Tool Developers

Learn how to create simulation and modeling tools and publish them on a Hub. Sections include:

- Overview of Tool Development Process
- Using Subversion Source Code Control
- Rappture Toolkit for Creating Graphical User Interfaces
- Rappture website

Overview

Tool Development Process

Each hub relies on its user community to upload tools and other resources. Hubs are normally configured to allow any user to upload a tool. The process starts with a particular user filling out a web form to register his intent to submit a tool. This tells the hub manager to create a new project area for the tool. The user then uploads code into a source code repository, and typically develops the code within a workspace or Jupyter Notebook. The user can work alone or with a team of other users. When the tool is ready for testing, the hub manager installs the tool and asks the development team to approve it. Then, the hub manager takes one last look at the tool, and if everything looks good, moves the tool to the "published" state. Of course, a tool can be improved even after it is published, and re-installed, approved, and published over and over again.

The complete process is explained in the tool maintenance documentation for hub managers. Additional details about this process can be found in the following seminars:

- Bootcamp Course for New Developers
- Overview of Tool Development Process
- Using Workspaces
- Using Subversion for Source Code Control

The tool contribution process currently supports use of Subversion and Git repositories.

Creating Graphical User Interfaces

If a tool already has a graphical user interface that runs under Linux/X11, then it can be published as-is, usually in a matter of hours. There are two caveats:

- **If the tool relies heavily on graphics, it may not perform very well within HUBzero execution containers.** Our containers run in cluster nodes without graphics cards, and are therefore configured with MESA for software emulation of OpenGL. This has much poorer performance than ordinary desktop computers with a decent graphics card, so frame rates are much lower. Also, all graphics are transmitted to the user's web browser after rendering, again lowering the frame rate. You can expect to achieve a few frames per second in the hub environment--good enough to view and interact with the data, but far below 100 frames/sec that you would normally see on a desktop computer.
- **Tools running within the hub have access to the hub's local file system--not the user's desktop.** Many tools have a *File* menu with an *Open* option. When a user invokes this option within the hub environment, it will bring up a file dialog showing the hub file system. The user won't see his own local files there unless he uploads them first via sftp, webdav, or the hub's importfile command.

The graphical user interface for any tool published in the hub environment can be created using standard toolkits for desktop applications--including Java, Matlab, Python/QT, etc.

If you're looking for an easy way to create a graphical interface for a legacy tool or simple modeling code, check out the Rappture Toolkit that is included as part of HUBzero. Rappture reads a simple XML-based description of a tool and generates a graphical user interface automatically. It interfaces naturally with many programming languages, including C/C++, Fortran, Matlab, Python, Perl, Tcl/Tk, and Ruby. It creates tools that look something like the following:

Rappture was designed for the hub environment and therefore addresses the caveats listed above. All Rappture-based tools have integrated visualization capabilities that take advantage of hardware-accelerated rendering available on the HUBzero rendering farm. Rappture-based tools also include options to upload/download data from the end user's desktop via the importfile/exportfile commands available within HUBzero.

For more details about Rappture, see the following links:

- Rappture Quick Overview
- Developing Scientific Tools for the HUBzero Platform (introductory course with 7 lectures)
- Rappture Reference Manual

Learn more about HUBzero Tools

Discover the power of HUBzero tools and how easy it is to visualize research using the HUBzero platform.

Jupyter Notebooks:

RStudio:

Launching tools with invoke scripts

Overview

Invoke scripts are small programs, usually written in sh or bash, used to setup the application container environment so the tool can run properly. More specifically, invoke scripts are responsible for:

- Locating tool.xml for Rappture applications
- Setting up the PATH and other optional environment variables
- Starting the window manager
- Starting optional subprograms, like filexfer
- Starting the application

For most applications, the invoke script is a single command that calls the default HUBzero invoke script, named invoke_app, with a few options set. In some rare situations, the tool needs the application container setup in a manner that invoke_app cannot handle. In these cases, the tool developer can modify the tool's invoke script to appropriately setup the application container.

The sections below list out details regarding the options of invoke_app, how to launch Rappture tools using an invoke script that calls invoke_app, and how to launch non-Rappture tools using an invoke script that calls invoke_app.

invoke_app and its options

HUBZero's default tool invocation script is called invoke_app. It is a bash script, usually located in /usr/bin. When called with no options, the script tries to automatically find the needed information to start the applications. There are a number of options that can be provided to alter the script's behavior.

invoke_app accepts the following options:

```
-A tool arguments
-c execute command in background
-C command to execute for starting the tool
-d working directory
-e environment variable (${VERSION} substituted with $TOOL_VERSION)
-f No FULLSCREEN
-S No submit
-n nanowhim version
```

```
-p add to path          (${VERSION} substituted with $TOOL_VERSION)
-r rappture version
-t tool name
-T tool root directory
-u use environment packages
-w specify alternate window manager
```

Here is a detailed description of the options:

-A	Pass the provided enquoted arguments onto the tool. Example usage: `-A "-q blah1 -w blah2"` The options -q and -w are not parsed by invoke_app, but are passed on to the tool
-c	Commands to run in the background before the tool launches. Example usage: `-c "echo hi" -c "filexfer"` This prints "hi" to stdout and starts filexfer
-C	Command to execute for starting tool. Tool's command line arguments can be included in this option, or can be placed in the -A option. Example usage: Call a program, named myprog, located in the tool's bin directory: `-C @tool/bin/myprog` Call a program, named myprog, located in the tool's bin directory, with program arguments "-e val1" and "-b val2":

	`-C "@tool/bin/myprog -e val1 -b val 2"`
	
	Call a program, named myprog, located in the tool's bin directory with arguments -e val1 and -b val2, used in conjunction with invoke_app's -A option:
	
	`-C @tool/bin/myprog -A "-e val1 -b val2"`
	
	Call a program, named myprog, located in the tool's bin directory. We can omit the path of the program if it is an executable and located in the tool's bin directory because the tool's bin directory is added to the PATH environment variable. This would not work for calling a Perl script in a fashion similar to **perl myscript.pl** because in this case, **perl** is executable and **myscript.pl** is the argument.:
	
	`-C myprog`
	
	Call simsim with no arguments:
	
	`-C simsim`
	
	Call simsim with the options -tool and -values, to be parsed by simsim:
	
	`-C "simsim -tool driver.xml -values random"`
	
	Call simsim with the options -tool and -values, to be parsed by simsim:

	`-C simsim -A "-tool driver.xml -values random"`
-d	Change to this working directory. By default change to session directory.
-e	Set an environment variable. Example usage: `-e LD_LIBRARY_PATH=@tool/../${VERSION}/lib:${LD_LIBRARY_PATH}` Within the value part of this option's argument, the text ${VERSION} is automatically substituted with the value of the variable ${TOOL_VERSION}. Similarly, the text @tool is substituted with the value of ${TOOLDIR}. By setting the environment variable, you are overwritting its previous value.
-f	no full screen - disable FULLSCREEN environment variable, used by Rappture, to expand the window to the full available size of the screen.
-p	Prepend to the PATH environment variable. Example usage: `-p @tool/../${VERSION}/bin` Within the value part of this option's argument, the text ${VERSION} is automatically substituted with the value of the variable ${TOOL_VERSION}. Similarly, the text @tool is substituted with the value of ${TOOLDIR}. By setting this option the PATH environment variable is adjusted, but not overwritten. The directory @tool/bin is automatically added to the PATH environment variable.
-r	Sets ${rappture_version} which dictates which version of Rappture is used. If left blank the version will default to the special keyword "system", which represents whichever version is pointed to by the default Rappture environment in "use". A "use -e -r rappture" will be performed to figure out where Rappture is

	installed. If set to the special keyword "none", searching for Rappture executables (rappture, simsim, about) will be skipped and use of these executables will be disabled. This flag works well on hubs where multiple versions of rappture are installed. Users can specify their own version of Rappture to use by updating the PATH environment variable to include the directory where the "rappture" executable is installed.
-S	Disable submit client and run job locally. This flag takes no arguments and is used for debugging. It disables the use of submit client from the -C command that will be executed. The default behavior, when the flag is not given, is to run the command through the submit client unless the command is "rappture", "simsim", "getrappturexml", or "nanowhim", none of which are run through the submit client. Setting the flag on the command line will add your command to the list of commands that do not run with the submit client.
-t	sets ${toolname} which is used while setting up tool paths for TOOLDIR and TOOLXML. ${toolname} is the short name (or project name) of the tool. It is the same as the name used in the source code repository. With respect to the tool contribution process, it is the "toolname" in the path /apps/toolname/version/rappture/tool.xml. Setting this option will change the paths searched while trying to locate tool.xml and the bin directory.
-T	Tool root directory. This is the directory holding a checked out version of the code from the source code repository. It typically has the src, bin, middleware, rappture, docs, data, and examples directories underneath it. With respect to the tool contribution process, it is the "/apps/toolname/version" in the path /apps/toolname/version/rappture/tool.xml. Setting this option will change the paths searched while trying to locate tool.xml and the

	bin directory. Typically when testing this option is used to specify where the tool directory is. In this case, its the present working directory: `-T $PWD`
-u	Set use scripts to invoke before running the tool. Example usage: `-u octave-3.2.4 -u petsc-3.1-real-gnu` These would setup octave-3.2.4 and petsc-3.1 in the environment that your tool would launch in.

-w	Set the window manager. The default value is to use the ratpoison window manager if it exists. If ratpoison is not installed on the system, look for the icewm captive window manager setup. Use this flag to choose an alternative window manager. If your application does not require a window manager specify headless. The possible options are headless, ratpoison, captive, and icewm. If multiple options are specified the first one listed is selected.
	
	Examples:
	
	Use the icewm captive window manager.
	
	`-w captive`
	
	Use no window manager.
	
	`-w headless`

invoke_app is called from within a tool's invoke script. The invoke script is stored in the middleware directory of the tool's source code repository.

Using invoke_app with Rappture tools

Invoke scripts should be placed in the middleware directory of the tool's source code repository. A typical invoke script for a Rappture application looks similar to this:

```
#!/bin/sh

/usr/bin/invoke_app "$@" -t calc \
                         -C rappture
```

In the invoke script above, invoke_app, located in the directory /usr/bin, is called with "$@" , "-t calc", and "-C rappture". "$@" represents all options that the invoke script itself received. "-t calc" tells invoke_app that the toolname is "calc". "-C rappture" tell invoke_app to execute the rappture command. This information is used by invoke_app to figure out which tool it is supposed to be launching and where that tool is installed.

For most Rappture applications, the invoke script is very simple. The above is enough for invoke_app to start looking for a tool.xml file. invoke_app looks for the file named tool.xml. It uses the TOOLDIR variable to help decide where to look. If the tool.xml file is not found in the ${TOOLDIR}/rappture directory, invoke_app will exit explaining that it could not find the tool.xml file. The TOOLDIR variable can be set from the command line using the -T flag:

```
/usr/bin/invoke_app "$@" -t calc \
                         -C rappture \
                         -T ${PWD}
```

Actually, it is more common to see the -T flag provided to a tool's invoke script, and the option is forwarded to invoke_app by "$@":

```
./middleware/invoke -T ${PWD}
```

In the above example, the TOOLDIR variable is set to the present working directory, which is stored in the variable PWD. Specifying the -T option is usually not needed, but can help when invoke_app is confused on what it is supposed to be launching.

Using invoke_app with Jupyter Notebook tools

Invoke scripts should be placed in the middleware directory of the tool's source code repository. A typical invoke script for a Jupyter Notebook application looks similar to this:

```
#!/bin/sh

/usr/bin/invoke_app "$@" -t calc \
                         -C "start_jupyter -t -A -T @tool/bin calc.ipy
nb" \
                         -u anaconda-7 \
                         -r none \
                         -w headless
```

In the invoke script above, invoke_app, located in the directory /usr/bin, is called with "$@", "-t calc", "-C start_jupyter ...", "-c filexfer", "-w captive". "$@" represents all options that the invoke script itself received. "-t calc" tells invoke_app that the toolname is "calc". This information is used by invoke_app to figure out which tool it is supposed to be launching and where that tool is installed. "-C start_jupyter ..." tells invoke_app that the command to run to start the tool is "start_jupyter". "start_jupyter has several typical arguments as shown. "-r none" tells invoke_app that Rappture is not required. "-w headless" tells invoke_app not to start a window manager.

Using invoke_app with GUI tools

Invoke scripts should be placed in the middleware directory of the tool's source code repository. A typical invoke script for a non-Rappture GUI application looks similar to this:

```
#!/bin/sh

/usr/bin/invoke_app "$@" -t calc \
                         -C calc \
                         -c filexfer \
                         -w captive
```

In the invoke script above, invoke_app, located in the directory /usr/bin, is called with "$@", "-t calc", "-C calc", "-c filexfer", "-w captive". "$@" represents all options that the invoke script itself received. "-t calc" tells invoke_app that the toolname is "calc". This information is used by invoke_app to figure out which tool it is supposed to be launching and where that tool is

installed. "-C calc" tells invoke_app that the command to run to start the tool is "calc". In this case calc is UI program built using something other than Rappture. Possible GUI builders include but are not limited to PyQt and MATLAB. "-c filexfer" tells invoke_app to start up the filexfer program before starting the tool's graphical user interface. "-w captive" tells invoke_app to use the icewm captive window manager. For non-rappture applications the icewm captive window manager may be preferred over the ratpoison window manager if there are multiple graphical user interface windows that could popup.

The invoke script above could be made more svelte if the we did not want to start filexfer and we wanted to use the ratpoison window manager. After all, not all applications require files from the user, so they don't need the filexfer program. Here's an example of the tool named calc (the "-t calc" option), that is started by the executable named calc (the "-C calc" option), and uses the default window manager which is ratpoison.

```
#!/bin/sh

/usr/bin/invoke_app "$@" -t calc \
                         -C calc
```

Other invoke script examples

Here are a few common invoke scripts examples that demonstrate using invoke_app options.

Use the -u option to setup Octave-3.2.4 in the path before starting the tool's graphical user interface. The -u option sources a "use" script (octave-3.2.4 in this example) from the /apps/environ directory.

```
#!/bin/sh

/usr/bin/invoke_app "$@" -t calc \
                         -C calc \
                         -u octave-3.2.4
```

Use the -A option to send additional arguments to the command to be executed:

```
#!/bin/sh

/usr/bin/invoke_app "$@" -t calc \
                         -C calc \
```

```
                          -A "-value 13 -value 5 -op add"
```

Or:

```
#!/bin/sh

/usr/bin/invoke_app "$@" -t calc \
                         -C "calc -value 13 -value 5 -op add"
```

Launching a Matlab tool (named app-fermi) with a Rappture graphical user interface:

```
#!/bin/sh

/usr/bin/invoke_app "$@" -t app-fermi \
                         -C rappture
```

Launching a Python tool (named app-fermi) with a Rappture graphical user interface:

```
#!/bin/sh

/usr/bin/invoke_app "$@" -t app-fermi \
                         -C rappture
```

Launching a Java tool (named app-fermi) with a Rappture graphical user interface:

```
#!/bin/sh

/usr/bin/invoke_app "$@" -t app-fermi \
                         -C rappture
```

Combining Tools

Overview

Some of the tools on any hub are really a collection of 3-5 programs acting like a "workbench" for a particular application. Berkeley Computational Nanoscience Class Tools is one such example. It is really a collection of several separate Rappture-based applications, all running on the same desktop, in the same tool session.

We've created a simple window manager called **nanoWhim** that makes it easy to switch back and forth between several applications on a desktop--without all of the fuss and bother associated with a typical window manager. A tool using nanoWhim looks like this:

The combobox at the top lets users switch between applications. Each window that pops up within an application is managed by a set of tabs.

nanoWhim is based on the Whim window manager written in Tcl/Tk. We needed something like this for nanoHUB to create a very simple tabbed interface, so users could easily switch between a couple of tools within the same tool session. A more comprehensive workflow interface is under development, but this simple solution is sometimes useful.

Flipping between tools

The following example shows a Rappture-based application that popped up a separate Jmol application for molecular visualization. Jmol pops up in its own tab, and you can easily switch back and forth between the original application and the Jmol popup by clicking on the tabs, as shown below:

You can click on the **x** on the Jmol tab to close that application.

You can select another application by using the combobox at the very top of the window. That brings up another Rappture-based application, with a different set of inputs and outputs.

You can run each program independently, and the outputs stay separate. If you flip back to the previous application, it will be sitting just the way you left it.

Configuring nanoWhim

To use nanoWhim, you'll need to create two files in the "middleware" directory for your tool: **nanowhimrc** and **invoke**.

The nanowhimrc File

This file configures the various applications that pop up within the tool session. Here's a very simple example:

```
# set an icon
set.config controls_icon header.gif

# first app is an xterm
start.app "Terminal Window" xterm

# second app is a web browser
start.app "Web Browser" firefox
```

Any line that starts with a pound sign (#) is treated as a comment.

The set.config command configures various aspects of the window manager. Right now, the only useful option is controls_icon, which sets the icon shown in the top-left corner of the window. Note that a relative file name is interpreted with respect to the location of the nanowhimrc file itself. In this case, we've assumed that the image header.gif is sitting in the same directory as nanowhimrc.

The rest of the file contains a series of start.app commands for each application that you want to offer. In this case, the first application is called "Terminal Window" and is just an xterm application. The second application is the Firefox web browser, which we label "Web Browser".

Here's a more realistic example:

```
#
# Customize the nanoWhim window manager
#
set.config controls_icon header.gif
```

```
start.app "Average" \
  /usr/bin/invoke_app -t ucb_compnano -T $dir/../rappture/avg

start.app "Molecular Dynamics (Lennard-Jones)" \
  /usr/bin/invoke_app -t ucb_compnano -T $dir/../rappture/ljmd

start.app "Molecular Dynamics (LAMMPS)" \
  /usr/bin/invoke_app -t ucb_compnano -T $dir/../rappture/lammps -u la
mmps-12Feb07

start.app "Monte Carlo (Hard Sphere)" \
  /usr/bin/invoke_app -t ucb_compnano -T $dir/../rappture/hsmc

start.app "Ising Simulations" \
  /usr/bin/invoke_app -t ucb_compnano -C "java -jar $dir/../bin/ising-
1.0.jar"
```

Each start.app command starts a different Rappture-based application. The first argument in quotes is the title of the application, which is displayed in the combobox at the top of the window. The remaining arguments are treated as the Unix command that is invoked to start the application.

The commands shown here all use the /usr/bin/invoke_app script to invoke a Rappture-based application. The -t argument for that script indicates the toolname. The -T argument indicates which directory contains the Rappture tool.xml file. You can use $dir here to locate the directory relative to the nanowhimrc file.

The invoke File

The nanowhimrc file configures the window manager, but the middleware/invoke script actually invokes it. Every tool on nanoHUB has its own invoke script sitting in the middleware directory. Your invoke script should look like this if you want to use nanoWhim:

```
#!/bin/sh

/apps/share/nanowhim/invoke_app "$@" -t ucb_compnano
```

This script invokes the nanoWhim window manager for the tool specified by the -t argument. This is the short name that you gave when you registered your tool. This script looks for the middleware/nanowhimrc file within your source code, and launches nanoWhim with that configuration.

Testing Your Tool

Normally, you develop and test tools within a workspace in your hub. If you're using nanoWhim, that's still true for the individual applications. In other words, you can test each application individually within a workspace. But to get the full effect of the nanoWhim manager running all applications at once, you'll have to get your tool to "installed" status, and then launch the application in test mode. For details about doing this, see the tool maintenance documentation for hub managers or the lecture on Uploading and Publishing New Tools. Look at the tool status page for your own tool project and find the *Launch Tool* button. This is what you would normally do to test any tool before approving it. Once you're in the "installed" stage and you're able to click *Launch Tool*, the nanoWhim configuration should take effect and you'll be able to test the overall combined tool.

Accessing Outside Computing Resources

Overview

Tools are hosted within a "tool session" running within the hub environment. The tool session supports the graphical interface, which helps the user set up the problem and visualize results. If the underlying calculation is fairly light weight (e.g., runs in a few minutes or less), then it can run right within the same tool session. But if the job is more demanding, it can be shipped off to another machine via the "submit" command, leaving the tool session host less taxed and more responsive.

This chapter describes the "submit" command, showing how it can be used at the command line within a workspace and also within Rappture-based tools.

Submit Command

Overview

submit takes a user command and executes it remotely. The objective is to allow the user to issue a command in the same manner as a locally executed command. Multiple submission mechanisms are available for run dissemination. A set of steps are executed for each run submission:

- Destination site is selected
- A wrapper script is generated for remote execution
- If needed a batch system description file is generated.
- Input files for a run are gathered and transferred to the remote site. Transferred files include the wrapper and batch description scripts.
- The wrapper script is executed remotely.
- Progress of the remote run is monitored until completion.
- Output files from the run are returned to the dissemination point.

Command Syntax

submit command options can be determined by using the help parameter of the submit command.

```
$ submit --help
Usage: submit [options]

Options:
  -h, --help            Report command usage. Optionally request listi
ng of
                        managers, tools, venues, or examples.
  -l, --local           Execute command locally
  --status              Report status for runs executing remotely.
  -k, --kill            Kill runs executing remotely.
  --venueStatus         Report venue status.
  -v, --venue           Remote job destination
  -i, --inputfile       Input file
  -p, --parameters      Parameter sweep variables. See examples.
  -d, --data            Parametric variable data - csv format
  -s SEPARATOR, --separator=SEPARATOR
                        Parameter sweep variable list separator
  -n NCPUS, --nCpus=NCPUS
                        Number of processors for MPI execution
  -N PPN, --ppn=PPN     Number of processors/node for MPI execution
```

```
  --stripes=NSTRIPES    Number of parallel local jobs when doing param
etric
                        sweep
  -w WALLTIME, --wallTime=WALLTIME
                        Estimated walltime hh:mm:ss or minutes
  -e, --env             Variable=value
  --runName=RUNNAME     Name used for directories and files created du
ring the
                        run. Restricted to alphanumeric characters
  -m, --manager         Multiprocessor job manager
  -r NREDUNDANT, --redundancy=NREDUNDANT
                        Number of identical simulations to execute in
parallel
  -M, --metrics         Report resource usage on exit
  --detach              Detach client after launching run
  --attach=ATTACHID     Attach to previously detached started server
  -W, --wait            Wait for reduced job load before submission
  -Q, --quota           Enforce local user quota on remote execution h
ost
  -q, --noquota         Do not enforce local user quota on remote exec
ution
                        host
  --tailStdout          Periodically report tail of stdout file.
  --tailStderr          Periodically report tail of stderr file.
  --tail                Periodically report tail of application file.
  --progress            Show progress method. Choices are auto, curses
,
                        submit, text, pegasus, or silent.
  --asynchronous        Asynchronous simulation - results will not be
returned
```

Additional information is available by requesting user specific lists of choices for some command options. The available option lists are generated for a user based on configured restrictions and availability. The values listed here are for example only and may not be available on all HUBs.

```
$ submit --help tools

Currently available TOOLs are:
   lammps-03Mar20-parallel
   lammps-03Mar20-serial
   lammps-05Jun19-parallel
   lammps-05Jun19-serial
```

```
   lammps-11Aug17-parallel
   lammps-11Aug17-serial
   lammps-22Aug18-parallel
   lammps-22Aug18-serial
   lammps-31Mar17-parallel
   lammps-31Mar17-serial

$ submit --help venues

Currently available VENUES are:
   OSG
   brown
   datalimited@brown
   ncn-hub@brown
   standby@brown

$ submit --help managers

Currently available MANAGERs are:
   lammps-03Mar20_mpi
   lammps-03Mar20_serial
   lammps-05Jun19_mpi
   lammps-05Jun19_serial
   lammps-11Aug17_mpi
   lammps-11Aug17_serial
   lammps-22Aug18_mpi
   lammps-22Aug18_serial
   lammps-31Mar17_mpi
   lammps-31Mar17_serial
   mpi
   mpich
   mpirun
   parallel
   serial
```

Examples of how to use the submit command to execute parameter sweeps are provided by asking for help on examples.

```
$ submit --help examples
Usage: submit [options]

Options:
  -h, --help            Report command usage. Optionally request listi
```

```
ng of
                        managers, tools, venues, or examples.
  -l, --local           Execute command locally
  --status              Report status for runs executing remotely.
  -k, --kill            Kill runs executing remotely.
  --venueStatus         Report venue status.
  -v, --venue           Remote job destination
  -i, --inputfile       Input file
  -p, --parameters      Parameter sweep variables. See examples.
  -d, --data            Parametric variable data - csv format
  -s SEPARATOR, --separator=SEPARATOR
                        Parameter sweep variable list separator
  -n NCPUS, --nCpus=NCPUS
                        Number of processors for MPI execution
  -N PPN, --ppn=PPN     Number of processors/node for MPI execution
  --stripes=NSTRIPES    Number of parallel local jobs when doing param
etric
                        sweep
  -w WALLTIME, --wallTime=WALLTIME
                        Estimated walltime hh:mm:ss or minutes
  -e, --env             Variable=value
  --runName=RUNNAME     Name used for directories and files created du
ring the
                        run. Restricted to alphanumeric characters
  -m, --manager         Multiprocessor job manager
  -r NREDUNDANT, --redundancy=NREDUNDANT
                        Number of identical simulations to execute in
parallel
  -M, --metrics         Report resource usage on exit
  --detach              Detach client after launching run
  --attach=ATTACHID     Attach to previously detached started server
  -W, --wait            Wait for reduced job load before submission
  -Q, --quota           Enforce local user quota on remote execution h
ost
  -q, --noquota         Do not enforce local user quota on remote exec
ution
                        host
  --tailStdout          Periodically report tail of stdout file.
  --tailStderr          Periodically report tail of stderr file.
  --tail                Periodically report tail of application file.
  --progress            Show progress method. Choices are auto, curses
,
                        submit, text, pegasus, or silent.
  --asynchronous        Asynchronous simulation - results will not be
returned
```

Parameter examples:

```
submit -p @@cap=10pf,100pf,1uf sim.exe @:indeck
```

Submit 3 jobs. The @:indeck means "use the file indeck as a template file." Substitute the values 10pf, 100pf, and 1uf in place of @@cap within the file. Send off one job for each of the values and bring back the results.

```
submit -p @@vth=0:0.2:5 -p @@cap=10pf,100pf,1uf sim.exe @:indeck
```

Submit 78 jobs. The parameter @@vth goes from 0 to 5 in steps of 0.2, so there are 26 values for @@vth. For each of those values, the parameter @@cap changes from 10pf to 100pf to 1uf. 26 x 3 = 78 jobs total. Again @:indeck is treated as a template, and the values are substituted in place of @@vth and @@cap in that file.

```
submit -p params sim.exe @:indeck
```

In this case, parameter definitions are taken from the file named params instead of the command line. The file might have the following contents:

```
# paramters for my job submission
parameter @@vth=0:0.2:5
parameter @@cap = 10pf,100pf,1uf
```

```
submit -p "params;@@num=1-10;@@color=blue" job.sh @:job.data
```

For someone who loves syntax and complexity... The semicolon separates the parameters value into three parts. The first says to load parameters from a file params. The next part says add an additional parameter @@num that goes from 1 to 10. The last part says add an additional parameter @@color with a single value blue. The parameters @@num and @@color cannot override anything defined within params; they must be new parameter names.

```
submit -d input.csv sim.exe @:indeck

        Takes parameters from the data file input.csv, which must be i
n comma-
separated value format. The first line of this file may contain a seri
es of
@@param names for each of the columns. Whitespace is significant for a
ll
values entered in the csv file. If it doesn't, then the columns are as
sumed to
be called @@1, @@2, @@3, etc. Each of the remaining lines represents a
 set of
parameter values for one job; if there are 100 such lines, there will
be 100
jobs. For example, the file input.csv might look like this:

                @@vth,@@cap
                1.1,1pf
                2.2,1pf
                1.1,10pf
                2.2,10pf

        Parameters are substituted as before into template files such
as
@:indeck.

submit -d input.csv -p "@@doping=1e15-1e17 in 30 log" sim.exe @:infile

        Takes parameters from the data file input.csv, but also adds a
nother
parameter @@doping which goes from 1e15 to 1e17 in 30 points on a log
scale.
For each of these points, all values in the data file will be executed
. If the
data file specifies 50 jobs, then this command would run 30 x 50 = 150
0 jobs.

submit -d input.csv -i @:extra/data.txt sim.exe @:indeck

        In addition to the template indeck file, send along another fi
le
extra/data.txt with each job, and treat it as a template too.

submit -s / -p @@address=23 Main St.,Hometown,Indiana/42
Broadway,Hometown,Indiana -s , -p @@color=red,green,blue job.sh @:job.
```

```
data

        Change the separator to slash when defining the addresses, the
n change
back to comma for the @@color parameter and any remaining arguments. W
e
shouldn't have to change the separator often, but it might come in han
dy if
the value strings themselves have commas.

submit -p @@num=1:1000 sim.exe input@@num

        Submit jobs 1,2,3,...,1000. Parameter names such as @@num are
recognized not only in template files, but also for arguments on the c
ommand
line. In this case, the numbers 1,2,3,...,1000 are substituted into th
e file
name, so the various jobs take their input from "input1", "input2", ..
.,
"input1000".

submit -p @@file=glob:indeck* sim.exe @@file

        Look for files matching indeck* and use the list of names as t
he
parameter @@file. Those values could be substituted into other templat
e files,
or used on the command line as in this example. Suppose the directory
contains
files indeck1, indeck10, and indeck2.  The glob option will order the
files in
a natural order: indeck1, indeck2, indeck10.  This example would launc
h three
jobs using each of those files as input for the job.

submit -p @@file=globnat:indeck* sim.exe @@file

        This option has been deprecated.  The functionality is now ava
ilable
with the glob option.
```

By specifying a suitable set of command line parameters it is possible to execute commands on configured remote systems. The simple premise is that a typical command line can be prefaced

by submit and its arguments to execute the command remotely.

```
$ submit -v clusterA echo Hello world!
Hello world!
```

In this example the echo command is executed on the venue named clusterA where runs are executed directly on the host. Execution of the same command on a cluster using a batch scheduler such as SLURM would be done in a similar fashion

```
$ submit -v clusterB echo Hello world!
(2586337) Simulation Queued Wed Oct  7 14:45:21 2009
(2586337) Simulation Done Wed Oct  7 14:54:36 2009
$ cat 00577296.stdout
Hello world!
```

submit supports an extensible variety of submission mechanisms. HUBzero supported submission mechanisms are

- local - use batch submission mechanisms available directly on the submit host. These include condorHT, and Pegasus batch queue submission.
- ssh - direct use of ssh. Submit manages access to a venue using a common ssh key, essentially serving as a proxy for the HUB user.
- ssh + remote batch submission - use ssh to do batch run submission remotely. Again methods for common batch schedulers PBS, condorHT, Pegasus, and SLURM are provided. Additional interfaces to SGE, Load Leveler, BOINC, LSF, and Tapis are also available.

In addition to single site submission the -r/--redundancy option provides the option to simultaneously submit runs to multiple remote venues. In such cases the successful completion of a run at one venue cancels runs at all other venues. If none of the runs are successful results from one of the runs are returned to the user. Redundant submission is not allowed when performing parametric sweeps.

A venue for remote execution is selected in one of the following ways, listed in order of precedence:

- Execute the command within the user tool session, -l/--local option
- User specified on the command line with -v/--venue option.
- Randomly selected from remote sites associated with pre-staged application.
- Select randomly from all configured sites

Venues that do not meet the resource requirements of the run request are not considered.

Venues are typically configured with limits on the number of cores, walltime, or core-hours.

Any files specified by the user plus internally generated scripts are packed into a tarball for delivery to the remote site. Individual files or entire directory trees may be listed as command inputs using the -i/--inputfile option. In addition, command arguments that exist as files or directories will be packed into the tarball. If using ssh based submission mechanisms the tarball is transferred using scp.

The job wrapper script is executed remotely either directly or submitted to a batch queue. The job is subject to all remote queuing restrictions and idiosyncrasies.

Remote batch jobs are monitored for progress. Methods appropriate to the batch queuing system are used to check job status at a configurable frequency. A typical frequency is on the order one minute. Job status changes are reported to the user. The maximum time between reports to the user is set on the order of five minutes even in the absence of change. The job status is used to detect job completion.

The same methods used to transfer input files are applied in reverse to retrieve output files. Any files and directories created or modified by the application are be retrieved. A tarball is retrieved and expanded to the home base directory. It is up to the user to avoid the overwriting of files.

In addition to the application generated output files additional files are generated in the course of remote run execution. Some of these files are for internal bookkeeping and are consumed by submit, a few files however remain in the home base directory. The remaining files include RUNID.stdout and RUNID.stderr, it is also possible that a second set of standard output/error files will exist containing the output from the batch job submission script. By default, RUNID represents unique job identifier assigned by submit. If preferred a user can specify a different RUNID using the --runName command argument.

Pegasus Workflow Submission

Overview

Functionality has been included in submit to support workflow management using Pegasus. Two use cases are available: automatic workflow generation for parametric sweeps on one or more variables, or user constructed workflows. In both instances submit is used to configure access to one or more computational resources eliminating the need for a user to supply a site catalog thereby simplifying use of the workflow management system.

Parametric Sweeps

submit command options -p/--parameters and -d/--data provide support for specifying parameter sweeps in a compact general way. The user is relieved of the chore of generating entire sets of input files and command arguments comprising a parameter sweep. Substitutable parameters are declared on the submit command line. Values of these parameters can then be systematically substituted into data files or application command line parameters. submit performs the necessary substitutions to cover all parameter combinations. Each combination of parameters is abstractly represented as a node in a workflow and concretely executed as a job on the designated computational resource. A simple curses interface is provided to monitor progress of the simulation run.

User Constructed Workflows

Parameter sweeps are represented as a simple workflow consisting of many individual independent nodes. That is data is not shared between nodes or jobs in the run. There are cases where this simple approach is not sufficient to describe a workflow required to achieve a developer's or user's objective. Under these circumstances a developer may create a workflow and build an application around it where the user supplies values for selected inputs. In such cases the Pegasus API's may be used to generate the abstract workflow description in the form of a dax file. The dax file can then executed by a simple submit command.

```
submit pegasus-plan --dax daxFile
```

In cases where more than one venue is capable of executing Pegasus runs a specific venue can be requested on the command line, otherwise submit will choose a venue at random.

```
submit -v DiaGrid pegasus-plan --dax daxFile
```

There are several additional options to pegasus-plan command that are supplied by submit. A few of the command options may be provided on the command line. submit reserves the option to silently ignore options as it sees fit.

In addition to remote execution of Pegasus runs it is also possible to do the execution locally with in the tool session. Simply use the submit -l/--local option.

```
submit --local pegasus-plan --dax daxFile
```

The use command can be employed to put pegasus-plan and all other Pegasus commands in the PATH environment variable. In additional to setting PATH, other environment variables are set allowing use of the Python and java dax generation API's.

Rappture Integration with Submit

Overview

It is possible to use the submit command to execute simulation jobs generated by Rappture interfaces remotely. A common approach is to create a shell script which can exec'd or forked from an application wrapper script. This approach has been applied to TCL, Python, Octave, MATLAB, and Perl wrapper scripts. To avoid consumption of large quantities of remote resources it is imperative that the submit command be terminated when directed to do so by the application user (Abort button).

TCL Wrapper Script

submit can be called from a TCL Rappture wrapper script for remote batch job submission. An example of what code to insert in the wrapper script is detailed here.

An initial code segment is required to catch the Abort button interrupt. Setting execctl to 1 will terminate the process and any child processes.

```
package require Rappture
Rappture::signal SIGHUP sHUP {
   puts "Caught SIGHUP"
   set execctl 1
}
Rappture::signal SIGTERM sTERM {
   puts "Caught SIGTERM"
   set execctl 1
}
```

A second code segment is used to build an executable script that can executed using Rappture::exec. The trap statement will catch the interrupt thrown when the wrapper script execution is Aborted. Putting the submit command in the background allows for the possibility of issuing multiple submit commands from the script. The wait statement forces the shell script to wait for all submit commands to terminate before exiting.

```
   set    submitScript "#!/bin/sh\\n\\n"
   append submitScript "trap cleanup HUP INT QUIT ABRT TERM\\n\\n"
   append submitScript "cleanup()\\n"
   append submitScript "{\\n"
   append submitScript "   kill -TERM `jobs -p`\\n"
```

```
   append submitScript "   exit 1\\n"
   append submitScript "}\\n\\n"

   append submitScript "cd [pwd]\\n"
   append submitScript "submit -v cluster -n $cores -w $walltime\\\\\\
n"
   append submitScript "       COMMAND ARGUMENTS &\\n"
   append submitScript "sleep 5\\n"
   append submitScript "wait\\n"

   set submitScriptPath [file join [pwd] submit_script.sh]
   set fid [open $submitScriptPath w]
   puts $fid $submitScript
   close $fid
   file attributes $submitScriptPath -permissions 00755
```

The standard method for wrapper script execution of commands can now be used. This will stream the output from all submit commands contained in submit_script.sh to the GUI display. The same output will be retained in the variable out.

```
set status [catch {Rappture::exec $submitScriptPath} out]
```

Each submit command creates files to hold COMMAND standard output and standard error. The file names are of the form JOBID.stdout and JOBID.stderr, where JOBID is an 8 digit number. These results can be gathered as follows.

```
   set out2 ""
   foreach errfile [glob -nocomplain *.stderr] {
      if [file size $errfile] {
         if {[catch {open $errfile r} fid] == 0} {
            set info [read $fid]
            close $fid
            append out2 $info
         }
      }
      file delete -force $errfile
   }
   foreach outfile [glob -nocomplain *.stdout] {
      if [file size $outfile] {
         if {[catch {open $outfile r} fid] == 0} {
            set info [read $fid]
            close $fid
            append out2 $info
         }
```

```
      }
      file delete -force $outfile
   }
```

The script file should be removed.

```
file delete -force $submitScriptPath
```

The output is presented as the job output log.

```
$driver put output.log $out2
```

All other result processing can proceed as normal.

Python Wrapper Script

submit can be called from a python Rappture wrapper script for remote batch job submission. An example of what code to insert in the wrapper script is detailed here.

An initial code segment is required to import some predefined functions that manage typical aspects of remote submission. An important aspect is the handling of user interruption via the Abort button.

```
import os
from Rappture.tools import executeCommand as RapptureExec
```

A second code segment is used to build a list containing an executable submit command to be executed using RapptureExec. RapptureExec will trap signals initiated by pressing the Abort button. The submit command must terminate before RapptureExec exits and returns control to the application wrapper script.

```
   submitCommand = ["submit","-v",venue,"-n",cores,
                              "-w",walltime,COMMAND,ARGUMENTS]
   exitStatus,stdOutput,stdError = RapptureExec(submitCommand)
```

The standard method for wrapper script execution of commands can now be used. This will stream the output from the submit command to the GUI display. The same output will be retained in the variable stdOutput.

The submit command creates files to hold COMMAND standard output and standard error. By default the file names are of the form JOBID.stdout and JOBID.stderr, where JOBID is an 8 digit number. These results can be gathered as follows.

```
reStdout = re.compile(".*.stdout$")
reStderr = re.compile(".*.stderr$")

out2 = ""
errFiles = filter(reStderr.search,os.listdir(os.getpwd()))
if errFiles != []:
   for errFile in errFiles:
      errFilePath = os.path.join(os.getpwd(),errFile)
      if os.path.getsize(errFilePath) > 0:
         f = open(errFilePath,'r')
         outFileLines = f.readlines()
         f.close()
         stderror = ''.join(outFileLines)
         out2 += 'n' + stderror
      os.remove(errFilePath)

outFiles = filter(reStdout.search,os.listdir(os.getpwd()))
if outFiles != []:
   for outFile in outFiles:
      outFilePath = os.path.join(os.getpwd(),outFile)
      if os.path.getsize(outFilePath) > 0:
         f = open(outFilePath,'r')
         outFileLines = f.readlines()
         f.close()
         stdoutput = ''.join(outFileLines)
         out2 += 'n' + stdoutput
      os.remove(outFilePath)
```

The output is presented as the job output log.

```
lib.put("output.log", out2, append=1)
```

All other result processing can proceed as normal.

Perl Wrapper

submit can be called from a perl Rappture wrapper script for remote batch job submission. An example of what code to insert in the wrapper script is detailed here.

An initial code segment is required to catch the Abort button interrupt.

```
use Rappture

my $ChildPID = 0;

sub trapSig {
    print "Signal @_ trapped\\n";
    if($ChildPID != 0) {
        kill 'TERM', $ChildPID;
        exit 1;
    }
}
$SIG{TERM} = \&trapSig;
$SIG{HUP}  = \&trapSig;
$SIG{INT}  = \&trapSig;
```

A second code segment is used to build an executable script that can executed using Rappture.tools.getCommandOutput. The trap statement will catch the interrupt thrown when the wrapper script execution is Aborted. The wait statement forces the shell script to wait for the submit command to terminate before exiting.

```
$SCRPT = "submit_app.sh";
open(FID,">$SCRPT");
print FID "#!/bin/sh\\n";
print FID "\\n";
print FID "trap cleanup HUP INT QUIT ABRT TERM\\n\\n";
```

```
print FID "cleanup()\\n";
print FID "{\\n";
print FID "   kill -s TERM `jobs -p`\\n";
print FID "   exit 1\\n";
print FID "}\\n\\n";

print FID "submit -v cluster -n $cores -w $wallTime COMMAND ARGUMENTS
&\\n";
print FID "wait %1\\n";
print FID "exitStatus=\\$?\\n";
print FID "exit \\$exitStatus\\n";
close(FID);
chmod 0775, $SCRPT;
```

The standard fork and exec method for wrapper script execution of commands can now be used. Using this approach does not allow streaming of the command outputs.

```
if      (!defined($ChildPID = fork())) {
    die "cannot fork: $!";
} elsif ($ChildPID == 0) {
    exec("./$SCRPT") or die "cannot exec $SCRPT: $!";
    exit(0);
} else {
    waitpid($ChildPID,0);
}
```

Each submit command creates files to hold COMMAND standard output and standard error. The file names are of the form JOBID.stdout and JOBID.stderr, where JOBID is an 8 digit number. These results can be gathered with standard perl commands for file matching, reading, etc. All other result processing can proceed as normal.

Octave/MATLAB Script

submit can be called from a Octave or MATLAB Rappture wrapper script for remote batch job submission. An example of what code to insert in the wrapper script is detailed here.

```
 -- Function: [EXITSTATUS] = rpExec(COMMAND,STREAMOUTPUT)
 -- Function: [EXITSTATUS, STDOUTPUT] = rpExec(COMMAND,STREAMOUTPUT)
 -- Function: [EXITSTATUS, STDOUTPUT, STDERROR] = rpExec(COMMAND,STREA
```

```
MOUTPUT)

     Execute COMMAND with the ability to terminate the process upon
     reception of a interrupt, hangup, or terminate signal. Doing so
     allows the process to terminated when the Rappture "Abort" button
     is pressed.  COMMAND should contain a set of strings that compris
e
     the command to be executed.  If STREAMOUTPUT equals 1 the stdou
     and stderr from COMMAND are piped back to the current process
     stdout and stderr descriptors as COMMAND executes.

     On output EXITSTATUS indicates whether or not an error occurred.
     EXITSTATUS equals 0 indicates that no error occurred. If STDOUTPU
T
     is supplied it will contain a copy of stdout from COMMAND.  In th
e
     same manner if STDERROR is supplied it will contain a copy of
     stderr from COMMAND.

     Example:

     [exitStatus,stdOutput,stdError] = rpExec({"submit","-wallTime","3
0","lammps-12Feb14-serial","-in","lmp.in"},1);
```

Accesing your home directory

Overview

Accessing your home directory on the HUB is easy with the three methods described in this section. While sFTP is the most common, you will find which method works best for you.

sFTP

Accessing your home directory via sFTP

sFTP, or secure FTP, is a program that uses SSH to transfer files. Unlike standard FTP, it encrypts both commands and data, preventing passwords and sensitive information from being transmitted in the clear over the network. It is functionally similar to FTP, but because it uses a different protocol, you can't use a standard FTP client to talk to an sFTP server, nor can you connect to an FTP server with a client that supports only sFTP.

The following tutorial should help you in using sFTP to connect to and from your HUBzero server(s).

Graphical Clients

Using graphical SFTP clients simplifies file transfers by allowing you to transmit files simply by dragging and dropping icons between windows. When you open the program, you will have to enter the name of the host (e.g., yourhub.org) and your HUB username and password.

Windows Clients

- WinSCP
- BitKinex
- FileZilla
- PuTTY

Mac OSX Clients

- Transmit
- Fetch
- Cyberduck
- Flow
- Fugu

Command-line

You can use command line SFTP from your Unix account, or from your Mac OS X or Unix workstation. To start an SFTP session, at the command prompt, enter:

```
yourmachine:~ you$ sftp username@host
yourmachine:~ you$ username@host password:

host ~
```

Some standard commands for command-line sFTPCommand Description

cd	Char
chmod	Char comp
chown	Char comp
dir (or ls)	List t remo
exit (or quit)	Close and e
get	Copy local
help (or ?)	Get h
lcd	Char
lls	See a the lo
lmkdir	Crea
ln (or symlink)	Crea comp
lpwd	Show direc
lumask	Char
mkdir	Crea
put	Copy remo
pwd	Show direc
rename	Rena
rm	Delet
rmdir	Rem direc
version	Displ
!	In Ur enter SFTP !pwd) dropp

WebDAV

Accessing your home directory via WebDAV

WebDAV is the Distributed Authoring and Versioning extension to the standard HTTP/HTTPS web protocol. It allows a client to browse a remote filesystem, usually with a graphical browser that makes it appear that your files are on your desktop. You may access your hub storage using only the secure version of this service (HTTPS). We do not support HTTP. Most modern computer platforms support HTTPS transport for WebDAV with either small adjustments or freely available software.

Linux/Unix

Firefox and Chrome browsers provide text-mode support by connecting to `https://hubname.org/webdav`. You will be prompted for hub login and password. If you use the KDE graphical desktop environment, you can access your hub storage with the Konqueror browser by typing the special URL `webdavs://hubname.org/webdav/` into the Location field of the browser. It will prompt you for your hub login and password. Thereafter, you traverse your home directory by clicking on folders and you can drag and drop files to your desktop.
Cadaver is a text-mode WebDAV browser. It can be used if it is compiled with SSL support. Invoke it with the command `cadaver https://hubname.org/webdav/` and it will prompt you for your hub login and password. You can then use it in a manner similar to FTP.
If you are using Linux, you can use the davfs kernel module to mount your hub storage area as a local filesystem.

Macintosh

MacOS versions 10.4 and higher support HTTPS transport for WebDAV using the Finder.

1. Select the Go menu in the Finder and choose "Connect to Network Server".
2. Enter the URL `https://hubname.org/webdav/` into the address field.
3. When prompted, enter your Network ID credentials.

You should now be able to drag files and folders between your computer and the site to which you just connected.

Windows 10, 8.1, 8 and 7

Windows 10, 8.1 and 8 use the **WebClient Services** to connect to a WebDAV Servers, by default the WebClient service is disabled, so we need to enable it.

1. From the Start menu, choose Control Panel, then System and Security, then Administrative Tools, and then Services.
2. Scroll down to WebClient, set the service to Automatic, and then click Apply.
3. If the service is not already running, click Start.
4. Click OK to close the Control Panel and close other windows.

Windows 10

To set up a WebDAV connection in Windows 10:

1. From the **Start Menu** go to **File Explorer** and select **This PC** on the left hand pane
2. Select **Computer** from the top ribbon
3. Click on **Map Network Drive**
4. Click **Connect to a Web site that you can use to store your documents and pictures.**
5. Click **Next**
6. Select **Choose another network location** and click **Next**
7. Enter "**`https://hubname.org/webdav/`**". Replace "hubname.org" with the URL of the destination hub and click **Next**
8. Enter your **password**, and click **Ok**
9. Click **Next**, then **Finish**
10. When prompted, enter your Hub credentials.
11. You should see a new Network Drive under your Computer/This PC. Double click on it to open.

You should now be able to drag files and folders between your computer and the hub via the network drive to which you just connected.

Windows 8.x

To set up a WebDAV connection in Windows 8.x:

1. Using the Search interface in tile mode, locate and select the Computer tile.
2. In the quick menu at the top of the screen, click Map Network Drive.
3. In the "Folder" field, enter a URL that points to the destination hub similar to the following URL "`https://hubname.org/webdav/`" Replace "hubname.org" with the URL of the destination hub.
4. Select the Connect using different credentials box, and then click Finish.
5. When prompted, enter your Hub credentials.
6. You should see a new Network Drive under your Computer/This PC. Double click on it to open.

You should now be able to drag files and folders between your computer and the hub via the network drive to which you just connected.

Windows 7

To set up a WebDAV connection in Windows 7.

1. From the Start menu, right-click Computer, and select Map network drive.
2. Enter a URL that points to the destination hub similar to the following URL "`https://hubname.org/webdav/`". Replace "hubname.org" with the URL of the destination hub. Clink finish.
3. When prompted, enter your Hub credentials.
4. You should see a new Network Drive under your Computer/This PC. Double click on it to open.

You should now be able to drag files and folders between your computer and the hub via the network drive to which you just connected.

If you have difficulty dragging and dropping, right-click the file or folder you want to copy, and choose Copy. Then right-click the directory you want to put it in, and choose Paste.

filexfer (in Workspace tool)

Accessing your home directory files via filexfer (in Workspace tool)

Filexfer, short for 'file transfer', is a utility that you can call from within the workspace tool to transfer a file into your home directory from your local machine or download a file from your home directory. Type 'filexfer' in the xterm provided to you when you start the workspace tool. You should now see the filexfer GUI as it appears in the image below.

Click on "Upload" or "Download" for the desired action. Please note, that you must enable pop-ups in your browser to proceed. The pop-up windows that appear allow you to browse or download per your selected action.

filexfer may also be included by other contributed tools as a secondary application. This provides a consistent and reliable method for exchanging files with tool sessions.

Tool Paths

Overview

Providing the tool user access to input and output files is an important part of building a tool. Where can I place example files that a user will select on first use? Where can I place temporary generated files during the tool runtime? Where can I save a user's work? Where can I place simulation results for the user? Can these results be available in a future session? All these questions and more are addressed in the following sections.

Environment Variables

Environment Variables

A number of environment variables are available in a tool session. A few are discussed here. A full list can be viewed by running the "env" command from a terminal in the workspace tool.

*Note: tools are invoked by the user's account and permissions set accordingly. A tool can save files to a user's home directory because the tool runs as that user.

SESSION="session id"

This variable stores the session ID or session number that is currently active. It's the ID of the session you are currently using.

USER="username"

This variable stores the current username of the user running the tool.

SESSIONDIR=/home/"hub hostname"/"username"/data/sessions/"session id"

This variable stores the current session directory of the open tool. This session directory is a separate directory created for each new tool session. This is a good place for temporary files generated by your tool.

*This is the default path for a tool on invoke.

RESULTSDIR=/home/"hub hostname"/"username"/data/results/"session id"

This variable stores the results directory located in the user's home directory. This is a good place to place simulation results for the user to access later. This is where Rappture saves results.

*Be mindful of the user's quota limits.

PWD="present working directory"

This variable stores the present working directory.

HOME=/home/"hub hostname"/"username"

This variable stores the path of the user's home directory. This is useful if a tool provided an option to save the user's current work. Please create a directory for the tool to save files here, to prevent cluttering the user's home directory too much. A best practice would be to create a new directory for a tool in the user's data directory. For example, "$HOME/data/toolname".

Passing path variables with the Invoke Script

Overview

Passing variables for use in the runtime tool environment is a common occurrence, in particular "@tool" .

See the full invoke_app documentation.

@tool

The variable "@tool" can be passed into your tool via the invoke script. This is important information for you tool to know so that the tool can access example input files and static data files that reside in the respective directories. There are two way to pass the "@tool" location via the invoke script to the tool.

1) As an argument to the tool

    -A @tool

2) As an environment variable

    -e TOOL_REPO_PATH=@tool

Example Files

Small Files

Files less than 100MB, can be placed in the 'data' or 'examples' directory within the tool repository.

Large Files

Files greater than 100MB, should be placed in the appropriate /data directory outside of the tool repository. This path can be accessed directly. In typical situations /data is organized by tool and/or group. Either of these locations is suitable for maintaining large static datasets.

Note: this is by special request only, please contact your HUB administrator.

Tool Generated Files

Temporary Files at runtime

Temporary files that are generated by a tool at runtime should be written to the default session directory using the SESSIONDIR environment variable. Ideally, these temporary files should be removed when they are no longer needed. Storing temporary files in the ${SESSIONDIR} affords the user an opportunity to purge them later using the storage manager available on the member dashboard.

Temporary files can also be written to /tmp. The /tmp directory is not shared with other sessions and is cleared when a session is terminated.

Simulation Results Files

Simulation output files that are generated by a tool should be written to the results directory using the RESULTSDIR environment variable. This directory is created in the user's home directory for the user to easily find the simulation results. The tool may also read from the directory and present a list of the result files to the user.

Importing and exporting user files

Overview

We provide two scripts to facilitate the uploading and downloading of files for a tool. This allows users to upload their own input file to the tool via the web interface. The export script allows a user to download a tool simulation result directly from the tool via the web interface without accessing their home directory separately.

Both scripts are included in the filexfer application. filexfer can be used as a secondary application with standard tools as a means for users to transfer files between their computer and tool sessions. Rappture tools do not need the secondary application as file transfer methods are already included in the UI.

Import File

Overview

"importfile" is the command line tool that, when run, opens a pop-up window prompting the user to browse and select file(s) from their computer to be uploaded. If a filename is provided for the imported file the pop-up window can alternatively be used to paste text from the user clipboard. This is most useful in workspace tools which do not allow text to be directly pasted to terminals or other applications.

You can use this command to transfer one or more files from your desktop to your tool session via a web browser. This command causes a web page to pop up prompting you for various files on your desktop. Choose one or more files and submit the form. The selected files will be uploaded to your tool session and saved with the file names specified on the command line. **You must have popups enabled for this to work properly.**

Implementation

This script should be implemented as a background process in a non-rappture tool. Typically a pipe is used to run this script off the main process of the tool. The piped process should be monitored by the tool code for a response upon completion of the user's file upload. The script will continue to wait for a file to be uploaded indefinitely. Please code appropriately for this.

Help text

```
USAGE: /usr/bin/importfile [-f|--for text] [-l|--label text] file file
 ...

  options:
    -h or --help
      Prints this help message.

    -f or --for <text>
      Short explanation of what the data will be used for; for
      example, "for CNTBands 2.0".  If given, this text is inserted
      into the upload form to help explain what it will be used for.

    -l or --label <text>
      Prompt for subsequent file arguments using this label string.
      The default label just uses the file name.
```

```
-m or --mode acsii|binary|auto
  In "binary" mode, files are transferred exactly as-is.  In
  "ascii" mode, control-M characters are removed, which helps
  when loading Windows files into the Linux environment.  The
  default is "auto", which removes control-M from text files
  but leaves binary files intact.

-p or --provenance
  Print more verbose results showing the provenance information
  for all files uploaded.  Instead of a series of space-separated
  file names, this produces one line for each file showing the
  final file name and where it came from, which is either the
  file name on the user's desktop or @CLIPBOARD meaning that the
  user pasted information into the text entry area.  For example:
     foo.tgz <= gui15.tar.gz
     bar.txt <= @CLIPBOARD

--
  Remaining arguments are treated as file names, even if they
  start with a -.

file
  Uploaded file will be saved in this file name within your
  tool session. If file is @@ then the file is given the same
  name it had before it was uploaded. If no file arguments
  are included, the default is "@@", meaning upload a single
  file and use the name it had on the desktop.
```

You can use this command to transfer one or more files from your desktop to your tool session via a web browser. This command causes a web page to pop up prompting you for various files on your desktop. Choose one or more files and submit the form. The files will be uploaded to your tool session and saved in the file names specified on the command line.

This command returns a list of names for files actually uploaded.

Export File

Overview

You can use this command to transfer one or more files from your tool session to your desktop via a web browser. A separate web browser page is opened for each file. **You must have popups enabled for this to work properly.**

Implementation

This script can be implemented as a background process in a non-rappture tool. This allow the user to continue to use the tool while the file downloads to their machine in the background. Typically a pipe is used to run this script off the main process of the tool.

Help text

```
USAGE: /usr/bin/exportfile [-t|--timeout secs] [-d|--delete] [-m|--mes
sage file] [-f|--format raw|html] file file...

  options:
    -h or --help
      Prints this help message.

    -t or --timeout <seconds>
      Forget about the file after this timeout.  Default is 86,400
      seconds (1 day).

    -d or --delete
      Delete the file after the timeout or when the tool is shut
      down.  Should be used only with temporary files.

     -m or --message
       File containing a fragment of HTML text that will be displayed
       above the download.  It might say "Here is your data," or
       "If you use this data, please cite this source."

     -f or --format <type>
       Choices are "raw" and "html".  Default is "raw".  The "html"
       format causes the server to rewrite links embedded within
       the HTML, so that images can be displayed and links can be
```

```
       traversed properly.

    --
      Remaining arguments are treated as file names, even if they
      start with a -.
```

You can use this command to transfer one or more files from your tool session to your desktop via a web browser. A separate web browser page is opened for each file. You must have popups enabled for this to work properly.

Large Data Paths

Overview

A directory structure for the storage of large data may be available upon request.

Communication with the specific Hub PI is required to determine the scope of resources needed for the implementation of large data sets. Hard disk space allocation will take into consideration the amount of available disk space remaining and the size of the data set that is to be placed on the hub. When approved, please submit a support ticket to have the directory created.

Shared data directory for a specific tool and its tool developers

A possible directory of "/data/tools/[toolname]" is to be set with permissions of 775, including write access for only members listed as developers of the tool (app-[toolname] group). Data stored here is intended to be used in a tool.

/data/tools should be mounted in the tool containers.

Shared data directory for a specific group and its members

A possible directory of "/data/groups/[groupname]" is set with permissions of 775, including write access for only members listed members of the group. Data stored here is intended to be maintained by group members and will be available for use in one or more tools.

/data/groups should be mounted in the tool containers.

Jupyter Notebooks

Jupyter Notebooks

Publishing Jupyter Notebooks

The Jupyter tool is a useful place to develop code and analyses in a notebook format. Hub users can easily share their notebooks with other users by *publishing* their notebooks as tools. A published Jupyter notebook enables other users to interact with the notebook, stepping through its cells and even changing them. But, when users run your published notebook, any changes they have made to it will not persist.

This set of instructions takes you through publishing a hub tool based on your existing Jupyter notebook. Here, we'll assume that the short name for your tool is *toolname* and that you are a registered, logged-in user. To develop the notebook tool, all you need is access to Jupyter. You'll navigate between your Jupyter home directory, the Jupyter terminal, and your tool's status page from the Tool Pipeline (The Tool Pipeline is typically found at yourhub.org/tools/pipeline but this may vary by Hub).

Jupyter version

When developing Jupyter notebooks or Jupyter based tools, you should use the most recent version of Jupyter deployed on your Hub.

To deploy a Jupyter notebook:

1. CREATE THE TOOL

To create the tool for your Jupyter notebook, navigate to Tools and click "Create a New Tool" on the upper left. Fill in the Create Tool page that the system displays:

1. Give your tool a brief name (no spaces or hyphens), a full title, and the at-a-glance description.
2. Select the repository hosting option. If you select the external hosting option be sure to supply the appropriate URL.
3. Select "Deploy as Jupyter notebook", and add your username in the development team.
4. The Access section enables you to restrict tool access to a specific hub Group, if you wish.
5. For other fields, you may accept the defaults, and submit.
6. Finally, flip the tool status to Registered, and click Apply Change.

2. REGISTER AS A Debian10 TOOL

For some hubs, you will need to submit a Hub ticket, indicating the short name of your tool, and

asking that it be registered as a Debian10 tool. This will ensure that the new tool uses current packages and kernels. Ask your Hub administrator if this applies to you.

3. CHECK OUT THE TOOL REPO

Your notebook tool's code repo should now have status Created and be ready to use. To do so, we must first check out the repo.

Open the Jupyter tool, navigate to your home notebooks directory, and open a terminal by selecting New, and then Terminal. Using the terminal, check out the newly created tool repo locally. In the section below, *toolname* is the brief name you gave your tool on the Create Tool page.

Using Subversion (svn)

To use a HUB hosted subversion repository, specify this command:

```
svn checkout https://yourhub.org/tools/toolname/svn/trunk toolname
```

Using Git

To use a HUB hosted git repostitory, specify this command:

```
git clone https://yourhub.org/tools/toolname/git/toolname toolname
```

To use an externally hosted git repository, specify this command:

```
git clone https://yourURL toolname
```

4. ADD NOTEBOOK CODE

It's now time to add the code that will run for your notebook. Back in the Jupyter tool file listing, you should see the *toolname* directory under your home notebooks directory. Into that directory, copy a working notebook (or develop one in place).

You can configure your notebook to access additional Python packages by loading an alternate kernel in the Jupyter notebook UI. To do so, consult the Kernel dropdown in the Jupyter interface. Different kernels may be available now on your Hub with additional packages. File a ticket or get in touch to let us know what packages you need.

You may need additional data files or code to run the notebook. The Hubzero team recommends putting the main notebook in the top level tool directory. Other files your notebook

needs (say, pythonfile.py) can be organized in subdirectories such as bin/. Then, you can load any Python files in your notebook as if they were modules. Your notebook will load the Python source data/pythonfile.py this way:

```
import bin.pythonfile
```

5. EDIT INVOKE SCRIPT

Finally, to tell the hub how to launch the notebook, you need to edit the invoke script that was automatically created at tool creation time. The invoke shell script is found in the *toolname*/middleware/ directory. To edit it, double-click on the invoke script in the Jupyter file listing, and the editor will launch.

In the invoke script you specify the filename of your Jupyter notebook, the version of Anaconda to use, and other parameters. Here we suppose that your notebook is called *your-jupyter-notebook-name.ipynb*.

For a Jupyter notebook using anaconda-X, your script will look like this :

```
#!/bin/sh

/usr/bin/invoke_app "$@" -t toolname \
                         -C "start_jupyter -T @tool your-jupyter-
notebook-name.ipynb" \
                         -r none \
                         -w headless \
                         -u anaconda-X
```

If your notebook needs additional modules, list them as -u module pairs. Be sure to add line continuation as needed (\).

For details on invoke script command line options, refer to the Hubzero invoke documentation.

6. TESTING

Next, you'll test that your working notebook starts properly as a Hub tool. When the notebook passes testing, you are ready to proceed.

7. COMMIT CHANGES

Once you have saved your invoke script and your notebook, check them in to the repository management software. You'll use subversion or git.

Using Subversion (svn)

From a Jupyter terminal, navigate to your tool's directory (get there as we did in step 3. above). First, add the notebook to svn (similarly, add any other needed files, using "svn add"):

```
svn add your-jupyter-notebook-name.ipynb
```

then, once all files have been added in this way, commit the changes:

```
svn commit -m "commit message"
```

The commit message should briefly indicate why the commit is being done or what the commit accomplishes. Commit messages serve as documentation for your work.

Using Git

From a Jupyter terminal, navigate to your tool's directory (get there as we did in step 3. above). First, add the notebook to git (similarly, add any other needed files, using "git add"):

```
git add your-jupyter-notebook-name.ipynb
```

then, once all files have been added in this way, commit the changes:

```
git commit -m "commit message"
```

The commit message should briefly indicate why the commit is being done or what the commit accomplishes. Commit messages serve as documentation for your work.

Once all files have added and committed the changes need to be pushed to the repository accessed by yourhub.

```
git push
```

To alert the administrator that your tool is ready for installation, you can now visit your tool's status page, either from the Tool Pipeline, or specifying a URL like this:

`https://yourhub.org/tools/toolname/status`

Here, click the link that reads, "My code is committed, working, and ready to be installed." If you have special instructions, caveats, compile steps, or other dependencies for your installation, enter them in the available text box now. The tool administrators will be alerted about your tool status and perform the installation along with any required steps you describe.

8. INSTALL

It's time to install the tool source. This action will depend on your access privileges; you may need the help of an administrator. On the hub, visit your tool's status page, either from the Tool Pipeline, or specifying a URL like this:

`https://yourhub.org/tools/toolname/status`

Here you can click the Install button and then on success message, flip the status to Installed and apply the change.

If the Install button is not available to you, this task will be executed by an administrator. You will receive a status email when it is complete.

9. TEST AND PUBLISH

To test your tool, go to the hub's Tool Pipeline and select your tool's link, or specify the tool URL directly:

`https://yourhub.org/tools/toolname`

In the status page, click the button to test run the tool. If the tool does not display or otherwise fails your test, there is still work to do. Revisit your development steps, starting with the TEST section above.

If the notebook test is successful, and it displays and functions as expected, you are almost done! Return to the tool status page. There you can indicate to administrators that you Approve the tool for publication.

You will receive a status change email when the tool has transitioned to Published. When you receive word that your tool is Published, you should verify again that it works as expected.

That should do it--your Jupyter notebook is now a published tool available to other Hub users. If

you have questions, concerns, or run into a snag, please email the Hub administrator. Include any error messages you see, and we'll give you a hand.

10. MAKING CHANGES

To make changes to a published notebook, you must only revisit some of the steps outlined above.

To make edits to the tool:

- Change your notebook code as necessary, revisiting the TEST and COMMIT CHANGES steps above when complete.
- INSTALL your changed code as above
- TEST AND PUBLISH the notebook tool as above

Each time you make changes, be sure to test the notebook and confirm that it works properly.

Jupyter Tool Deployment Styles

Jupyter tool deployment styles

How do you want your Jupyter Notebook-based tool to behave when a user runs it? You have several choices, which we describe here:

1. Tool or App Mode style--code cells are hidden, UI widgets are visible
2. Notebook style--all code cells are shown. This is the default.

If you're working out how to develop and deploy the notebook, please refer to the other articles in this series. Continue reading to choose and set the tool display style.

Setting App Mode

You can achieve the dashboard-style effects by just selecting the App Mode in the Jupyter menu bar. Toggle back to edit your notebook using the Edit App button.

This screenshot shows the Jupyter tool menu bar, with its *Appmode* button

Deploying as App Mode

Then, to deploy the tool in App Mode, specify an -A in your invoke script, as follows:

```
/usr/bin/invoke_app "$@" -t toolname \
                         -C "start_jupyter -A -T @tool MyNotebook.ipyn
b" \
                   -u anaconda-X \
                         -w headless \
                         -r none
```

The tool user will still be able to toggle the tool to show its code cells. To suppress that behavior, add a -t to the start_jupyter call.

```
/usr/bin/invoke_app "$@" -t toolname \
                         -C "start_jupyter -A -t -T @tool MyNotebook.i
pynb" \
                   -u anaconda-X \
                         -w headless \
                         -r none
```

Using Python packages from Jupyter Notebooks

Using Python packages from Jupyter Notebooks

The Jupyter tool is a useful place to develop Python, R, or Octave code and analyses in a notebook style. Hub users can easily share their notebooks with other users by *publishing* notebooks as tools. A published Jupyter notebook enables other users to interact with the notebook, stepping through its cells and even changing them. When users run your published notebook, any changes they make to it will not persist.

Here we assume you are running: anaconda-7; debian10 container.

Python packages

Python has been extended to work with hundreds of specialized packages. For example, see the Anaconda package repo. A number of scientific Python packages are installed and accessible on the hub.

The hub uses Jupyter kernels to safely load needed Python packages. You can select a Jupyter kernel to set paths to a self-contained installation of specified packages, making them available in your notebook. This page will show you how to set access to Python packages from Jupyter Notebooks.

Note that we must install packages on the hub to make them available as a kernel. Submit a ticket to request new packages or a new kernel.

Selecting a kernel

New notebook

To select the kernel for a new notebook, start a Jupyter tool. In the upper right, select 'New', then the kernel you want from the kernel menu. You can now import and use the kernel's packages in your notebook.

Be sure to save the notebook after changing the kernel.

Existing notebook

If you need to change the kernel for an existing notebook, first open the notebook in a Jupyter tool.

1. If the notebook is already running, you must first shut it down by selecting Kernel: Shutdown from the menus.
2. Then, you can select the kernel of your choice using the menu Kernel: Change Kernel: *somekernel*. After you have made the selection, check the displayed kernel name on the upper right of the notebook. It should match what you just selected.

3. Finally, save the notebook, and your kernel choice will be saved along with it.

Kernel availability

How do we know what packages are available in what kernels?

1. Check the conda env specification file associated with the kernel.
2. Run conda commands to interrogate the packages. Read the next section for further information.

Using conda to list installed packages

The kernels we have created to support different sets of Python packages are based on conda environments ("envs"). You can interrogate these conda envs to list the packages a given kernel supports. This is general to Anaconda package manager (more is available here). Below are a few tips.

Note that creating a conda env is an administrator action. If you need a new env or additional packages, enter a ticket to request them.

Example

The kernel named modgrnld-python3 contains the following packages and their dependencies:

- matplotlib
- rasterio
- georaster
- hublib
- python 3.7
- netCDF4
- numpy
- pyproj
- scipy

What envs are available?

To access a conda env, first start a Workspace10 tool. On the command line, type the following command to set the anaconda installation in your path:

```
use anaconda-7
```

Now, to show the names of available envs:

```
conda info --envs
```

Note that we may not have created kernels for all the available envs.

What packages are in this env?

If you have an env enabled currently, to list packages there, type:

```
conda list
```

Or for an arbitrary env, someenv:

```
conda list -n <someenv>
```

Export current conda env

To export a list of the packages and versions installed in the env to a text-based .yml file:

```
conda activate <envname>
conda env export > <filename>.yml
```

Testing Jupyter-based tools

Testing Jupyter-based tools

The proxied Jupyter tool is a useful place to develop code and analyses in a notebook style. Hub users can easily share their notebooks with other users by publishing notebooks as tools.

These instructions take you through *testing* the deployment of a Jupyter notebook based tool. Here, we'll assume that the short name for your tool is *toolname*. To test the notebook tool, it's handy to use the hub's Workspace tool, since this allows you to fully test the deployment in the context of the hub.

1. Create the tool

Once you have your notebook working to your satisfaction on the hub, you next create the tool to house it, and edit the invoke script. Once this is done, it is time to test.

NOTE that any new development, and any updates to existing tools, should make use of Debian10 containers. Develop these using the appropriate Jupyter tool and Workspace10.

2. Test invoke the tool

From the Workspace10 tool, or the Jupyter terminal tool, navigate to the directory where your tool's repo is located. For a tool you've called *toolname* and stored in the apps subdirectory of your home, this will be something like:

```
~/apps/toolname
```

Now, navigate to your tool's middleware directory, and call the invoke script for your tool, by typing:

```
cd middleware
./invoke
```

Check the command line output to determine the success of your tool invoke call. Errors will display here if a problem is encountered. Use these to aid in your troubleshooting. If you see errors, you will need to revisit the tool sources, retesting to see if your fixes have worked, before going on with this procedure. Note that you may also see warnings displayed, as well as informational output. Neither warnings nor information indicate issues that need to be fixed.

A successful invoke script call will output in part:

```
The Jupyter notebook is running
```

...followed by a lengthy URL that reads, in part:

```
https://proxy.yourhub.org/weber/
```

Congratulations, your notebook-based tool is now running!

3. Check the running notebook

In the Workspace10 tool's command line output, note the informational output, denoted by lines starting in "I", and warning output, denoted by lines starting in "W", that is also displayed. These messages can be ignored safely; refer to the figure below.

Within the Workspace10 tool, you can now start a browser and paste into it the URL of the running Jupyter kernel. This test is not possible from outside your development environment, in order to protect your unreleased tool.

Copy the running kernel's URL

First, locate and highlight the kernel URL provided in the output from the invoke script.

Start the Workspace10 browser

Start the Firefox browser from the Workspace tool's menu. To do so, click the black button at the bottom left of the Workspace10 and access the Firefox menu item.

Navigate to the running kernel

A browser window will display, running inside your Workspace tool session on your hub. Finally, click (mouse wheel or both mouse buttons) to paste the running Jupyter kernel's URL into the browser navigation bar, as shown in this graphic

Now, the Workspace browser will display the running tool. This is the notebook running as a tool, and should be a good indication of how the tool will run once it is deployed.

One important error type you should watch for is the URL timeout. If there are URLs that your notebook needs access to in order to run, they will likely time out during this test. Collect any such URLs and include them in a support ticket on your hub to your hub administrators. The administrator will need to approve (whitelist) these URLs in order for them to be accessible to your notebook once it is a deployed tool. Be sure to explain this in your ticket, and clearly identify the tool name and the reason each URL is needed.

To stop testing, close the browser session running inside your Workspace10, then type "control-c" in the terminal where you called the invoke script. Once the prompt returns, your notebook kernel has stopped.

You can make any adjustments needed to the underlying code before you flag your tool as Uploaded and continue with the deployment process.

Environment Variables

Environment variables

A number of environment variables are available in a hub tool session. A few are discussed here. A full list can be viewed by running the *env* command from a terminal in the Workspace tool, the noVNC Desktop, or the Terminal.

Remember: tools are invoked by the current user's account and all permissions are set accordingly. Therefore, a tool can save files to a user's home directory, *because the tool runs as that user.*

SESSION

This variable stores the session ID or session number of the currently running tool. It's the ID of the session you are currently using. You can assume it's unique.

Notice that your current SESSION number is visible in your browser URL when you are running a tool. Here's an example:

https://proxy.yourhub.org/weber/20064/...

USER

The USER variable stores the username of the user running the current tool.

SESSIONDIR

The SESSIONDIR variable stores the current session directory of the current tool run. A separate directory is created for each new tool session. Since it is created in the current user's home directory, the tool can write to this directory.

This is the recommended location for writing temporary files generated by your tool. Be mindful of the user's quota limits when writing temporary results. It's wise to delete these once the run is complete.

Session directories take the form:

```
/home/HUBNAME/USER/data/sessions/SESSION
```

RESULTSDIR

The RESULTSDIR variable stores the results directory for the current tool run. It is located in the user's home directory. This is a good place to place simulation results and output for the user to access later.

Be mindful of the user's quota limits when writing results.

Results directories take the form:

```
/home/HUBNAME/USER/data/results/SESSION
```

PWD

This variable stores the present working directory.

HOME

This variable stores the full path to the user's home directory. This can be useful if a tool provides an option to save the user's current work. Tool developers should create a directory for the tool to save files relative to HOME, to prevent cluttering the user's home directory. For example, "$HOME/data/*toolname*".

Note that unlike the RESULTSDIR and SESSIONDIR described above, "$HOME/data/*toolname*" will not be created for each tool run.

Home directories take the form:

```
/home/HUBNAME/USER
```

Accessing environment variables on a hub

For a full list of environment variables, type this in a Workspace terminal:

```
printenv
```

To view the value of an environment variable from the Workspace terminal (sh or bash shell, e.g.):

```
echo $SESSION
```

From Jupyter's Python kernel, for example, use the shell escape:

```
!echo $SESSION
```

Invoke scripts for Jupyter notebooks

Invoke scripts for Jupyter notebooks

The hub tool invoke script is located in the tool's middleware/ subdirectory. When you first create a tool, the basic invoke script provided must be edited to work with Jupyter notebook tools.

This writeup shows you how to create Jupyter tools with three different appearances: notebook, App, and Tool mode.

invoke_app and start_jupyter

To deploy a Jupyter notebook as a tool on your hub, you call the invoke_app executable, which in turn calls start_jupyter. Each have their own arguments:

arguments for start_jupyter

```
  -h, --help  show this help message and exit.
  -d          Show debug (verbose) output.
  -t          Run as a Tool with no notebook controls.
  -c          Copy instead of link notebook files.
  -A          Run in AppMode.
  -T dir      Search for notebook starting in dir.
  --themes    Enable notebook themes

arguments for invoke_app

  -t Tool name
  -C command to execute
  -r Rappture version to use (normally specify none for notebook tools
)
  -w headless
  -u environment package (repeat as necessary)
```

invoke_app: starting point

The basic invoke script for Jupyter notebooks looks like this:

```
/usr/bin/invoke_app "$@" -t TOOLNAME \
                  -C "start_jupyter -T @tool APP.ipynb" \
                  -r none \
                         -w headless \
```

```
                -u anaconda-X
```

Invoking a Jupyter tool this way gives a notebook with all its code cells displayed to the user.

Where:

- TOOLNAME is the short name of the tool
- APP is the name of the main notebook that runs the tool
- anaconda-X is the current anaconda installation

start_jupyter arguments

Control the way the notebook appears when run as a tool, using the arguments passed to the start_jupyter executable.

You can run a Jupyter tool in three ways:

- notebook mode, in which all code cells are displayed to the user (shown above)
- app mode, in which code cells are initially hidden but can be displayed
- tool mode, in which code cells are hidden and cannot be displayed

for App Mode

For a notebook tool that hides its code cells and shows only the UI and markdown elements on initial run, add the -A argument in the start_jupyter call:

```
/usr/bin/invoke_app "$@" -t TOOLNAME \
                -C "start_jupyter -A -T @tool APP.ipynb" \
                -u anaconda-X \
                       -w headless \
                -r none
```

The tool user can toggle the tool's "Edit App" button to show the underlying code cells, making this a great teaching/demo option.

NOTE that this differs from the invoke_app -A argument.

for Tool Mode

To permanently hide code cells from the user in App Mode, specify the -A and -t arguments in

the start_jupyter call:

```
/usr/bin/invoke_app "$@" -t TOOLNAME \
                  -C "start_jupyter -A -t -T @tool APP.ipynb" \
                  -u anaconda-X \
                         -w headless \
                  -r none
```

The Edit App button will not be displayed to the tool user.

NOTE that this differs from the invoke_app -t argument.

errors

specify no rappture

```
/usr/bin/invoke_app "$@" -t TOOLNAME \
                  -C "start_jupyter -T @tool APP.ipynb" \
                  -u anaconda-X \
                         -w headless \
                  -r none
```

Error:

Running the tool's invoke script from a workspace, returns:

```
"could not find a rappture installation: RAPPTURE_PATH=,"
```

Fix:

Be sure to supply the "-r none" argument in the invoke_app call, as above. No quotation marks are needed.

Tool Repository Structure

The directory structure for tool repositories should start from the following:

```
toolname/
   bin/
      .keep
   data/
      .keep
   doc/
      .keep
   examples/
      .keep
   middleware/
      invoke
   rappture/
      .keep
   simtool/
      TOOLNAME.ipynb
   src/
      Makefile
```

The .keep files are present to force git to track initially empty directories. Should the directories become populated the .keep file can be removed. The simtool directory should be present only if the tool was registered with the publication option = Sim2L. The rappture directory is only required for tools actually using Rappture. The src directory should contain any source code that needs to be compiled and a Makefile to direct building of executables. The Makefile must have the install, clean, and distclean targets. The bin directory should contain any executables created by issuing the make install command in the src directory. The bin directory is automatically added to the PATH environment variable used to search for executable files. It is common practice to put Jupyter notebook files in either the top level or bin directories. It is also permissible to create additional directories as deemed necessary.

Invoke script template

All tools require that the executable file middleware/invoke be present. The specifics of the invoke file depend on the tool publication classification. Templates for the three categories of tools are shown here. For tools where the repository is maintained on the HUB known values such as tool name are substituted in the template to provide an initial invoke file.

Sim2L

```
#!/bin/sh

#
# Sim2L
#
/usr/bin/invoke_app "$@" -t @TOOLNAME@ \
                         -C "start_jupyter -T @tool -t @TOOLNAME@Examp
le.ipynb" \
                         -u anaconda-X \
                         -r none \
                         -w headless
```

Jupyter Notebook

```
#!/bin/sh

#
# jupyter tool
#
/usr/bin/invoke_app "$@" -t @TOOLNAME@ \
                         -C "start_jupyter -T @tool -t @TOOLNAME@.ipyn
b" \
                         -u anaconda-X \
                         -r none \
                         -w headless
```

Rappture/Linux GUI

```
#!/bin/sh

#
# standard tool
#
/usr/bin/invoke_app "$@" -t @TOOLNAME@ \
                         -C rappture
```