Passing parameters to tools

Overview

This document proposes an updated method of passing parameters to tools when invoking them. We used to do this another way. The intent of this proposal is to make it work properly across all tools on all hubs. **Middleware v2 required.

Steps

Step 1: A tool is launched with various parameters as an argument

a) Parameters are defined with a simple syntax that captures the parameter type, an optional parameter name, and its value:

```
directory(context):/tmp
file(input):/data/groups/testgroup/dropbox/input.txt
file:/home/neeshub/mmclennan/.bashrc
```

Each parameter value has the form type(name):value, where type is either file or directory. We may add other types as time goes on, but weâ€™ll never add something general like string. Itâ€™s far too dangerous to let the application parse untrusted string values, so all types must be well-defined and validated by the middleware.

b) The tool invocation URL takes a params field value with a newline-separated list of parameters. For example:

```
https://nees.org/tools/indeed/invoke/current?params=directory%28contex
t%29%3a%2ftmp%0a%0dfile%28input%29%3a%2fdata%2fgroups%2ftestgroup%2fdr
opbox%2finput.txt
```

As you can see, the ()â€˜s, :â€˜s, and other punctuation characters are encoded to the URL query notation. But the original text before the encoding for this example was quite simple:

```
directory(context):/tmp
file(input):/data/groups/testgroup/dropbox/input.txt
```

Just like the original example, but only the first two parameters. Note the separator characters %0a%0d in the URL. These are good, since theyâ€™ll never conflict with other syntax and they mimic the syntax that we eventually get in the parameter file.

Step 2: The web server receives and validates the information

The web server examines the params field and parses the syntax for all elements. It scans through and validates all elements, checking their type and value. For file and directory types, the given file path must reside in a white-listed set of known places. For NEES.org, the whitelist will include the /home and /nees directories. The web server will also perform small translations on the params string, like collapsing and stripping extra newlines. If a value is bad, or an argument type is not known, the web server should halt processing and display an error.

The web server will not check for the existence of the file or directory because it does not have complete access to the directories and the intended use of the parameter is unknown. Parameters could specify output file names, in which case the file would not exist. File validation is is the responsibilitiy of the application.

Step 3: The middleware is told to start a session with the arguments

The middleware already has an option for appopts. Weâ€™ll leave that alone for backward compatibility and create a similar params option. The params option will receive the URL-encoded params string from the web server. It will decode it using â€˜urllib2.unquote(params).decode(â€œutf8â€•)â€™ and write it to a file called parameters.hz in the userâ€™s session directory. It will also set an environment variable TOOL_PARAMETERS=parameters.hz within the session indicating to the tool that parameter file exists. Thatâ€™s all the middleware needs to do. No need to pass any parameters into the invoke script. The tool (or perhaps the invoke_app wrapper) will take it from there.

Step 4: The tool is invoked.

Many tools will use the new invoke_app script to look for arguments and pass along appropriate arguments. For those like inDEED that may want to handle the arguments themselves, they can. They would simply look for the $TOOL_PARAMETERS environment variable. If set, it points to a file containing the sanitized tool arguments in the form shown earlier:

```
directory(context):/tmp
file(input):/data/groups/testgroup/dropbox/input.txt
file:/home/neeshub/mmclennan/.bashrc
```

The tool would then read this file and parse the various lines to extract arguments.

File Format

Hereâ€™s a proposal for the format of the parameter file. To start with, only 2 types of arguments will be recognized and allowed:

- file parameters (file names)
- directory parameters (directory names)

In the future, we might add others such as url or datastore.

A sample file might look something like this:

```
file:/data/groups/me581/assmt1.txt

file(diagram):/home/nanohub/mmc/indeed/cache0012

directory(context):/tmp
```

In general, the syntax is

```
type:value or type(name):value.
```

The value for the file keyword must be an absolute file path for the desired file. The cms and middleware do not perform file path existence checks. The cms checks the for an allowed type and will only check that the url encoded string matches a set of

The value for the directory keyword must be an absolute directory path. The middleware checks the file path to make sure that the directory exists and throws an error if not.

If we added a datastore type some day, the value might be some sort of table identifier, optional view identifier, and an optional series of rows. For example, a value like

```
datastore:132/stdview/1,5,8
```

where 132 is the datastore table, stdview is the view name, and 1,5,81 are the desired rows. These values are separated by /.

Any parameter can have an optional name, so that clients can request values by name via an API call. Parameters can also be referenced by index, such as #1, #2, and so forth. Typical Use Case

In a typical use case, a professor wants to put a link to a tool in a wiki page, so that students can easily start the tool with a specified file name. The HTTP GET request for that case might look something like this:

```
http://nanohub.org/tools/octaview/invoke/current?params=file%3a%2fdata
%2fgroups%2fme581%2fassmt1.txt
```

Here, params is the parameter string, and there will be only one line in the parameter specification file with

```
 file:/data/groups/me581/assmt1.txt.
```

If a tool needs to load 2 files, then the params list should include two file declarations separated by the escape sequence %0a%0d (CRLF).

Shell Commands

A typical use case for parameter passing is passing one or more file names as command-line arguments. We want to make that easy for many tools and add appropriate error checking.

To do this, we will extend the existing "invoke_app" command to handles the following syntax:

```
invoke_app ... -C  <command> ?-C <command> ...? ?-C <command>?
Tries to find a template that matches a set of parameters
and executes that command.  Each can contain embedded
references to parameters:
@@type(name) ... parameter with the specified type and name
@@type(#N) ..... parameter with the specified type and position #N on
the command line
```

The parameter type can be either "file" or "directory". (More types may be added later.)

Examples

```
invoke_app "$@" -C 'firefox @@file(url)' -C 'firefox'
```

If thereâ€™s a file parameter named url, then run firefox with that as an argument. Otherwise, run firefox with no arguments.

```
invoke_app "$@" -C 'tkdiff @@file(#1) @@file(#2)' -C 'gedit @@file(#1)
'
```

If there are two file parameters, then run â€œtkdiffâ€• on the two files. Otherwise, run â€œgeditâ€• with one file. There is no default command (with no parameters), so there must be at least one file parameter or the tool will throw an error.

Errors

If a matching command can be found and there are no other errors, then the command is executed and thatâ€™s that. But if there are no matching commands, then toolparams shows an error dialog in the tool session. If there are subtle errors (such as extra parameters that are not used in the command template), then toolparams pops up a smaller window in the lower-right corner with error messages.

Rappture API

How to pass parameters to Rappture tools

This document proposes a method of passing parameters to tools built with Rappture.

Small step: Automatically use parameters to set the default values for string Rappture elements.

Use the current parameter syntax to describe the Rappture element to be set. Example:

```
file(input.string(indeck).default):/path/to/file.txt
```

Rappture reads in the parameters.hz file if one exists and sets up an associative array of element paths. The value of each path is the name of a file. In the previous example, the textentry widget in Rappture will check to see if it has a parameter entry for itself (.e.g $path.default) in the global parameters array. If it does, it will open the file and use the contents of the file as the default value (ignoring the specified default value in the tool.xml).

If you use XML paths in parameters, it makes your the tool fragile. If you change your tool.xml, you have to fix all the URLs.

One possible remedy is to create a mapping in the tool.xml. This maps a name to an XML path.

```
<parameters>
<parameter>
<name>indeck</name>
<variable>input.string(indeck).default</variable>
</paramater>
</parameters>
```

Then the parameter would be

```
file(indeck):/path/to/file.txt
```

On the other hand, specifying XML paths has the advantage of setting parameters on any Rappture tool on any XML tag, not just ones that the tool author specified by a parameter mapping.

Bigger step: Extend the parameter passing syntax to include â€œstringâ€œ parameters.

This will let you inline the values in the URL. Example:

```
string(input.number(temperature).default):300K

string(input.choice(model).default):boltzman
```

The real advantage of the inline string is you can then override any tag in Rappture.

```
number(input.number(temperature).min):3K

number(input.number(temperature).max):10000K
```

Loader issues

The parameter passing mechanism has to cooperate with the loader. You could override any XML value with a parameter on construction. This makes sure you load the parameters only once. The loader overwrites the parameter value with itâ€™s default loader value. This happens after the construction of the widgets. If you set parameters after the widgets initial construction, then you can only set current values, not any tag.