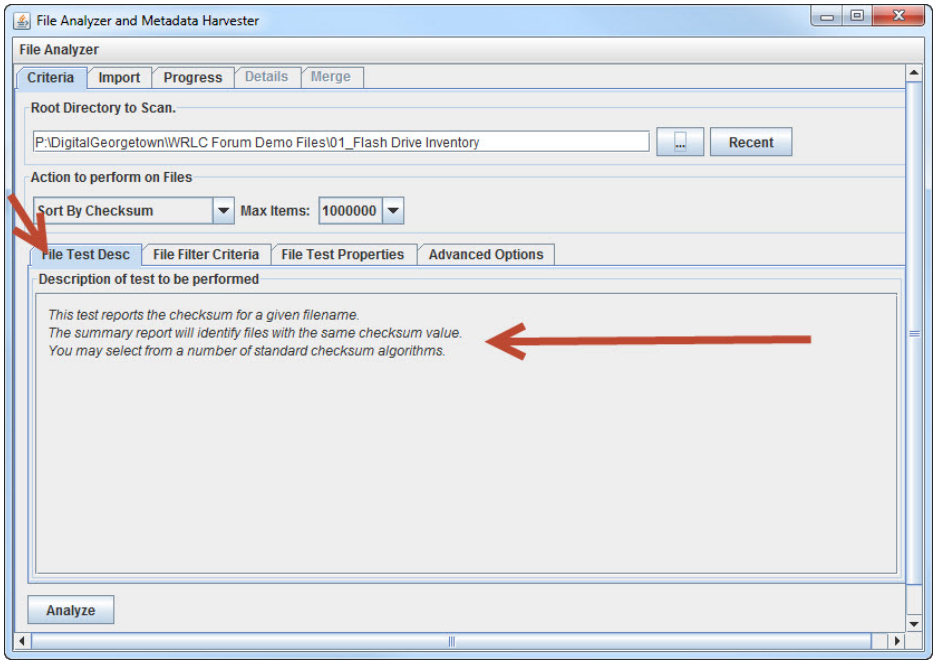
A File Test is a simple set of actions that are performed upon a single file such as filename validation, file size statistical analysis, checksum calculation, file type extraction. Depending on the action, the content of the file may or may not be read. Each File Test is configured with filters that determine which files will be processed by the File Test (i.e. only image files).

Each File Test will generate a table of results. The number of columns and the definition of the columns will vary from test to test. For example, a file type analysis will report the file extension and the number of files discovered with that extension. The checksum file tests will report the name of a file and the checksum string associated with that file.

The File Analyzer tool can be run as a GUI in which the results are displayed in a table. The File Analyzer can also be run in batch mode. In batch mode, the results will be written to a tab-separated file. The GUI version of the application allows the results of multiple executions to be merged. The merged information can be filtered to display matching values and mismatched values.

Components of a File Test

Name and description: explains the File Test to a user

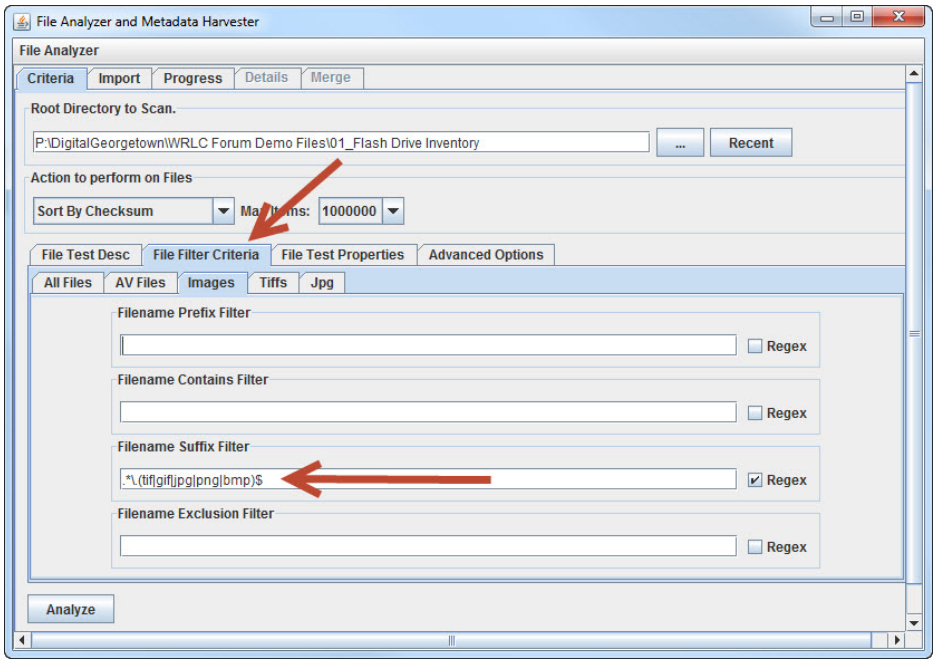


```
public class NameChecksum extends DefaultFileTest {

public String toString() {
    return "Sort By Checksum";
}
public String getShortName(){return "Checksum";}

public String getDescription() {
    return "This test reports the checksum for a given filename.\n" +
        "The summary report will identify files with the same checksum value.\n" +
        "You may select from a number of standard checksum algorithms.";
    }
}
```

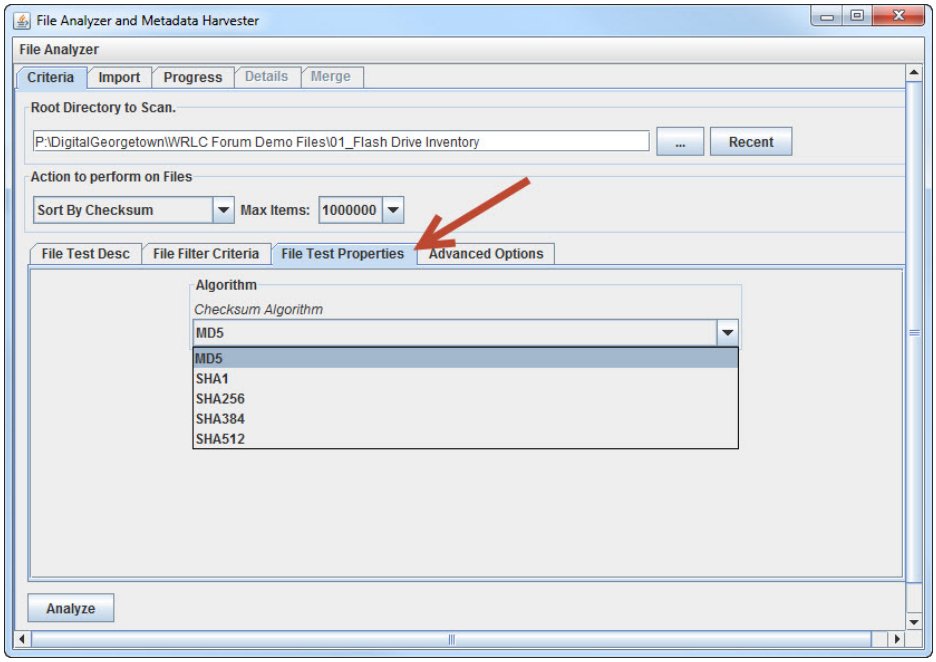
Filters: determine the files that the test will operate upon




```
public void initFilters() {
    initAllFilters();
}

/* from DefaultFileTest.java*/
public void initAllFilters() {
    filters.add(new DefaultFileTestFilter());
    filters.add(new AVFileTestFilter());
    filters.add(new ImageFileTestFilter());
    filters.add(new TiffFileTestFilter());
    filters.add(new JpegFileTestFilter());
}
```

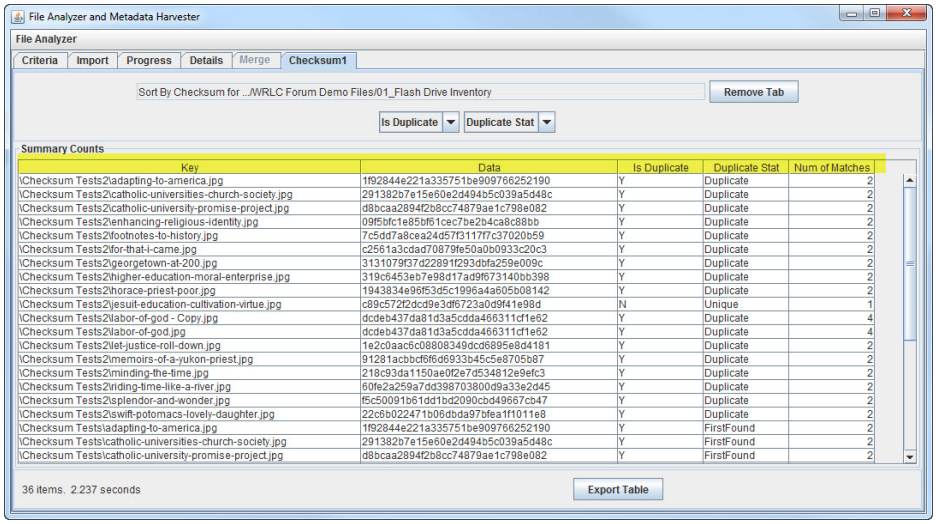
Properties: runtime parameters that the user can pass to the File Test




```
public static final String ALGORITHM = "Algorithm";
static enum Algorithm {
    MD5("MD5"),
    SHA1("SHA-1"),
    SHA256("SHA-256"),
    SHA384("SHA-384"),
    SHA512("SHA-512");
    String algorithm;
    Algorithm(String s) {algorithm = s;}
    MessageDigest getInstance() throws NoSuchAlgorithmException {
        return MessageDigest.getInstance(algorithm);
    }
}

public NameChecksum(FTDriver dt) {
    super(dt);
    keymap = new HashMap<String, List<ChecksumStats>>();
    this.ftprops.add(new FTPropEnum(dt, this.getClass().getName(), ALGORITHM, "algor
            "Checksum Algorithm", Algorithm.values(), Algorithm.MD5));
}
```

Result Stats: defines the resulting information that will be displayed to the user (as a table)



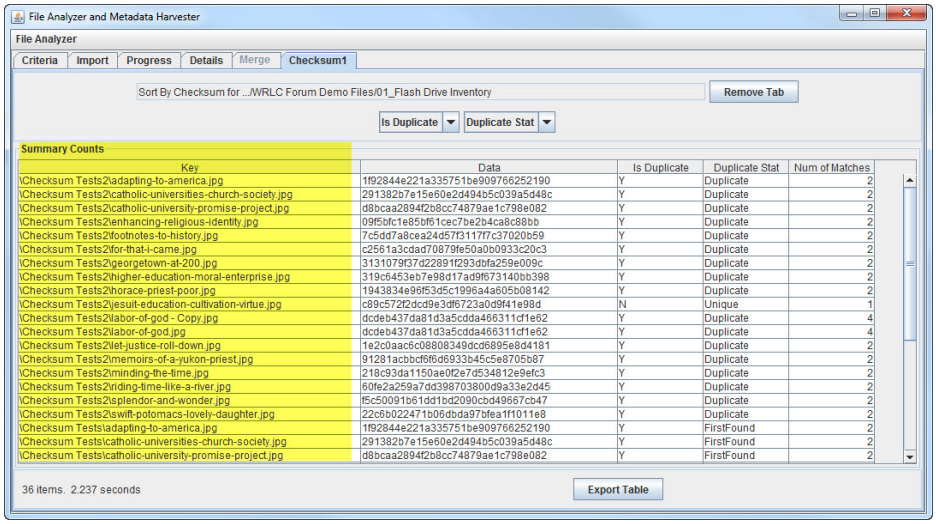

```
public Stats createStats(String key){
    return ChecksumStats.Generator.INSTANCE.create(key);
}
public StatsItemConfig getStatsDetails() {
    return ChecksumStats.details;

}

/*from ChecksumStats.java*/
public class ChecksumStats extends Stats {
    public static enum DUP {Unique, FirstFound, Duplicate;}
    public static enum ChecksumStatsItems implements StatsItemEnum {
        Key(StatsItem.makeStringStatsItem("Key", 400)),
        Data(StatsItem.makeStatsItem(Object.class, "Data", 300).setInitVal("")),
        IsDuplicate(StatsItem.makeEnumStatsItem(YN.class, "Is Duplicate").setInitVal
        DuplicateStat(StatsItem.makeEnumStatsItem(DUP.class, "Duplicate Stat").setIn
        MatchCount(StatsItem.makeIntStatsItem("Num of Matches").setInitVal(1));

        StatsItem si;
        ChecksumStatsItems(StatsItem si) {this.si=si;}
        public StatsItem si() {return si;}
    }
    public static enum Generator implements StatsGenerator {
        INSTANCE;
        public ChecksumStats create(String key) {return new ChecksumStats(key);}
    }
}
```

Result Key: defines the unique key value that will be saved for each file (or set of files) that is processed




```
public String getKey(File f) {
    return getRelPath(f);
}

/*from DefaultFileTest.java*/
public String getRelPath(File f) {
    return f.getAbsolutePath().substring(getRoot().getAbsolutePath().length());
}
```

Code the FileTest

In the example displayed above, a checksum is generated on the file using the algorithm provided by the user.

```
public String getChecksum(File f) {
    Algorithm algorithm = (Algorithm)getProperty(ALGORITHM);
    FileInputStream fis = null;
    try {
        MessageDigest md = algorithm.getInstance();
        fis = new FileInputStream(f);
        byte[] dataBytes = new byte[1204];
        int nread = 0;
        while((nread = fis.read(dataBytes)) != -1){
            md.update(dataBytes, 0, nread);
        }
        byte[] mdbytes = md.digest();
        StringBuffer sb = new StringBuffer();
        for(int i=0; i<mdbytes.length; i++){
            sb.append(Integer.toString((mdbytes[i] & 0xFF) + 0x100, 16).substring(1)
        }
        return sb.toString();
    } catch (NoSuchAlgorithmException e) {
        e.printStackTrace();
    } catch (FileNotFoundException e) {
        e.printStackTrace();
    } catch (IOException e) {
        e.printStackTrace();
    } finally {
        if (fis!=null)
            try {
                fis.close();
            } catch (IOException e) {
                e.printStackTrace();
            }
    }
```

```
    return null;
}

public Object fileTest(File f) {
    return getChecksum(f);
}
```

Indicate how to handle directories and files (beyond the filter settings)

```
public boolean isTestable(File f) {
    return true;
}

public boolean isTestDirectory() {
    return false;
}

public boolean processRoot() {
    return false;
}

public boolean isTestFiles() {
    return true;
}
```

Provide an initial task and a summary task (if needed)

```
@Override public void init() {
    keymap.clear();
}

@Override public void refineResults() {
    for(List<ChecksumStats> matches: keymap.values()) {
        if (matches.size() == 1) continue;
        int count = 0;
        for(ChecksumStats match: matches) {
            match.setVal(ChecksumStatsItems.IsDuplicate, YN.Y);
            if (count == 0) {
                match.setVal(ChecksumStatsItems.DuplicateStat, ChecksumStats.DUP.Fir
            } else {
                match.setVal(ChecksumStatsItems.DuplicateStat, ChecksumStats.DUP.Dup
            }
            count++;
            match.setVal(ChecksumStatsItems.MatchCount, matches.size());
        }
    }
}
```

Register the FileTest with the File Analyzer

```
public class ActionRegistry extends Vector<FileTest> {

private static final long serialVersionUID = 1L;
boolean modifyAllowed = true;

public ActionRegistry(FTDriver dt, boolean modifyAllowed) {
    this.modifyAllowed = modifyAllowed;
```

```
...
add(new NameChecksum(dt));
```

+ Add a custom footer