DTF
Development Guide

Table of Contents

Introduction

This document will cover the details of how to develop within the DTF framework, including the steps to create your own plug and extend DTF's functionality so it can meet your needs. The plug-in describe in later sections and all of the code developed within this guide can be found under the ***dtf/doc/foo-dtf.tgz*** (uncompress that into your Eclipse workspace and you can see the actual changes and the end result to using this document).

XML to Java Relation

Currently each tag in XML land has a corresponding class in Java land and the relation between them is quite visibly obvious. Lets take the simple Echo tag and analyze it better:

XSD definition:

```
<xs:element name="echo">
    <xs:complexType mixed="true">
        <xs:attribute name="message" type="xs:string" use="optional" />
    </xs:complexType>
</xs:element>
```

Java (omitted imports/package declaration section):

```
public class Echo extends CDATA {

    private String message = null;

    public Echo() { }

    public void execute() throws DTFException {
        String msg = getMessage();
        String cdata = getCDATA();
        if (msg != null)
            getLogger().info(msg);
        else if (cdata != null)
            getLogger().info(cdata);
        else
            throw new DTFException("Echo does not contain a message to be printed.");
    }

    public String getMessage() throws ParseException { return replaceProperties(message); }
    public void setMessage(String message) { this.message = message; }
}
```

XML Example:

```
<echo>${prop1} : ${prop2} : ${prop3}</echo>
<echo message="Echoing some System properties"/>
<echo>Arch: ${os.arch}</echo>
```

As we can see above the Java code needs to extend the Action class and then implement the execute method and the properties from the XSD definition of the Echo tag have to match up with the Java in order for the properties to make it from the XML to the Java Object during execution. In this specific example the Echo tag is extending from the CDATA class which already handles the case of child text nodes and groups them to be easily retrieved using the *getCDATA* method.

Somethings to notice is that whenever you have a property that exists in the xml and also needs to be handled in the Java code you need to define the *public* methods for getting/setting that value following the usual Java convention of naming the getter *getMyProperty* and setter *setMyProperty*. Another thing to remember is that if your property can contain properties in the XML representation (properties defined as *${my.property}*) then you have to use the utility function *replaceProperties(String property)* to resolve the property value before making use of it within your own code.

Tag Documentation

Tag documentation is generated from the code itself just as JavaDoc generates Java Api documentation from the @dtf.xxx tags within the code. Building the documentation from the source is done by running the ant doc (in the dtf directory). The other documentation about the framework architecture can be found in the ODF and PDF documents in the doc folder.

Now we'll show some how the Java Code itself can be documented so the framework can generate some nice DTF XML documentation directly from the code. Here's an example of the java code for the Echo tag without any documentation at all:

```
package com.yahoo.dtf.actions.basic;

import com.yahoo.dtf.actions.util.CDATA;
import com.yahoo.dtf.exception.DTFException;
import com.yahoo.dtf.exception.ParseException;

public class Echo extends CDATA {

    private String message = null;

    public Echo() { }

    public void execute() throws DTFException {
        String msg = getMessage();
        String cdata = getCDATA();
        if (msg != null)
            getLogger().info(msg);
        else if (cdata != null)
            getLogger().info(cdata);
        else
            throw new DTFException("Echo does not contain a message to be printed.");
    }

    public String getMessage() throws ParseException { return replaceProperties(message); }
    public void setMessage(String message) { this.message = message; }
}
```

The above class won't have any XML documentation generated when you run the *ant doc* task is executed. There are quite a few JavaDoc tags that can be used to describe the Action as well as each of the individual attributes. The best way to see these is to look at the previous example with the documentation in place and explain each of them individual tags, the comments will be inline in a different color below:

```
package com.yahoo.dtf.actions.basic;

import com.yahoo.dtf.actions.util.CDATA;
import com.yahoo.dtf.exception.DTFException;
import com.yahoo.dtf.exception.ParseException;

/**
 * @dtf.tag echo                                          → name of the XML tag being documented
 *
 * @dtf.since 1.0                                         → version where this appeared
 * @dtf.author Rodney Gomes                               → original author of this tag
 *
 * @dtf.tag.desc The echo tag is used to print out testcase information during
 *               the execution of the testcase.           → small description of the tag
 *
 * @dtf.tag.example                      → Used to show as many example usages as you'd like
 * <local>
 *     <echo message="Hello World"/>
 * </local>
 *
 * @dtf.tag.example
 * <local>
 *     <echo>Hello World</echo>
 * </local>
 */
public class Echo extends CDATA {

    /**
     * @dtf.attr message        → For each attribute you should describe what values it accepts.
     * @dtf.attr.desc The message attribute contains the message to be logged to
     *                the test execution log.
     */
    private String message = null;

    public Echo() { }

    public void execute() throws DTFException {
        String msg = getMessage();
        String cdata = getCDATA();
        if (msg != null)
            getLogger().info(msg);
        else if (cdata != null)
            getLogger().info(cdata);
        else
            throw new DTFException("Echo does not contain a message to be printed.");
    }

    public String getMessage() throws ParseException { return replaceProperties(message); }
    public void setMessage(String message) { this.message = message; }
}
```

Now the above is a simple example and there are some more advanced uses, such as being able to write HTML directly in the middle of those descriptions and then you can define tables and lists right there in your description nicely. The best thing to do at this stage is to have a look at the other actions that are documented and experiment a bit with what you can and can't do.

Creating a Plugin for DTF

Why would I do this ?

The main reason to create a plug-in for DTF is because you're going to use functionality that DTF doesn't currently have. To add this functionality without destabilizing the main framework you can create a plug-in with the new functionality (be it new tags or extensions to some of the features within DTF) and at a later time if your functionality is required by other teams it can be pulled up the stack to the main DTF framework.

How do I start ?

Before you start make sure to checkout the latest **DTF** code (follow the instructions in the DTF User's Guide on how to do this with Eclipse) and inside of the *src/plugin-dtf* you'll find a full functional plug-in that has an example Tag. Also note that there is a *pluginc-dtf* and *plugin-dtf*, the *pluginc* includes an example of how to create an external **CLI** to talk to another **API** that is not **Java**, we'll only cover working with the *plugin-dtf* since its written all in **Java** and easier to understand the basic concepts with at first.

Steps to make your own plugin for dtf:

1. Copy the directory *dtf/src/plugin-dtf* to the same level as your dtf source tree resides so that both your *plugin-dtf* directory and *dtf* directory are at the same level.

2. Rename the plugin word in *plugin-dtf* so that it has something to do with your own product. (for these steps I'll name it *foo-dtf*)

3. Now if you're using Eclipse:
   - Then go into Eclipse and right click on your **Package Explorer** perspective on the left side (just click on any white space) and make sure to select **New → Java Project** and type foo-dtf into the project name (this will automatically detect you have this directory already in your workspace). Click on **Finish** and the foo-dtf project should appear in your Eclipse workspace ready to be built.

4. Now either through Eclipse or through the command line open up both the *build.xml* and *build.properties* within your *foo-dtf* directory and edit those so that all properties start with the prefix *foo* (your product name) and not *plugin*. You don't really have to do this but it makes things easier to read. In the build.properties file make sure that the property **dtf.plugins** points to the right location, in this case *../foo-dtf*. This directory is in relation to where the *dtf* directory is. Also make sure that the **dtf.dir** also points to where the *dtf* checkout is in relation to the *foo-dtf* directory, in this case we just point it to *../dtf* .

5. Building without Eclipse:
   Go to the same location the *build.xml* is and issue the following *"ant clean ; ant build"* and you should see:

```
build:
     [echo] DTF build at /home/neo/yahoo/workspace/dtf/build/dtf/dist

BUILD SUCCESSFUL
Total time: 15 seconds
```

6. Build with Eclipse:

   Make sure you've highlighted the *build.xml* file within your *foo-dtf* project in Eclipse and go to the top of eclipse where there is a little green arrow with a small red suitcase next to it and click on the small arrow pointing down (alternatively go to **Run → External Tools Configuration**) Then click on the **Ant Build** on the left pane, followed by clicking on the little sheet icon with a plus sign on it (tool tip says ***New launch configuration***) this will create a new entry named *"foo-dtf build.xml"*. You can rename that on the right side now as well as picking which Targets to run from the ant file. There's a **Targets** tab that you can go to and you should click on the clean and build targets (be sure to have the order that they'll be executed is correct). You should also go to the **Refresh** tab and make sure it has the option selected to refresh your workspace at the end of each build. Once you've done the above, you can hit the **Run** button at the bottom right hand corner and you should see the build being executed as it would from the command line but within eclipse with some syntax coloring of the logs.

In this example plugin-dtf I've created a new tag:

- plugin_action
  - this action just shows how to simply create an action that has a single attribute and an empty execution method. The meat of your action would go into the execute method.

The pluginc-dtf has the 2 additional tags:

- test
  - Test is a more advanced example of how to use an external CLI and this falls into the realm of testing that is done in another language other than Java. This small example shows how to keep a CLI Pool and use that to execute actions in a CLI which has been created. This external CLI exists in the src/c directory and has an associated Makefile to build for this part of the example.

- get_test_stats
  - Because the test action has an external CLI implementation, that CLI is responsible for outputting its events to an external file that can later be collected using the get_test_stats tag. The reasoning for this is that we don't want the external CLI to really have to return every event back into the Java land process and we can really execute tests at the speed of the language being tested with zero overhead from the testing framework.

How do I add my own external libs or scripts to the DTF Build ?

This is quite easy to do, when your plugin is being built you are placing your build under the directory **dtf/build/dtf/plugin-dtf/dist** directory and there are already 4 folders you'll find there: bin, lib, src and tests. The src folder is only used to generate the JavaDoc from while the other three are copied as is into the **dtf/build/dtf/dist** directory. The bin contents are copied to the **dtf/build/dtf/dist** while the lib directory's contents are copied into **dtf/build/dtf/dist/lib** and the tests directory is renamed and placed in the **dtf/build/dtf/tests/plugin-dtf** directory (to separate testware per plugin).

The build.xml file that is supplied already copies the tests, source and libs). If you want o add some new java libraries just add them to the lib folder in your plugin and then refresh your eclipse workspace and right click on each of them and do Build Path → Add To Build Path to be able to reference those classes in your eclipse code.

As for your build.xml you don't need to add anything unless you want to copy libs from a different source then you should add that copy call in your compile target right after the first mkdir which is creating your plugin output directory, otherwise your libraries won't be included in time for compilation. Now you'll be set to add any other future libs with out having to make any changes to your build script. You'll only have to make sure to add your new jar file to the build path in eclipse to be able to write the code that references those libraries

What can I do now that I have my plug-in ?

Within your new tags/actions you can now use utilities available to all actions for logging, event recording, result reporting, etc. To see examples of these just have a look at DTF's actions under ***dtf/src/com/yahoo/dtf/actions***. You'll notice that when you want a logger you'll just type within your action *getLogger().info()* or *.warn()*, and when you want to record events you'll create your **Event**, name it and use it to measure a certain activity and at the end you can just issue *getRecorder().recorder(event)* and you're done. The best thing to do is really look at already implemented tags under ***dtf/src/com/yahoo/dtf/actions/http*** and ***dtf/src/com/yahoo/dtf/actions***. For more information on adding your own tag see the example in the following section: **Adding a New Action.**

How can I interact with C code ?

There is always more than a single way to interact with another language but currently within **DTF** there are two preferred ways (**JNI** won't receive any attention because its just a hideous piece of code to maintain and should only be done as a last resort or by those who are very familiar with using **JNI**):

- **CLI (Command Line Interface)**
  - By creating an external process that can respond to command line instructions it is pretty easy to create a process pool within **DTF's** Java code and manage which **CLI**'s are in use and issue new commands to this **CLI** to have that **CLI** translate into the underlying language execution. This method should also rely on the **CLI** to create the event outputs in the same format as **DTF** does right now. An example of a **CLI** exists in the **dtf/src/pluginc-dtf** example.
- **JNA (Java Native Access)**
  - **JNA** provides Java programs easy access to native shared libraries without writing anything but Java code. This is a lot easier than **JNI** and if you just need to call these libraries to manipulate things in the underlying system then this is by far the best option. An example of using **JNA** can be found in the **dtf/src/plugin-dtf** plugin and shows how easy it is to just call **C** level functions such as *printf*.

CLI vs JNA

For each of the previously mentioned methods there are always pros and cons which we should analyze closely before deciding on which approach servers better the testing purpose of the task at hand. Now here's a list that tries to identify the main features to be compared for this comparison to be fair.

Maintenance:

CLI

The **CLI** involves having to maintain an external piece of **C** code which has to adapt to changes in the library being tested. This code must of course be compiled and shipped as a binary with the **DTF** plugin or readily available on the systems being tested.

JNA

You must adapt to library changes but you don't have to compile any **C** code and instead have to make any necessary changes to your Java code where you already are use to writing code.

Debugging:

CLI

Debugging issues in the **CLI** can be easier than **JNA** because you're able to execute the **CLI** on its own and debug it with appropriate **C** debugging tools.

JNA

You'll have to be quite good at identifying who's at fault when the **JVM** dies with a segmentation fault because there are 3 possible culprits: **JVM**, **C** function being called or the way the **Java** land code is passing arguments to this function. This is exactly the same issue that **JNI** has, where it can be complicated to sometimes to figure out exactly where the code is failing, specially if you're not familiar with debugging **C** code.

Ease of writing:

CLI

Writing new **CLI** code or additions isn't hard if you're use to writing the secondary language. So this really depends on your familiarity with the **CLI** language that you'd have to maintain.

JNA

JNA couldn't be easier to write, when compared to **JNI** its incredibly simple to write and even easier to read and understand. Its also completely portable between *nix systems and Windows.

Performance:

CLI

CLI can be perform extremely well because the secondary process is left running and al you do is pass in more commands to execute in the secondary language. This is probably the fastest way of talking between Java and another language.

JNA

JNA is definitely not as fast as a **CLI** when it comes to executing multiple different **C** functions and it is slower than **JNI** since it has to do dynamic type binding at runtime.

With some small tweaks you can be close to **JNI** performance but it definitely wouldn't be able to do hundreds of thousands of operations per second.

Portability:

CLI

The **CLI** is as portable as you'll write it but that also means you'll have to maintain different builds in order to use the **CLI** in different environments.

JNA

JNA is very portable and runs on both *nix systems and Windows making it very useful for cases where you may need to test on other environments.

Encoding and Bindings:

CLI

Managing string encodings and data type bindings in the **CLI** has to be done carefully and can result into complicated **C** code to figure out what is being passed in by the Java land process. There's also the issue of Encoding getting mangled on the command line if the system you're running in happens to have its Locale set to something that you didn't expect.

JNA

JNA handles all of the encoding issues and data type bindings and mappings are well explained so that you can easily write up the method signatures in Java to match your C/C++ land counter parts.

So now that we've covered all of those areas you can probably see better which fits the necessities of your task and should be able to pick which one fits your needs better. There is a following section that goes into explaining how to write up the **CLI** and **JNA** implementations for the **C** function *gettimeofday* and you can also see the benefit of using each of the approaches.

Accessing gettimeofday with a CLI

So we'll start by using the existing **CLI** and adding a new command that can be executed named *gettimeofday* and that command will execute the *gettimeofday* **C** function and will print to the **STDOUT** an event that can be easily read back by the **Java** land code to return the same event back into **DTF** Testcase. The small changes to the CLI code are the following (All code referenced in this section can be found under **dtf/src/pluginc-dtf**):

```
} else if ( strcmp(command,"gettimeofday" ) == 0) {
       struct timeval tv;
       op.name = GETTIMEOFDAY_EVENT;
       op.start = getMilliseconds();
       op.succeeded = (gettimeofday(&tv,NULL) == 0);
       op.stop = getMilliseconds();
       op.result = tv.tv_sec;
       printOp(op);
}
```

This new case to be handled by the **CLI** will just execute that command and then create the appropriate operation result and pass the number of seconds that were returned by the *gettimeofday* function in that **Event**. This implementation is as simple as possible and will be used by the cli_gettimeofday tag to get back to the DTF code the time of day in seconds. There are quite a few things that need to be developed in order to use the CLI's in your Java code in a thread safe way with the ability to return events back to Java land, these include:

- The **CLI** has 2 new commands to turn on and off the returning of event information over the **STDOUT**.

- Each **PluginCLI** instance needs to keep track of if its in the return events mode or not and this code can get quite tricky and complex (just have a look at the *pushCommand* method in the **PluginCLI** code).

- The **CLIPool** needs to keep track of all of the **CLI's** that are created and pool them when possible to reuse the same **CLI** resource.

When completed all of these changes we have the ability to use the *gettimeofday* function right from within **DTF** with a simple **cli_gettimeofday** tag that returns an event named ***cli_gettimeofday*** with the attribute result that contains the number of seconds since Epoch. The complete solution depends on quite a few pieces to work correctly including:

- `PluginCLi.java`
- `PluginCLIPool.java`
- `plugin_cli.c`
- `Cli_gettimeofday.java`

As you can see there is quite a lot of development involved to get things to work nicely between Java land and C land code.

Accessing gettimeofday with JNA

In **JNA** things are quite simple because **JNA** has taken the work out of writing any **C** code out of our hands and has given us only two tasks. The first is to define the interfaces that match what the external **C** function looks like and the second to define as well the structures that this interface uses. Then using **JNA** to load the external library that matches the previously defined interfaces we can use those **C** functions as if they were in Java land. Here is what we need to write to have access to the *gettimeofday* function through **JNA**:

```
public interface CLib extends Library {
    CLib INSTANCE = (CLib) Native.
           loadLibrary((Platform.isWindows() ? "msvcrt" : "c"), CLib.class);

    // int gettimeofday(struct timeval *tv, struct timezone *tz);
    int gettimeofday(TIMEVAL tv, TIMEZONE tz);
}

/*
 *  struct timeval {
 *      time_t      tv_sec;
 *      suseconds_t tv_usec;
 *  };
 */
public static class TIMEVAL extends Structure {
    public int tv_sec;
    public int tv_usec;
}

/*
 *  struct timezone {
 *      int tz_minuteswest;
 *      int tz_dsttime;
 *  };
 */
public static class TIMEZONE extends Structure {
    public int tz_minuteswest;
    public int tz_dsttime;
}
```

The above code makes the *gettimeofday* function readily available to anyone who has access to the **CLib.INSTANCE** object. The structure required to talk to *gettimeofday* have also been defined in Java to easily map between **C** land and **Java** land. Using this function is as easy as doing the following:

```
TIMEVAL tv = new TIMEVAL();
int rc = CLib.INSTANCE.gettimeofday(tv, null);
```

That is much simpler than **JNI** and when compared to the **CLI** a lot easier to maintain because there is literally 1 class that was written to have this method exported in Java which happens to be the same class. Look at **dtf/src/plugin-dtf/src/java/com/yahoo/dtf/actions/plugin/Jna_gettimeofday.java** you will see that all of the **DTF** code and **JNA** code adds up to a mere 60 lines of code.

Of course the **JNA** library has one simple *jna.jar* file that needs to be in your **plugin-dtf/lib** directory but that is included with the plugin-dtf. For more information on JNA and to find documentation on converting data types please refer to https://jna.dev.java.net/

CLI vs JNA real world analysis

The previous sections covered two ways of getting the same information from a **C** code, one using the **JNA** approach and the other using the creation of a **CLI**. They are both quite different and offer different benefits now here we'll analyze exactly how these two compare in terms of lines of code to write, performance of the final tag, etc. Here is a table with the various quantitative measurements that can be compared between these two approaches:

Area of Analysis	**JNA**	**CLI**
Lines of code	60	> 200 *
Performance (executions/sec)**	14,000	22,000***

* Adding all of the lines of code of the tag code + **CLI** pooling code and *plugin_cli.c* is around 300, but with a few lines dedicated to a different tag (maybe some 50 lines of code). So I'd just rather say that its over 200 to have a similar solution to the **JNA** case.

** This test is done with a single thread executing the specific tag implemented in a loop 100,000 times in a row and then calculating what the average executions per second was.

*** This number is actually higher for a simple CLI call such as the one done by the cli_test tag which just calls test CLI command with two arguments. In this scenario you can see a performance of 80,000 executions per second, this of course when you're not getting back the Event data at runtime and instead logging to the external event file that you later gather for analysis (see the **dtf/src/pluginc-dtf/tests/cli_test.xml** for an example).

It is important to understand that these numbers aren't set in stone and that even on the **JNA** page there is information on how to get better performance by using certain data types. So when writing your own **JNA** implementation be aware that there are better ways of doing the same thing.

Now looking at these numbers its pretty interesting to see that **JNA** is only 40% slower than a **CLI** approach, specially since the solution itself is a lot easier to write. Another thing to note is that this is for a single thread looping as fast as it can and that you could easily scale your test with a few more threads and not worry about the fact that the **JNA** solution has a small overhead on each call being executed.

What else can I extend inside DTF ?

Event Recorder

There are quite a few things that can be extended within DTF, lets start with the ability to change the way you record your events. DTF currently only supports recording to a TXT format which just prints each event as a continuous block of keys=value for each event and then separates the events by two new lines. Now lets say we wanted to create a new Recorder that allows us to output our events into a XML format instead. First thing to do is to extend from the class **RecorderBase**, when you do this you'll have to create the methods record, stop and start. You also needs to make sure that you have a constructor that accepts the correct arguments URI uri, boolean append and String encoding. So here's how the class will look initially:

```
package com.yahoo.dtf;

import java.net.URI;

import com.yahoo.dtf.exception.RecorderException;
import com.yahoo.dtf.recorder.Event;
import com.yahoo.dtf.recorder.RecorderBase;

public class XMLRecorder extends RecorderBase {

    public XMLRecorder(URI uri, boolean append, String encoding) {
        super(uri, append, encoding);

    }

    @Override
    public void record(Event counter) throws RecorderException {

    }

    @Override
    public void start() throws RecorderException {

    }

    @Override
    public void stop() throws RecorderException {

    }
}
```

Now we're basically ready to really give this class functionality. So in the constructor we'll need to figure out where the output is, we have a URI and what we should use is the built-in StorageFactory to get the location of this URI that was passed as an argument. So if you look at another Recorder such as TextRecorder under src/java/com/yahoo/dtf/recorders you'll see that this is how we use the StorageFactory to get our output stream:

```
BufferedOutputStream _os = null;

public XMLRecorder(URI uri, boolean append, String encoding) throws StorageException {
    super(uri, append, encoding);
    StorageFactory sf = Action.getStorageFactory();
    _os = new BufferedOutputStream(sf.getOutputStream(uri,append));
}
```

Now why do we use this **StorageFactory** and not just open up a file on the filesystem ? Well the **StorageFactory** will allow us to support having data on a remote server somewhere and be able to add that functionality into the test without having to make changes to those who use the **StorageFactory** because the Input/OutputStreams being returned can easily be to a socket or a file on the filesystem.

Now lets get into the rest of the code so the real important method is the record() method which gets each and every event that the system is throwing. When you're writing this you have to understand that the record method right now can have an impact on the amount of work the system can do. We may look into making this activity asynchronous with the work being do so that the **Recorder** would not affect performance in the future but for the time being this approach is a lot simpler and in most cases you're able to recorder in the order of 30K plus events per second per thread which is a lot more than most testing actions can do. Now back to our method we want to write out a valid XML document containing the Events that were throw in in the framework, we'll need to make sure we have a root XML node, lets call the <events> and then we need to know what type of information does the Event carry. So an Event has a name, start time, stop time and a list of key, value pairs that are referred to as attributes. With this in mind our XML output could like like so:

```xml
<events>
    <event name="xxx" start="xxx" stop="xxx">
        <attribute1>value1</attribute1>
        <attribute2>value2</attribute2>
    </event>
</events>
```

So now here's what the class may look like:

```java
package com.yahoo.dtf;

import java.io.BufferedOutputStream;
import java.io.IOException;
import java.net.URI;
import java.util.ArrayList;

import com.yahoo.dtf.actions.Action;
import com.yahoo.dtf.actions.event.Attribute;
import com.yahoo.dtf.exception.ParseException;
import com.yahoo.dtf.exception.RecorderException;
import com.yahoo.dtf.exception.StorageException;
import com.yahoo.dtf.recorder.Event;
import com.yahoo.dtf.recorder.RecorderBase;
import com.yahoo.dtf.storage.StorageFactory;

public class XMLRecorder extends RecorderBase {

    BufferedOutputStream _os = null;

    public XMLRecorder(URI uri, boolean append, String encoding) throws StorageException {
        super(uri, append, encoding);
        StorageFactory sf = Action.getStorageFactory();
        _os = new BufferedOutputStream(sf.getOutputStream(uri,append));
    }

    @Override
    public void record(Event event) throws RecorderException {
        StringBuffer line = new StringBuffer();
        line.append("    <event name=\"");
        line.append(event.getName());
        line.append("\" start=\"");
        line.append(event.getStart());
        line.append("\" stop=\"");
        line.append(event.getStop());
```

```
            line.append("\">\n");
            try {
                _os.write(line.toString().getBytes());

                ArrayList<Attribute> attributes =event.findActions(Attribute.class);
                for (int i = 0; i < attributes.size(); i++) {
                       line = new StringBuffer();
                       Attribute attribute = attributes.get(i);
                       line.append("        <" + attribute.getName() + ">");
                       line.append(attribute.getValue());
                       line.append("</" + attribute.getName() + ">\n");
                       _os.write(line.toString().getBytes());
                }

                _os.write("    </event>\n".getBytes());
            } catch (IOException e) {
                throw new RecorderException("Error writing to XML recorder.",e);
            } catch (ParseException e) {
                throw new RecorderException("Error handling event attributes.",e);
            }

        }

        @Override
        public void start() throws RecorderException {
            try {
                _os.write("<events>\n".getBytes());
            } catch (IOException e) {
                throw new RecorderException("Unable to write to file.",e);
            }
        }

        @Override
        public void stop() throws RecorderException {
            try {
                _os.write("</events>".getBytes());
            } catch (IOException e) {
                throw new RecorderException("Unable to write to file.",e);
            }

            try {
                _os.close();
            } catch (IOException e) {
                throw new RecorderException("Unable to close file.",e);
            }
        }
    }
```

You'll notice we're not handling the value of the attribute and making sure to encode that for XML because it could contain special characters such as < or >. This is for the walk through purpose only because the *XMLRecorder* isn't really useful in the main DTF framework for the time being. Now that we have our recorder all we have to do is in our InitHook class in the plugin itself we need to register this recorder by calling the following API:

```
        RecorderFactory.registerRecorder("xml", XMLRecorder.class);
```

So that's it our new recorder is available with the name “xml” set in the attribute type of the record tag we are able to use our new record tag, like so (test in the *dtf/doc/foo-dtf/tests* folder):

```
    <script xmlns="http://dtf.org/v1" name="xmlrecorder">
        <info>
            <author>
                <name>Rodney Gomes</name>
                <email>rlgomes@yahoo-inc.com</email>
            </author>
            <description>test out the new xml recorder</description>
        </info>
        <local>
            <createstorage id="INPUT" path="${dtf.xml.path}/input"/>
            <createstorage id="OUTPUT" path="${dtf.xml.path}/output"/>
        </local>

        <record type="xml" uri="storage://OUTPUT/events.xml">
            <event name="event1"/>

            <event name="event2">
                <attribute name="flag1" value="true"/>
                <sleep time="1s"/>
            </event>

            <event name="event3">
                <attribute name="current_time" value="${dtf.timestamp}"/>
                <attribute name="random_int" value="${dtf.randomInt}"/>
                <sleep time="1s"/>
            </event>
        </record>
    </script>
```

The output from running this tests is the following XML:

```
    <events>
        <event name="event1" start="1238087882045" stop="1238087882045">
        </event>
        <event name="event2" start="1238087882046" stop="1238087883047">
            <flag1>true</flag1>
        </event>
        <event name="event3" start="1238087883048" stop="1238087884049">
            <current_time>1238087883048</current_time>
            <random_int>-2116988479</random_int>
        </event>
    </events>
```

Event Query

As you can see it is quite easy to get a new Recorder into place, but now you may be thinking what about the Query tag that does the opposite ? You have the exact same steps to do just a different class to extend from which is called QueryIntf and then calling the: `QueryFactory.registerQuery("txt", TxtQuery.class);` Will register this class for use in the Query tag. I won't go into that one because it is very similar to the previously done Recorder work.

Property Transformers

Make sure to read the Property Transformation section in the DTF User's Guide first before reading this section to understand what Transformers are used for. Now there are some built-in Transformers in DTF that allow you to easily transform a property by applying an XPath expression or by manipulating the value of a property URL Encoding or Decoding it. But what if you need a transformer that allows you to transform your data in a different way ? What if you want to be calculate the length of the property value within the framework at runtime ? Well transformers can be used for both of these tasks since they are applied at runtime and allow you to Transform your property value into something else based on the arguments passed to the Transformer. Lets make a Transformer that will do string related operations such as calculating a strings length.

First thing to start with is create a class that implements the Transformer interface, here's the skeleton of this class:

```
package com.yahoo.dtf.transformers;

import com.yahoo.dtf.config.transform.Transformer;
import com.yahoo.dtf.exception.ParseException;

public class StringTransformer implements Transformer {

    public String apply(String data, String expression) throws ParseException {
        return null;
    }

}
```

That's pretty much it, you need to implement the apply method. The data is the value of the property at runtime and the expression is what is passed to your transformer using the following syntax:

```
${property:transformer_name:expression}
```

So you can pass whatever you'd like in the expression part, be it a list of arguments separated by commas or some other format you decide is more appropriate for your **Transformer**. For our example we'll call the **Transformer** itself *string* and we'll use the expression to pass the name of functions to apply such as *length, substring, reverse,* etc. So lets implement the *length* function and see how easy it is to get our Transformer up and running:

```
package com.yahoo.dtf.transformers;

import com.yahoo.dtf.config.transform.Transformer;
import com.yahoo.dtf.exception.ParseException;

public class StringTransformer implements Transformer {

    public String apply(String data, String expression) throws ParseException {
        if ( expression.equalsIgnoreCase("length") ) {
            return "" + data.length();
        }

        throw new ParseException("Unkown string expression [" + expression + "]");
    }
}
```

That couldn't have been easier and after registering your Transformer like so:

```
TransformerFactory.registerTransformer("foo", new StringTransformer());
```

You can now use your transformer on any property like this: **`${myproperty:foo:length}`**. There is an example test at ***doc/foo-dtf/tests/string.xml*** that will show how this new Transformer is used. The transformer was called foo because the **StringTransformer** was moved to the main DTF branch and is now only in the **foo-dtf** example plugin for demo purposes (under a different name to avoid conflicts).

Results Outputting

Currently DTF supports only outputting results to a predefined XML format, this is just because there hasn't been a need to output to any other format and you could easily apply an XSL stylesheet to convert this same XML to HTML or even another XML format used by some reporting tools of your preference. Now what if you wanted to report your results into some other service and at the end of a test run instead of just outputting the results to a file you'd like to POST the results to a webservice somewhere ? Well this is quite straightforward by extending from the class **ResultsBase** you can now create your own Results class that will receive every single test result thrown during a test run. You can then register this **ResultsBase** class by using the ***ResultsFactory.registerResults()*** method. I'm not going to show any examples since there already is a **ConsoleResutls** and **XMLResults** that show two different approaches and can easily be used as a starting example on how to output testing results to another format/destination.

Dyanmic Properties

Dynamic properties are very useful when you want to access something that changes all the time but don't want to write a tag for it since the property itself can easily be used in any point in the testcase without having to tell it to save the result to another property name. So you can easily reference ***${dtf.timestamp}*** whenever you need to have the ***System.currentMillis()*** value from Java in your test case. In DTF we also use dynamic properties for generating random numbers and also for generating streams (see the Stream Generators section in the User's Guide for more information on that). As an example lets create a simple Dynamic Property that can just give us access to the underlying OS Environment variables. So firstly we need to create a new class that implements the **DynamicProperty** interface. This interface only requests that you create a method ***getValue(String args)*** the String passed as an argument comes from the possible use of ***${your.dynamic.property(XXX)}*** and the format of your arguments is really up to you. Here's what our implementation of our new dynamic property may look like:

```
package com.yahoo.dtf.config;

import com.yahoo.dtf.exception.ParseException;

/**
 * Example dynamic property that can be used to retrieve environment variable
 * state right within your DTF test through a simple property.
 *
 *  ${sys.env(HOME)} - would return the HOME environment variable value.
 *
 */
public class EnvProperty implements DynamicProperty {

    public static String SYS_ENV = "sys.env";

    public String getValue(String args) throws ParseException {
        return System.getenv(args);
    }

}
```

The only thing missing is to register this new **DynamicProperty** in your **InitHook** like so:

```
Config.registerDynamicProperty(EnvProperty.SYS_ENV, new EnvProperty());
```

This is really a simple thing to add and you'll notice that most dynamic properties are quite simple because they just call up some existing functionality that exposes a bit of information in a dynamic way to the test case. Here's what the usage of this new dynamic property would look like:

```
<script name="env_test" xmlns="http://dtf.org/v1">
    <info>
        <author>
            <name>Rodney Gomes</name>
            <email>rlgomes@yahoo-inc.com</email>
        </author>
        <description>
        This test makes use of the sys.env dynamic property to show you how easy
        it is to use this new property once it has been created and register
        in your plug-in's InitHook.
        </description>
    </info>

    <local>
        <createstorage id="OUTPUT" path="${dtf.xml.path}/output"/>
        <createstorage id="INPUT" path="${dtf.xml.path}"/>

        <property name="iterations" value="200000"/>
    </local>

    <log>USER = ${sys.env(USER)}</log>
    <log>HOME = ${sys.env(HOME)}</log>

    <!-- and how fast does this perform ?  -->

    <event name="test">
           <distribute property="i" range="1" iterations="${iterations}">
               <property name="user" value="${sys.env(USER)}" overwrite="true"/>
           </distribute>
    </event>

    <subtract op1="${test.stop}" op2="${test.start}" result="duration_ms"/>
    <divide op1="${duration_ms}" op2="1000" result="duration_s"/>
    <divide op1="${iterations}" op2="${duration_s}" result="ops_per_sec"/>

    <log>Operations/second: ${ops_per_sec}</log>

</script>
```

You'll notice how easy it really is to use the new property just reference it and pass the arguments you need and things get resolved automatically. The other thing we did in this small test is to measure how many dynamic properties resolutions we can do per second. When I ran this exact test I could get 100K per second, which means this should meet the needs of just about anyone using DTF.

Stream Handlers

Stream handlers are what allow your test to send and receive large amounts of data without ever having to contain the complete object in memory. The way this is achieved is by using a property called `${dtf.stream(type,size,arguments)}` and in your code instead of using ***replaceProperties()*** you use the ***replacePropertiesAsStream()*** function which gives you back an **InpuStream** that can be used to then pass to any other APIs you maybe using or write directly to some socket you'd like to handle. If the data being passed in the attribute happens to not be a stream its still handle correctly and converted into an InputStream for you so can just handle that piece of data as a stream no matter what.

Now you'll notice there are only 2 stream handlers right now, there's the random pattern generator and the repeat handler that will just repeat a piece of data N times till it fills your stream up to the size you wanted. So with this you'll probably want to work on creating your own stream handler to generate certain types of data that follow patterns you need for testing the product at hand. For our example I'm going to go over generating sentences. We'll start by creating our DTFInputStream and then start adding the logic to take into account a dictionary where we can look up English words and generate sentences. So here's what the class will look like before we've actually given it some “meat”:

```
package com.yahoo.dtf.streaming;
```

```
import java.io.IOException;

import com.yahoo.dtf.exception.ParseException;

public class SentencesInputStream extends DTFInputStream {

    public SentencesInputStream(long size, String pattern) throws ParseException {
        super(size, pattern);
    }

    public int readByte() throws IOException {
        return -1;
    }

    public int readBuffer(byte[] buffer, int offset, int length) {
        return -1;
    }

    public String getAsString() {
        return super.getAsString();
    }
}
```

Now we'll create a dictionary for words to lookup along with some random way of deciding when to place commas, periods, exclamation marks, etc. into our sentences. So we'll create a simple array with some words in it (for this example 850 English words) and we'll just store that statically in our class. We'll create an array with other punctuation symbols that belong in sentences such as commas, periods, etc. Then we'll have to come up with a neat way of picking words and punctuation symbols in a way to construct sentences that almost seem like they're grammatically correct (there are ways of constructing grammatically correct English but I don't want to go into that in this document). Without showing the 850 English words here is what the code could look like:

```
package com.yahoo.dtf.streaming;

import java.io.IOException;
import java.util.Random;

import com.yahoo.dtf.exception.ParseException;

public class SentencesInputStream extends DTFInputStream {

    /*
     * Those are 850 English words copied from somewhere on the web, this is
     * just a simple way of having those words available to make it easy to
     * generate the data we want.
     */
    public static String[] WORDS = { "a", "able", ... };

    private long _seed = 0;

    private Random _random = null;

    private StringBuffer[] _sentences = null;

    public SentencesInputStream(long size, String args)
           throws ParseException {
        super(size, args);

        try {
            _seed = new Long(args);
        } catch (NumberFormatException e ) {
            throw new ParseException("Argument should be a long, not [" + args
                                     + "]");
        }
        _random = new Random(_seed);
        int max = (1 + ((int) size / 1024)) % 32 ;
        _sentences = new StringBuffer[max];
```

```
            for (int s = 0; s < _sentences.length; s++) {
                StringBuffer sentence = new StringBuffer();
                   for (int i = 0; i < 8; i++) {
                       String sep = " ";

                       if ( i % 5 == 0 )
                          sep = ", ";

                       int wordindex = _random.nextInt(WORDS.length);
                       sentence.append(WORDS[wordindex] + sep);
                   }
                   sentence.replace(sentence.length()-1, sentence.length(), ". ");
                   _sentences[s] = sentence;
            }
        }

        private StringBuffer _sentence = null;
        private int _readbytecount = 0;
        public int readByte() throws IOException {
            if ( _sentence == null || _sentence.length() <= _readbytecount ) {
                _readbytecount = 0;
                _sentence = _sentences[Math.abs(_random.nextInt() % _sentences.length)];
            }

            char ch =  _sentence.charAt(_readbytecount++);
            return ch;
        }

        public int readBuffer(byte[] buffer, int offset, int length) {
            int filled = 0;
            while ( filled < length ) {
                int whichsentence = Math.abs(_random.nextInt() % _sentences.length);
                byte[] bytes = _sentences[whichsentence].toString().getBytes();

                int howmuch = bytes.length;
                if ( bytes.length + filled > length )
                    howmuch = (length -filled);

                System.arraycopy(bytes, 0, buffer, filled+offset, howmuch);
                filled+=howmuch;
            }

            return length;
        }

        public String getAsString() {
            return super.getAsString();
        }
    }
```

In this implementation I've done some optimizing by generating a certain amount of sentences before hand to help make the streaming more efficient. You'll notice I only generate 1 sentence if we're creating a sentence that is going to be less than 1024 bytes, this helps make small operations way more efficient since we don't waste time generating possible sentences that we'd never use. To register this new DTFStream is always a very similar process in the InitHook of your plugin you would simply have the following line:

```
DTFStream.registerStream("words", SentencesInputStream.class);
```

The usage of this new stream would be like so: **`${dtf.stream(words,1024,12345)}`** where 1024 is the size of the phrase to generate and the 12345 is a the seed to use when randomizing the data. The argument to the stream handler could be easily extended to carry other additional parameters if we really wanted such as language or other things that would be interesting to allow the user to control.

Now that the code is done we can take advantage of the existing **JUnit** tests to validate how well our **DTFStream** is working and if it is behaving accordingly to be used within our DTF tests. To run the **JUnit** tests just use the same **junit** target that is available in the foo-dtf example and when you run it for the foot-dtf you'd see that our new **DTFStream** performs pretty well when compared to the other StreamHandlers. One thing to remember is that currently the junit test just validates performance for small and big objects from a single thread perspective. So these values can be much larger when using multiple threads on a machine that can handle that kind of workload, but doing the single threaded analysis gives us stable base numbers. Here's the output that contains performance numbers:

```
[junit] INFO  08/05/2009 14:48:21 StreamingTest   - repeat    Ops/sec:       185528
[junit] INFO  08/05/2009 14:48:22 StreamingTest   - words     Ops/sec:       145348
[junit] INFO  08/05/2009 14:48:23 StreamingTest   - random    Ops/sec:       204918
[junit] INFO  08/05/2009 14:48:23 StreamingTest   - repeat    MB/sec: 2147483647
[junit] INFO  08/05/2009 14:48:30 StreamingTest   - words     MB/sec: 67501993
[junit] INFO  08/05/2009 14:48:31 StreamingTest   - random    MB/sec: 2147483647
```

As you can see the operations per second on small operations is well within the range of the other two existing stream handlers. The megabytes per second is quite slower than the other two but that's because the buffered readBytes() method could be implemented in a more efficient manner using internal buffers to buffer data in a better way. That is not the point of this document, all we wanted to show here is how to build a new **DTFStream** and how to test it for correctness there are a few other tests that run before this to validate that the **DTFStream** handles all types of buffer sizes and that a few critical sizes are handled correctly. Using your new DTFStream is as mentioned before, as simple as referring to the property **`${dtf.stream(words,1024,12345)}`**, which will then make it possible for you to generate data that can easily be regenerated without ever having to save the exact data anywhere.

Share Point Types

Share points are used to communicate between components, they allow you to transfer actions from one thread on a component to another thread on another component or just even locally between threads that are executed on the same component. There are a few built in types in DTF but you may find that your scenario requires a special type of share point. So lets say we wanted a share point that would act like the queue share point but it degrades with time. The idea would be that it only contains the last few **share_sets** that happened within the last second so that older sets are just thrown away. This could be very useful for having lets say a group of readers only reading back data that was just written by another group of writers on a different component. So first lets write the actual share point type code and then we'll use this to write a test that simulates what we just mentioned.

The code for this share point is really going to be simple, we must extend from the class Share and implement the methods ***set*** and ***get***.

```
package com.yahoo.dtf.share;

import java.util.ArrayList;

import com.yahoo.dtf.actions.Action;
import com.yahoo.dtf.actions.flowcontrol.Sequence;
import com.yahoo.dtf.exception.DTFException;
import com.yahoo.dtf.util.TimeUtil;
```

```java
public class DecayQueueShare extends Share {

    public final static String NAME = "dqueue";

    private final static long DECAY_TIME = 1000;

    private ArrayList<ActionHolder> _actions = null;

    private class ActionHolder {
        private Action _action = null;
        private long _inserted = -1;

        public ActionHolder(Action action) {
            _action = action;
            _inserted = TimeUtil.getCurrentTime();
        }

        public boolean isValid() {
            return (TimeUtil.getCurrentTime() - _inserted < DECAY_TIME);
        }

        public Action getAction() { return _action; }
    }

    public DecayQueueShare(String id) throws DTFException {
        super(NAME, id);
        _actions = new ArrayList<ActionHolder>();
    }

    @Override
    public void setAction(Action action) throws DTFException {
        synchronized (_actions) {
            _actions.add(new ActionHolder(action));
        }
    }

    @Override
    public Action getAction() throws DTFException {
        synchronized (_actions) {
            for (int i = 0; i < _actions.size(); i++) {
                ActionHolder aholder = _actions.remove(0);

                // never return something that is older than the DECAY_TIME
                if ( aholder.isValid() )
                    return aholder.getAction();
                else
                    Action.getLogger().info("Action decayed.");
            }

            return new EmptyAction();
        }
    }
}
```

That implementation is pretty straightforward, we've opted to just skip over the entries that have decayed instead of having some background thread checking for stale entries. We have to be careful to always return a valid **Action** even if this Action is just an **EmptyAction** that would be executed and do absolutely nothing. There are **JUnit** tests that validate all registered share point types and validate they do not return nulls or not hold up when accessed concurrently, so make sure to run the **JUnit** unit tests before checking in your new share point.

Now lets use our new queue in a small test that you can find at **foo-dtf/tests/ut/dqueue.xml**, this test is very simple and just uses the queue with two threads setting and getting and making enough of a pause between sets and gets that the system has to throw some elements out. You'll see the "Action decayed." logs that are being printed from the **DecayQueueShare** class. The test itself also checks that there are no actions returned which are older than 1 second and does this by using the fact that all of the properties specified in any action on the **share_set** are resolved at that point in time while the action itself is executed only at **share_get** time, by using that small piece of knowledge we can calculate how long ago this Action was set.

Component usage and locking extensions

Within DTF it is possible to execute your own code at **component** lock time or when a **component** is about to be used. This is useful internally so we can synchronize the properties between the agent and the runner. For other people using DTF this can be useful to check that the component being used has the right libraries setup or that it knows about some other state change that is required before it starts to execute a certain group of actions. When adding new **LockHook** hooks you don't have to worry about performance that much because these are executed at ***lockcomponent*** time and not at any other time during the execution of your test. So spending extra time here to do some desired task is fine since this only happens once at the top of the test for each component being used. There is a JUnit test called **LockHookSuite** that can be used to identify exactly how much time is spent on your specific **LockHook** , but you'll only run this if you find that something is really eating up a lot of time at the start of the test and you want to make sure its not your code (See the **Running JUnit** tests section below for more information on running these tests).

Now when we you start registering new **ComponentHook** hooks you have to be more careful not to take up too much time with your extension otherwise you can effect significantly the performance of certain tasks. There are some tests in place to validate that your changes haven't affected things dramatically, the first is to run some of the existing performance (***tests/perf***) tests such as **eventperf.xml**, which validates the performance of executing an action (***event*** tag) on the ***component*** with the looping on the component side and with the looping on the runner side (effectively issuing a new component call for every action and going through). Comparing the results of the **"local record"** value before and after your changes will give you a good idea of the impact of your new extension. There is also a **JUnit** test that validates what the overhead of each of the registered component hooks is during the sending of an empty action to a group of components, the name of this **JUnit** test is **CompHookSuite** (See the **Running JUnit** tests section below for more information on running these tests).

Other Extensions

There are certainly other areas we may need to make pluggable in the future but I'm certain that all of these are very straightforward to accomplish following the example of existing plugin-in architecture for other features within DTF.

Unit testing extensions

Currently there are some built-in **JUnit** tests that cover the basic testing of some of the extensions mentioned in the previous sections. To run these unit tests just go to the base of the **DTF** directory and after having made a new clean build run ***"ant junit"*** and it will run all of the existing **JUnit** tests that validate the functionality of various extensions (including the ones you just registered). The way it works is that it starts up a fake **DTF** node that runs through the **InitHooks** you've registered and therefore has all of the new extensions registered making it possible to go through each one and validate their behavior against a group of unit tests. If you only want to run one of the Junit suites on its own then you can look for the names of existing suites under **com/yahoo/dtf/junit/XXXSuite.java**, and you can run the ***"ant junit -Djunit.test=XXXSuite"*** to just run that single suite.

Running JUnit tests to validate extensions

When extending functionality sometimes you'll run into the need to know how well this new extension works in comparison to existing implementations. This requirement isn't just as simple as the new extension is more efficient but that it also guarantees the same behavior as the existing extensions and can be used within DTF without any foreseen issues because you forgot to follow some specific rule in the way you generate/manipulate data. To avoid this we've created a Junit test suite that can be executed from the dtf directory itself (plugins can call that ant target from their build.xml just as is done in the plugin-dtf example) when you execute the ***junit*** target you'll execute all of the existing junit test suites which include right now the **StreamingSuite** and the **EventSuite**, we'll be adding more test suites soon but these are the existing ones and you can even execute a suite separately by using the **`-Djunit.test=StreamingSuite`** (the name of the test suite is the name of the class within the package **com.yahoo.dtf.junit**, so if you wanted to add your own test suites in your plugin make sure to place them in the same package structure to take advantage of the existing JUnit task) property to run juts a single JUnit test suite.

If the test suite you execute passes that means your extension is fine from a functional point of view, but you'll have to compare the performance numbers yourself for each of your extensions and see if the number presented is within your expectations for this extension. Remember that all performance numbers are for a single thread doing as much as it can in a loop, so on a multiple CPU machine running many threads can achieve way more than this. Running with just a single thread allows us to achieve stable numbers easily and then those can easily be compared across different implemetnations.

Creating a Yinst package

Creating a Yinst package for your plug-in is so simple it almost didn't deserve its own section. I did create this section because some plug-ins in the near future may require the need to add their own scripts to the Yinst package or the ability to control certain aspects that have been simplified right now. So we already have a place holder for when the Yinst package creation may have more information.

Right now to create a Yinst package all you really need to is open up the *build.xml* that you got from the plugin-dtf that you copied sometime earlier when you were creating your plug-in for DTF and check out the two targets: ***yinst_pkg*** and ***yinst_dist***. When here make sure that *plugin.name* is set to your plug-in's name and then you'll need to version your plugin somehow and set the *plugin.version* property correctly. For those on subversion you can easily use the svn command line to retrieve the current subversion revision number and use that as your version number. Those of you who may still be on CVS will have to tag your source code at the level you're about to create the plugin and use that version to set the *plugin.version* property. You can automate this versioning by just having the task call an external script or tool that will return the exact version number and store that in a property, the ant code would look something like so:

```
<exec executable="./revision" outputproperty="svn.revision"/>
```

The revision script would calculate the revision number and just spit it out to STDOUT which would then be saved in that property for your use.

Adding a new tag

When adding a new tag to DTF there are various things to take into account when coming up with the right XML format for your tag. Since you can choose to use attributes or just child tags to contain your tags information, the pros/cons of using each of these isn't always apparent. The good thing about XML is that even if you don't make the best decision now on your XML format you can later create an XSL stylesheet and changes all of the calls to your tag to a newer and better format.

For this example lets implement a tag that will talk to the DTF DebugServer, now the DebugServer is mentioned in the User's Guide and it's a server that you can telnet to and basically issue commands to check the state of DTF. You can also use this feature to change the logging level and get other DTF related information about the state of the component you are logged into.

So lets start by creating a very simple debugserver tag that will allow us to pass commands directly to the DebugServer and read back the response. There are really only two things to do to make an Action available within DTF:

1. Add it to the XSD of your plugin
2. Create your Java class under the com.yahoo.dtf.actions package so that its visible to the framework. You can place your class in any other sub package of the com.yahoo.dtf.actions pacakge as well.

Adding the XSD entry is quite straightforward and the only thing to be careful doing is to make sure that your tag is referenced from one of the following xs:groups:

- common_commands – your tag will be visible in both component tag and local tag (runner only).
- component_commands – your tag is only visible within a components tag.
- local_commands – your tag is now only visible within the local tag on the runner.

Almost always you reference your tag from the common_commands since your tags are written to be used on both runner and agent. Here's what our XSD changes are:

```
<xs:group name="common_commands">
    <xs:choice>
        <xs:element ref="plugin_action"/>
        <xs:element ref="debugserver"/>
    </xs:choice>
</xs:group>

<xs:element name="debugserver">
    <xs:complexType>
        <xs:attribute name="host" default="localhost"/>
        <xs:attribute name="port" default="40000"/>
    </xs:complexType>
</xs:element>
```

As you can see we referenced the **debugserver** tag from the common_commands group and we created our new **debugserver** tag with two attributes that have default values. Now we need to create our Java class that will be doing the execution of this tag, so here's our first implementation:

```
package com.yahoo.dtf.actions.foo;

import com.yahoo.dtf.actions.Action;
import com.yahoo.dtf.exception.DTFException;
import com.yahoo.dtf.exception.ParseException;

public class Debugserver extends Action {

       private String host = null;

       private String port = null;

       public void execute() throws DTFException {

       }

    public String getHost() throws ParseException { return replaceProperties(host); }
    public void setHost(String host) { this.host = host; }

    public String getPort() throws ParseException { return replaceProperties(port); }
    public void setPort(String port) { this.port = port; }
}
```

Important things to notice from here are that you must call the class exactly the name of the tag with the first letter in the name in upper case (VERY IMPORTANT: never name your class DebugServer because the framework can only lookup the class named Debugserver, this is a simple approach to finding the class). This same rule applies to the way you name your getter and setters for your attributes and you must always make sure to use the **replaceProperties**() tag on your getters because this is where the framework will dynamically resolve any properties in those attributes. There a few other utilities that do property replacement and data type conversion, see the **toInt**(), **toBoolean**(), **parseURI**(), etc. methods as examples.

Now we have to decide how we'll pass the actual command to this tag that will be executed on the DebugServer. We can pass the command as an attribute called command or we can have a child text node for the debugserver tag that we use to get the command from. Lets implement both right now and use how to detect which one has been set. We're doing both because this is an example tag, if you had to pick for your tag you'd have to decide if the information being used was structured or not, mainly because in XML attributes you can only have a piece of text with no enters,tabs,etc because XML parsers normalize this data to not contain any other whitespace characters than a single space. Now if you are trying to pass structured data then use the underlying text node or even another XML child to house that data for you. Firstly we need to add the attribute and tell the XSD that our tag allows text children:

```
<xs:element name="debugserver">
    <xs:complexType mixed="true">
        <xs:attribute name="host" default="localhost"/>
        <xs:attribute name="port" default="40000"/>
        <xs:attribute name="command"/>
    </xs:complexType>
</xs:element>
```

As you can see the `mixed="true"` tells the XSD that there can be a text child under this node and the command attribute is by default optional. So now in the code what do we need to do ? Well first we can extend our class from the **CDATA** that will give us some methods to access our text node and we have to create our attribute command along with its getter and setters, which would give us the following:

```
package com.yahoo.dtf.actions.foo;

import com.yahoo.dtf.actions.util.CDATA;
import com.yahoo.dtf.exception.DTFException;
import com.yahoo.dtf.exception.ParseException;

public class Debugserver extends CDATA {

        private String host = null;

        private String port = null;

        private String command = null;

        public void execute() throws DTFException {

        }

    public String getHost() throws ParseException { return replaceProperties(host); }
    public void setHost(String host) { this.host = host; }

    public String getPort() throws ParseException { return replaceProperties(port); }
    public void setPort(String port) { this.port = port; }

    public String getCommand() throws ParseException { return replaceProperties(command); }
    public void setCommand(String command) { this.command = command; }
}
```

Pretty simple changes so far, now lets actually start using these values in the *execute()* method and lets make the connection to the DebugServer followed by the actual logging of the response. Here is what the new code would look like (omitting imports and getter and setters):

```
...

public class Debugserver extends CDATA {

    ...

    public void execute() throws DTFException {
        Socket socket = null;
        try {
            socket = new Socket(getHost(), getPort());
        } catch (UnknownHostException e) {
            throw new PluginException("Unable to connect to DebugServer.", e);
        } catch (IOException e) {
            throw new PluginException("Unable to connect to DebugServer.", e);
        }

        try {
            InputStreamReader isr = new InputStreamReader(socket.getInputStream());
            BufferedReader reader = new BufferedReader(isr);
            PrintWriter writer = new PrintWriter(socket.getOutputStream());

            if (getCommand() == null && getCDATA() == null)
                throw new ParseException("No command specified.");

            if (getCommand() != null && getCDATA() != null)
                throw new ParseException("Set command with attribute or text node, not both");

            String cmd = getCommand();
            if (cmd == null)
                cmd = getCDATA();

            writer.write(cmd + "\n");
            writer.flush();

            writer.write("quit\n");
            writer.flush();
            String line = null;

            while ((line = reader.readLine()) != null) {
                getLogger().info(line);
            }
        } catch (UnknownHostException e) {
            throw new PluginException("Error talking to DebugServer.", e);
        } catch (IOException e) {
            throw new PluginException("Error talking to DebugServer.", e);
        } finally {
            try {
                if (socket != null)
                    socket.close();
            } catch (IOException e) {
                throw new PluginException(
                        "Error closing connection to DebugServer.", e);
            }
        }
    }

    ...
}
```

You'll notice we're very careful about reporting errors and use different exceptions to classify different types of errors. The ***getPort*** getter was changed to use the ***toInt***() method instead of the ***replaceProperties*** and this way returns the int that we need while doing the replaceProperties underneath and validating our attribute is in fact a number.

Now lets make this tag actually useful and not log the response back to the screen, lets start by creating a utility class that would house our **DebugServerClient** for us, this way our tag code is easier to read and also we move that logic into another class that can be used by other tags if necessary. So here's what our tag would look like now:

```
package com.yahoo.dtf.actions.foo;

import com.yahoo.dtf.actions.util.CDATA;
import com.yahoo.dtf.exception.DTFException;
import com.yahoo.dtf.exception.ParseException;
import com.yahoo.dtf.recorder.Event;
import com.yahoo.dtf.util.DebugServerClient;

public class Debugserver extends CDATA {

    private String host = null;

    private String port = null;

    private String command = null;

    public void execute() throws DTFException {
        Event event = new Event("debugserver");
        DebugServerClient dsc = new DebugServerClient(getHost(),getPort());

        if (getCommand() == null && getCDATA() == null)
            throw new ParseException("No command specified.");

        if (getCommand() != null && getCDATA() != null)
            throw new ParseException("Set command with attribute or text node, not both");

        String cmd = getCommand();
        if (cmd == null)
            cmd = getCDATA();

        event.start();
        String response = dsc.execute(cmd);
        event.stop();
```

```
            event.addAttribute("host", getHost());
            event.addAttribute("port", getPort());
            event.addAttribute("response", response);

            getRecorder().record(event);
        }

        public String getHost() throws ParseException { return replaceProperties(host); }
        public void setHost(String host) { this.host = host; }

        public int getPort() throws ParseException { return toInt("port", port); }
        public void setPort(String port) { this.port = port; }

        public String getCommand() throws ParseException { return replaceProperties(command); }
        public void setCommand(String command) { this.command = command; }
    }
```

I've made sure to present all code in the tag so you can see how its easier to read this version than any other previous one. The code for the ***DebugServerClient*** is in the foo-dtf.tgz tar ball, and its nothing more than an example and would need a lot more coding to be a reusable client that did its own connection pooling and what not. At this point we don't want to dig into that part of the code but instead focus on the features that DTF gives you to use when writing your tag. You'll notice that I introduced the **Event** mechanism very easily by creating our own named event and measuring the piece of the action that we required. We also attached some of our own attributes to this event that would be useful for someone executing the tag to process. The using of the ***getRecorder().record()*** method is what makes this event available in the framework for the tests that use the ***record*** tag to record these events. For an example of using this tag see the **foo-dtf/tests/debugserver.xml.**

Now we should start documenting our Java code using the DTF JavaDoc tags so that we can easily generate the XML documentation at the end for the other users who will be using our tag. I'm not going to add the new code here along with the JavaDoc since you can easily go have a look at the existing source code under ***foo-dtf/src/com/yahoo/dtf/actions/DebugServer.java***. You can embed HTML into any of these JavaDoc elements to make your output have tables and other elements that help show information in a more organized fashion.

We have a functional tag and it can even been used for performance measuring, what are some of the things that could be done better in our example ?

1. We should implement some pooling mechanism for the **DebugServerClient** so that when we call the constructor with the same host and port we'd just get an already connected session and not waste precious cycles reconnecting. This can be done right inside the **DebugServerClient** or you could easily create a pool from which you can do a simple ***checkOut/checkIn*** usage.

2. What if we wanted the previous but we wanted each thread in the system use its own connection ? For this we'd use the ***registerContext()*** and ***getContext()*** methods within our execute method to save any given object (in our case the **DebugServerClient**) between executions of the same tag by the same thread. The two methods mentioned can save any key,object pairs that you'd like and are completely thread safe, so you'll only see the value of a ***getContext()*** from a previous execution of a ***registerContext()*** by the same thread. The usage of the ***getContext/registerContext*** methods has been added to the latest implementation of the DebugServer tag so have a look at the source for how to use these. The only thing missing is the

correct clean up of these **TelnetClients** so that we don't leave clients hanging behind, this can be done implicitly in the **InitHook** in the **CleanUp** method or we can do it explicitly and create a tag for this effect. When using the ***registerContext()*** method you should also make sure your **InitHook** implements the **CleanUpHook** interface and then registers itself for clean up with the ***ReleaseAgent.addCleanUpHook()*** method (foo-dtf example shows how this is used) then when your agents are unlocked at the end of each test you can correctly clean up any state left behind. If you don't clean up the context information then the agent will do this and spam the logs with warnings.

3. Our tag could actually be converted into a group of another tags such as: debugserver_open, debugserver_xmltrace, debugserver_loglevel, debugserver_quit. Then we'd use the same ***registerContext/getContext*** methods to save the opened connection from the debugserver_open to the other subsequent calls on a per thread basis. What would be the advantages of using this tag layout instead ?

    1. You can control which commands are being executed so there's no bad commands being sent to the DebugServer.

    2. You can easily process the output knowing exactly the command you're executing and being able to give more useful attributes in the events returned by each of these tags.

    3. You allow the user that is writing the XML tags to manage his/her connections to each of the open debug server connections. How would you know which connection was open if you wanted to be able to open more than one connection from a single thread ? You'd have to include a clientid attribute in your all your tags, this clientid would be used when looking for the right **DebugServerClient** for that specific call.

    4. You have a finer control of individual events and can easily calculate statistics on different commands being used. It is possible to achieve the same thing with the previous implementation of the debugserver tag but you'd have to use the ***query*** tag to filter on the ***command*** attribute value.

There are always N different ways of doing the same thing in any language and XML has that same flexibility the difference is that even if you commit to a certain XML design now you can at a later date convert your XML syntax to a different one and convert all your old tests using XSL to easily adapt to the need of a different XML notation.

Using Storages

Using storages within your newly created tags is very simple and straightforward. So the first thing to note is that you should stick to naming your attributes that reference storages using names such as uri, inputuri, outputuri, etc. Once you have the attribute defined in your XSD something like this:

```
<xs:element name="import">
    <xs:complexType>
        <xs:attribute name="uri" type="xs:string" use="required" />
        <xs:attribute name="loadFuncs" type="xs:string" default="true" />
        <xs:attribute name="loadRefs" type="xs:string" default="true" />
    </xs:complexType>
</xs:element>
```

You can now create the getter and setter in the new tag class that you're going use like so:

```
public void setUri(String uri) { this.uri = uri; }
public URI getUri() throws ActionException, ParseException { return parseURI(uri); }
```

Using the built-in functions such as ***parseURI*** simplifies your worries about validating the syntax of the **URI** itself. Now that you've got that done in your code itself all you really need to do to get a hold of that storage is to access the **StorageFactory** like so:

```
StorageFactory sf = getStorageFactory();
```

The **StorageFactory** object now has methods to get an OutputStream/InputStream just by passing the uri like so:

```
sf.getInputStream(getUri());
```

There are other methods which allow you to check if a URI exists, get a **BufferedReader** instead of an **InputStream**, etc.

Now when using the same storages on a component you have nothing to change, its all handled nicely by the **StorageFactory** who will figure out which storage you're trying to read and write from and make sure to send any changes back to the **DTFX** when returning to the **DTFX** after executing all the actions on the component.

What APIs can I use when creating new tags?

Everything that is accessible within an action through a method that extends from the parent Action class is yours to use as you wish. This goes for things such as:

- *getComm()*
  - With this method you can now do communication with other components.
  - **NOTE**: this is not for just anyone to use and it merely exists for internal DTF tags.

- *getComponents()*
  - You can get access to the currently locked components and their internal Ids, this is also for internal usage and shouldn't be extended upon, unless a really valid reason comes up.
  - **NOTE**: this is not for just anyone to use and it merely exists for internal DTF tags.

- *getConfig()*
  - This gives you access to all of the properties define with your currently running test. You can use this to set,get and remove properties. Be very careful when removing properties that do not belong to your tag.

- *getContext()*
  - The context can be used to store information for subsequent calls to your tag. This context is thread safe and there is basically a copy of the parent context for every thread that is kept independent during the thread execution.

- *getFunctions()*
  - Gives you access to the currently available functions, again this is for internal use and has very little use to other properties.
  - **NOTE**: this is not for just anyone to use and it merely exists for internal DTF tags.

- *getGlobalContext()*
  - Unlike the the *getContext()* this context object is global and can allow you to share information between your threads or be used to pass information between parent and child actions.

- *getLogger()*
  - use this to do any of your important logging, remember that you can't just log everything because it has an overhead on the amount of work that can be done with your tag. For some tags that are not doing API calls you can log as much as you want. Also when your tag is about to throw an Exception and you want to report more information use the logger to report the error message directly in the logs instead of attaching that message to your exception.

- *getRecorder()*
  - use this to record the Events that your tag may create, remember to document these events in the JavaDocs within your JavaCode (See other tags such as the HTTP tags for examples).

- *getReferences()*
    - used to create referencable elemenst with in the framework. A referencable elment can be defined at one point in the test and later referenced using the attribute ***refid*** to link back to original definition. You'll see this is used by the ***Headergroup*** and ***CookieGroup*** tags.

- *getResults()*
    - Can be used to record test results, but is mainly used internally by other tags such as ***testscript***, ***testsuite*** and ***result.***
    - **NOTE:** this is not for just anyone to use and it merely exists for internal DTF tags.

- *getState()*
    - This gives you direct access to other thread state related items but again shouldn't be used by just any tag.
    - **NOTE:** this is not for just anyone to use and it merely exists for internal DTF tags.

- *getStorageFactory()*
    - This gives you direct access to the **StorageFactory** which can be used for you to register your own storages. This is useful for tags that need to possibly push data to another machine or if they just want to read information from a source that is not currently supported by DTF itself. You just need to create a class that implements the **StorageIntf** and then you'll be able to access those **InputStreams** within tags such as ***query***, ***record***, etc.

Those are the basic APIs that are acessible within the action and the ones that are not marked with the **NOTE:** are all public for a good reason and you should look into using them to do certain activities within your tag. For the ones marked with **NOTE:** you need to understand what the framework is doing at this point and most likely that functionality should be built right into the base DTF framework and not your plugin.