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, optionally, your tool's status page from the Tool Pipeline (The Tool Pipeline is typically found at yourhub.org/resources/add-contribution/tool-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 "Deploy as Jupyter notebook", and add your username in the development team.
3. The Access section enables you to restrict tool access to a specific hub Group, if you wish.
4. For other fields, you may accept the defaults, and submit.
5. Finally, flip the tool status to Registered, and click Apply Change.

1B. Register as a Debian10 tool

The final step of tool registration: 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.
2. ADD THE TOOL REPO

Next, we may need to add the repository containing your Jupyter notebook tool code. If the Add Repo button is now available to you in your tool status page:

- Click the Add Repo button.
- Then, if the page shows a success message, flip the status to Created and click Apply Change.

Otherwise, an administrator must add the repo for you; you should wait for a status email indicating the repo has been created.

3. CHECK OUT THE TOOL REPO

Now the new notebook tool's code repo is Created and ready to use. To do so, we must first check out the repo.

**Using Subversion (svn)**

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 using this command (toolname is the brief name you gave your tool on the Create Tool page):

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

**TODO: Using Git**

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 data/. 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:

```python
import data.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.

A basic script should look like the following for a Jupyter notebook with Python 3. Use your own notebook's name in place of 'your-jupyter-notebook-name.ipynb' :

```bash
#!/bin/sh
/usr/bin/invoke_app "$@" -C "start_jupyter -T @tool your-jupyter-notebook-name.ipynb" -r none -u anaconda-6
```

If your notebook needs additional modules, list them at the end after the -u.

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.

For subversion: 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
```
Subversion will now prompt you for a commit message. Type in a suitable initial commit message ("initial commit of toolname" will do) and press ctrl-X to exit.

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.

TODO: Describe how to check code into git

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.

TODO Edit: Final step for jupyter6deb10 tools: To publish the tool you'll need to let us know the tool name so we can update the docker image mapping so the tool loads in the deb10 env. To do this, enter a hub ticket letting us know tool name and asking for an update of the docker image mapping.

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. If special instructions or compilation steps need to be performed for your tool, indicate that here.

Depending on your access privileges on the Hub, you may be able to set the tool as Published. If the link is available to you, click Publish, and then on display of the green success message, flip the status to Published and apply the change.

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.

If you are planning to edit an existing Hub tool, please convert it into a Debian10 tool first (See 1B. above). To do so, submit a ticket on your hub, indicating the short name of your tool, and asking that it be registered as a Debian10 tool. This will ensure that the tool has access to current packages and kernels. Hub staff will let you know when the change has been made.

Next, 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

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
3. Autorun style--shows and runs code cells on load

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.

Using Jupyter on Deb10 version

If you are running the newer, Jupyter on Deb 10 tool (alias "jupyter6deb10") to develop notebooks, 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.

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 "$@" -C "start_jupyter -A -T @tool MyNotebook.ipynb" -t mytoolname -u anaconda-6 -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)

For the older Jupyter Notebooks tool, read on. Note that we encourage you to switch to use jupyter6deb10 if you are undertaking any substantial changes to your tool.
Using Jupyter (debian7 version)

For the older Jupyter Notebooks tool (alias "jupyter"), read on. Note that we encourage you to
switch to use jupyter6deb10 if you are undertaking any substantial changes to your tool or if you
encounter problems with changing the notebook style.

Dashboard ("Tool" or AppMode) style

To deploy a notebook as "tool" or dashboard style, the code will be hidden, but the widgets,
plots, and markdown will be displayed.

Two things must be done to deploy your tool in this style. From the Workspace tool's command
line, do the following:

First, mark the tool mode on your notebook:

```
jupyter_tool.py -t notebook-name.ipynb
```

Then, set your notebook as trusted:

```
jupyter trust notebook-name.ipynb
```

Finally, check the code in to subversion (svn).

Note that you cannot make changes to a notebook once it's marked as a trusted tool. If you
must make changes to a notebook that's deployed as a Dashboard, first follow the instructions
in Notebook style below, then make your changes, and finally re-mark the notebook to use tool
mode and trusted mode as above. Then you can check in the Dashboard-style tool's changes to
svn!

Notebook style

To deploy a notebook as notebook style, all code in the notebook will be displayed. This is the
default for Jupyter notebook based tools, so if it's a new tool, there's nothing special you must
do. However, if you're switching a notebook from tool style to notebook style, you must do these
things:

First, mark the tool mode to notebook:

```
jupyter_tool.py -n notebook-name.ipynb
```
Next, unsign your notebook:

```
jupyter_unsign.py notebook-name.ipynb
```

Open your notebook and make a change (if no code change is needed, just add a line of whitespace). Save that.

Lastly, check the source in to subversion (svn) and continue with the tool deploy.

Note that just changing the tool mode or signing/unsigning is insufficient. You must change the notebook itself then check it in in order to change its behavior.

**Autorun style**

For autorun, run with -a to autorun all cells on load of the notebook.
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-6; 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 the Jupyter Deb 10 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 the Jupyter Deb 10 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.
Finally, save the notebook, and your kernel choice will save 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 commands to set the anaconda installation in your path:

```
source /etc/environment
use anaconda-6
```

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 deploy 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 deploy 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 Jupyter on Debian 10 (jupyter6deb10) and Workspace10.

2. Test invoke the tool

From the Workspace 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 Workspace 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 Workspace 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 Workspace browser

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

**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:

```
env
```

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

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
- d show debug (verbose) output
- t run as a Tool with no notebook controls
- A run in AppMode.

arguments for invoke_app
- C command to execute
- r Rappture version to use (normally specify none for notebook tools)
- u environment package(s) to use
- t Tool name

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
- u anaconda-6
```

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-6 is the current anaconda installation
Notice that the script uses the -t, -C, -r, and -u invoke_app arguments.

**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-6
 -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 invoke_app.

**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-6
 -r none
```

The Edit App button will not be displayed to the tool user.  
NOTE that this differs from the invoke_app -t argument for invoke_app.
errors

specify no rappture

/usr/bin/invoke_app "$@" -t TOOLNAME
-C "start_jupyter -T @tool APP.ipynb"
-u anaconda-6
-r none

Error:

Running the tool's invoke script from the deb10 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.