Running Jupyter Notebooks on the Trantor cluster

1. Introduction

Jupyter notebooks are implemented as Web applications. The server implements the "notebook" abstraction and dispatches the code contained in the notebook to an interpreter, for its actual execution. In Jupyter terminology, such interpreters are called "*kernels*" (e.g. the kernel used to execute Python code is part of the "IPython" project: https://ipython.org/).

While it is common to run a single notebook server locally on the user's machine, the system described in this document allows the execution of several servers per user on the cluster nodes, thus allowing the execution of multiple calculations on high-performance machines in an interactive (or semi-interactive) fashion.

This system is a customized version of the JupyterHub application (https://jupyter.org/hub). In the rest of the document we will refer to it as "JupyterHub@Trantor".

2. Software requirements

To use JupyterHub@Trantor you just need a modern Web browser. Installing a Python / R distribution on your local machine is not required, since notebooks will be executed on the cluster.

3. Preliminary steps

To access JupyterHub@Trantor, you need a personal account on the Trantor cluster. To request a Trantor account, please send an email to hpcstaff@sns.it.

You also need to properly configure your bash shell to work with **conda**:

- Connect via ssh to a head node of the Trantor cluster (e.g. trantor01.sns.it).
- From your home directory, execute the following command:

  ```
  /cluster/shared/software/miniconda3/bin/conda init
  ```

- Finally, logout and login again, so that the changes can take effect.

After logging in, you'll see the prefix "(base)" at the left of the shell prompt (e.g. (base) [user@hostname ~]$). The name (or the path) inside the parentheses shows the currently active conda environment.

A brief introduction to the Conda utility is available on the HPC website (https://hpccenter.sns.it/page/wiki/). For further details, refer to the official Conda documentation (https://docs.conda.io).

Note
The Trantor cluster can be accessed only from the internal SNS network. If you are working off-site, you need to establish a VPN connection to the SNS network. Instructions on how to download and configure the VPN client can be found at the following web page: https://ict.sns.it/en/service/access/vpn

4. Logging into the system

To access JupyterHub@Trantor, visit the following web page (from the internal SNS network or via a VPN connection to the SNS network) : https://jupyter.sns.it

Here, you can log into the system by providing the credentials of your Trantor account. Please note that you will be prompted to re-enter your credentials every 7 days.

5. Notebook servers management

The home page (see Figure 1) reports several information about the notebook servers and provides the tools to manage them.

To create a new server, insert a name for the server in the form at the top of the home page and press the “Create Server” button (see Figure 1). Server names can contain only letters, digits and the underscore character. Furthermore, they must start with a letter. Each user can run up to 10 notebook servers at the same time.

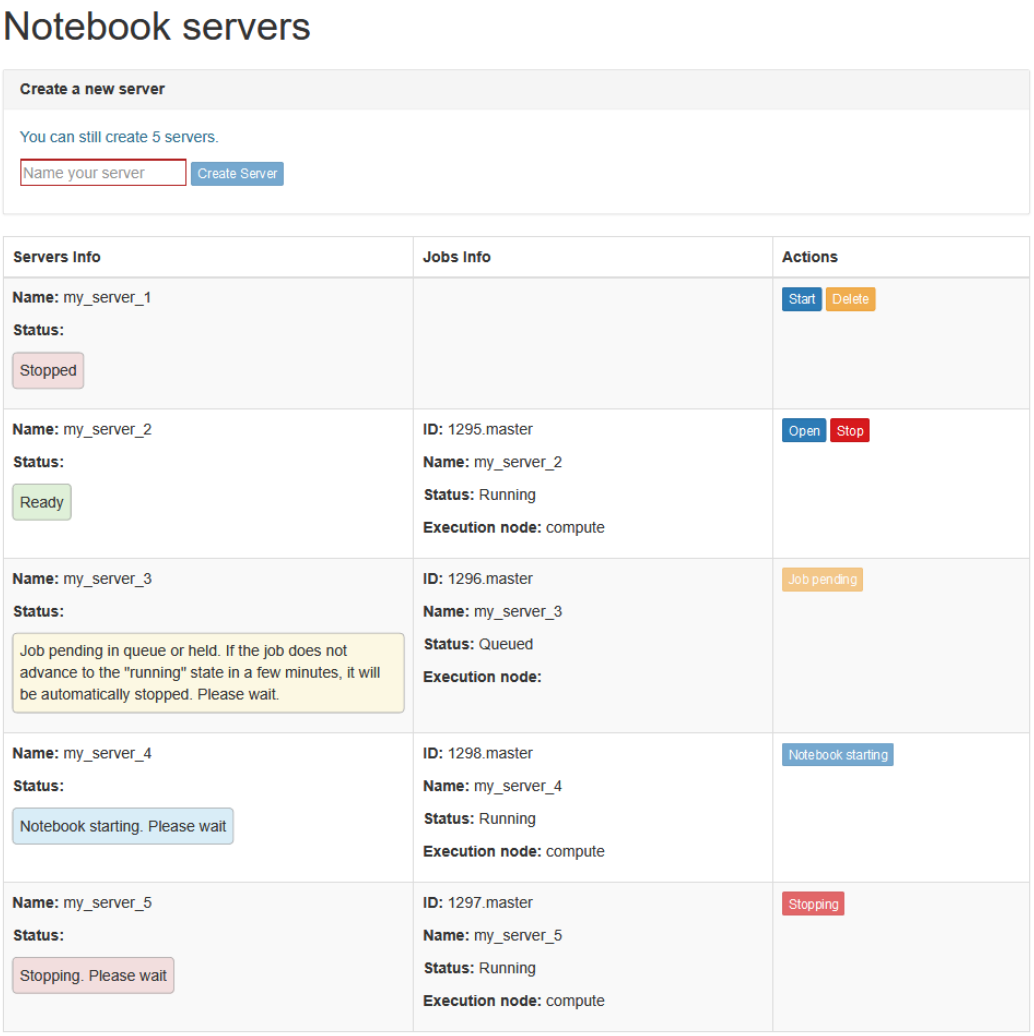


Figure 1: The home page

Once created, you will be redirected to the “Job” submission form, from which you can specify the resources to be reserved for the Job[1] (see Figure 2).

[1] You can find a brief introduction to job scheduling systems at the following Web page: https://hpccenter.sns.it/page/wiki/pages/Submitting_Inspecting_and_Cancelling_PBS_Jobs.html.

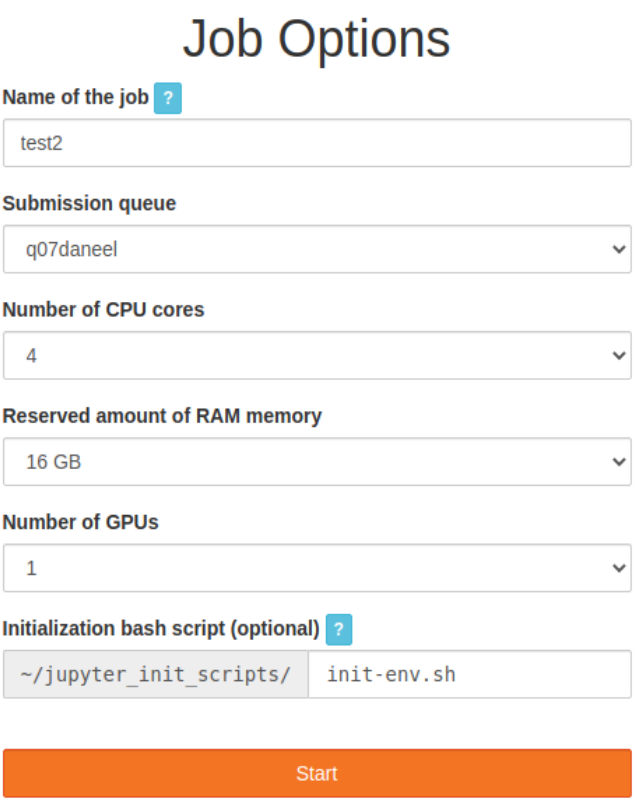


Figure 2: Job submission form

In particular, you will have to enter:

- A name for the job. By default the Job is named as the associated server, but you can rename it, if you wish.
- The submission queue.
- The number of CPU cores to be reserved.
- The amount of RAM memory to be reserved.
- The number of GPUs to be reserved.
- Optionally, you can also provide the name of a **bash script to be executed before starting the notebook** server, e.g. with the purpose of initializing the environment. The script **must be stored in a folder named "jupyter_init_scripts" within your home** directory.

You can then confirm the submission by pressing the "Start" button.

Note
The requested computational resources are allocated in a single node. Currently, JupyterHub@Trantor does not allow to reserve multiple compute nodes for a single Job.

If the notebook server begins its execution in a short time, the browser will be automatically redirected towards the JupyterLab application (see section 6). Otherwise it will return to the home page, from which you can monitor the starting process.

As shown in Figure 1, the main area of the home page consists in a table listing the notebook servers, a summary of their status and the buttons to manage them.

It is important to note that this table lists not only the active servers, but also the ones that already finished their execution. Such “stopped servers” are just placeholders: they don’t consume computational resources (there are no corresponding jobs), and their only purpose is to provide a shortcut by means of which a new server instance with the same name and requested resources can be started at a later time.

The first column of the table reports the name and the status of each server. The possible server states are the following:

- **Stopped** : The server is not currently active and it is not associated with any Job. This is just a “shortcut” to start a new server instance with the same name and requested resources. You can delete this “shortcut” by means of the “Delete” button on the third column.
- **Job pending** : The Job is pending in a queue or it was put in the “*Held*” state[2]. If the job does not advance to the "*Running*" state in a few minutes, it will be automatically stopped.
- **Notebook starting** : The job is being executed but the notebook server is still starting up.
- **Notebook ready** : The notebook server is up and running. Press the “Open” button on the third column to open the notebook. Press the “Stop” button to stop the server and the related Job.
- **Stopping** : The notebook server and the related Job are being stopped.
- **Checking** : Jupyterhub is checking the state of the notebook server.
- **Removing** : The job is terminated and the related entry is being removed from Jupyterhub.
- **Unexpected** : The server is in an inconsistent state. Please report the problem to hpcstaff@sns.it .

The second column of the table summarize the main information about the Job associated to the notebook server: Job ID, name, status (as reported by the “*qstat*” utility[3]) and the compute node on which the Job is being executed. Please provide these information when you contact the technical support.

---

[2] Please refer to the PBS Professional User's Guide for details.
[3] You can find a brief introduction to PBS at the following Web page:
https://hpccenter.sns.it/page/wiki/pages/Submitting_Inspecting_and_Cancelling_PBS_Jobs.html.

6. Jupyterlab usage

JupyterLab is the new web-based user interface for working with Jupyter notebooks. The complete JupyterLab user guide can be found at the following Web page: https://jupyterlab.readthedocs.io/ .

Please note that some JupyterLab features are not available in JupyterHub@Trantor, e.g. the ability to add extensions.

Files management and scratch areas

You can use the built-in file manager to browse the content of your home folder, open files in JupyterLab, and even download / upload files to / from your personal computer.

> **Note**
> The uploading functionality provided by JupyterLab is suitable only for small files (up to a few megabytes). In the case of large data files, please use tools like rsync, scp or sftp.

It is important to note that, while the file browser does not allow exit from the home folder, you can access other file system locations by using the built-in terminal (see Figure 3).

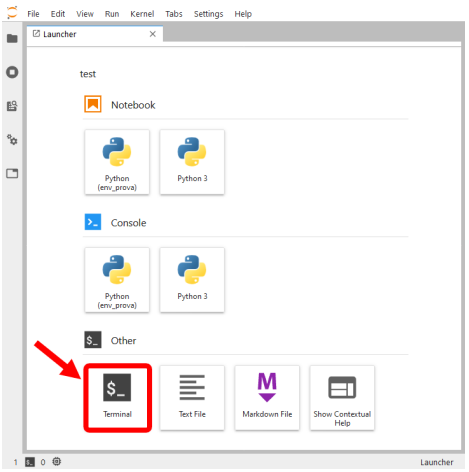


Figure 3: Opening a terminal from within JupyterLab

Home directories are mounted on Trantor nodes as NFS file systems. While this solution has the advantage of giving access to the same home folder from every node of the cluster, it is a major bottleneck for I/O operations. For this reason, Jobs **must** make use of local **scratch areas** when performing any significant I/O. In particular, every user has a scratch space on every computing node, under /scratch/$USER. Whenever possible, this storage area is allocated on the local hard drives of the compute nodes, thus providing a higher bandwidth and a lower latency than NFS file systems.

Here are some important notes on the use of scratch areas:

- **You must perform any significant I/O operation on the scratch area**. **Using your home directory as a scratch space is forbidden**.

- By the end of the computation, remember to copy any relevant file from the scratch folder back to your home. In fact, local scratch areas are not backed up and can be erased by the technical staff for maintenance purposes. Also remember to clean up your scratch folder by deleting the files you don't need anymore.

You can find further details on scratch areas at the following web page:
https://hpccenter.sns.it/page/wiki/pages/Submitting_Inspecting_and_Cancelling_PBS_Jobs.html#Scratch_Areas

Warning
When running multiple notebook servers, they can potentially operate on the same files. It should be noted, however, that **manipulating a file from more than one server is extremely dangerous, since it may lead to data loss or corruption!**
Our suggestion is to organize your work in such a way that each server operates on a different folder, so as to avoid opening the same files from multiple servers.

Long-running computations in Jupyter Notebooks

One of the advantages of using JupyterHub@Trantor is the ability to embed long-running computations in Jupyter notebooks and to execute them on the cluster. However, due to technical limitations of JupyterLab, you have to take some specific precautions to avoid losing results!

As an example, consider this simple Python function that prints a number every minute for the amount of minutes given as parameter :

```
[1]: import time

[2]: def test_comp_1(n):
         for i in range(1, n+1):
             time.sleep(60)
             print(i)
```

Let's call this function passing 180 as parameter, so that it will keep running for three hours:

```
[*]: test_comp_1(180)
     1
     2
     3
```

Let's assume that, after a few minutes, we save the notebook and **close the browser**. When we open the notebook again after three hours, we could be shocked by the fact that the only results on the notebook are those already present at the time of saving (i.e. before closing the browser) !

Furthermore, if we look at the bottom-left corner of the window, the status of the kernel is reported as “Unknown” (see below) or even “No Kernel!”, meaning that the application is not aware of the current status of the kernel.

This behaviour is due to the fact that when we close the tab of the browser where (the frontend of) JupyterLab is running, the connection between the kernel and the frontend is interrupted. It follows that all the following output generated by the kernel (intended to be displayed on the notebook) will be lost. Unfortunately this is a known limitation of the current implementation of Jupyter (https://github.com/jupyter/notebook/issues/1647).

For that reason, special precautions have to be taken when writing code that may run for a not negligible amount of time.

First of all, it is important to note that the internal state of the kernel is not affected by the client disconnection and that the connection between the client and the kernel can be restored. Therefore, if we save the results in a variable we can retrieve them at a later time, even if the client disconnects during the computation. Below, a different implementation of our example function, which exploits this observation, is shown:

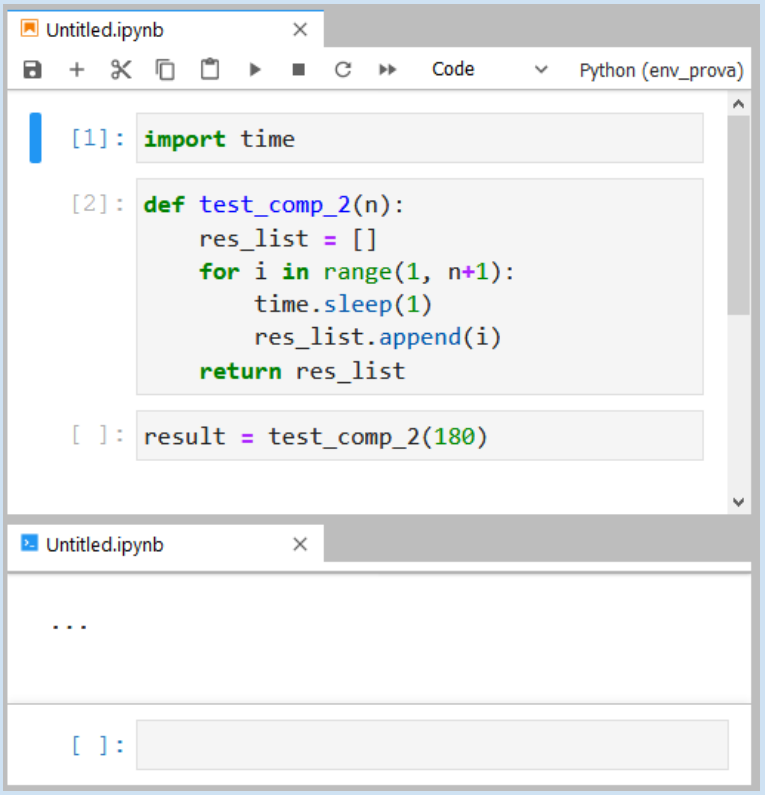
```
[1]: import time

[2]: def test_comp_2(n):
         res_list = []
         for i in range(1, n+1):
             time.sleep(60)
             res_list.append(i)
         return res_list

[*]: result = test_comp_2(180)
```

Now, if we close the browser tab (remember to save the notebook!) and we come back when the computation is over, we can retrieve the results stored in the variable (even if the client complains about not knowing the current state of the kernel):

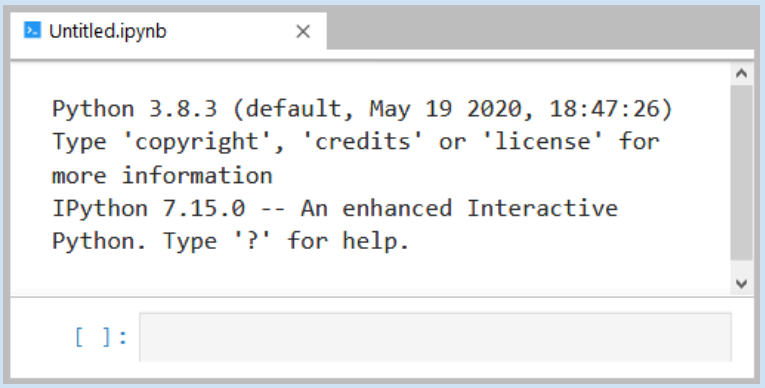
```
[ ]: result = test_comp_2(180)

[4]: result

[4]: [1,
      2,
      3,
      4,
      5,
      6,
      7,
      8,
      9,
      10,
```

Warning
If you shut down, restart or change the kernel, the entire kernel state is lost, including the results of your computation that have not been saved to file!

Note
Most of the time we can't know in advance the exact duration of a computation. So, how can we check if the computation is still running? A simple trick is to open an IPython console associated with the notebook (right click on an empty space on the notebook and select the item "*New console for notebook*" from the contextual menu):



If the content of the console is an ellipsis (a series of three dots), it means that the kernel is currently busy. When kernel will end the execution of all the previously submitted commands, the ellipsis will be replaced by a print of the Python and IPython versions:



If you try to execute a cell when the kernel is busy, the code will be appended to a queue of commands in the wait of being executed.

An even better solution is to **save both final and intermediate results to file**. By doing so, most of your work is preserved even in the case of software crashes (and some types of hardware failures). Longer is the computation, more important is to save intermediate and final results to file.

Finally, a useful tool for dealing with third-party code that prints data on screen is the "*capture*" magic command provided by IPython. By inserting the directive

```
%%capture variable
```

at the beginning of a cell, all the output generated by the cell and directed to the standard output, to the standard error or displayed by means of the IPython's display module, is instead captured and stored in the specified variable. It is then possible to display the captured content by calling the *show()* method on the variable:

```
[1]: import time

[2]: def test_comp_1(n):
         for i in range(1, n+1):
             time.sleep(60)
             print(i)

[ ]: %%capture captured_output
     test_comp_1(180)

[4]: captured_output.show()
```

```
1
2
3
4
5
6
7
8
9
10
```

Further details on the "capture" magic command can be found on the IPython official documentation: https://ipython.readthedocs.io/en/stable/interactive/magics.html

7. Service availability

JupyterHub@Trantor is unavailable every night from 02:00 to 02:15, due to the execution of automatic backup procedures. Note that the stop of JupyterHub does not affect the notebook servers running on compute nodes, even if you will not be able to connect to them.

8. List of preconfigured Conda virtual environments

- "python3"

  Base python 3 environment.

- "callisto"

  Python 3 environment for scientific computing. It includes the following packages:

  - Python 3.8.5
  - IPython 7.16.1
  - Numpy 1.19.1
  - Pandas 1.0.5
  - Scipy 1.5.0
  - Scikit-learn 0.23.1
  - Seaborn 0.10.1
  - Matplotlib 3.2.2

8. User-defined Conda virtual environments

Besides the preconfigured Conda virtual environments described before, users can create custom virtual environments according to their specific needs. The files related to these environments will be stored in the user's home directory.

The following sections describe the steps required respectively for creating and removing a Python virtual environment with the Conda utility. **The following commands must be executed by connecting via ssh to a head node of the Trantor cluster** (e.g. trantor01.sns.it)

If you are new to Conda, you can find a brief introduction at the HPC Center's Web site: https://hpccenter.sns.it/page/wiki/

[Optional] Setting conda-forge as primary channel

It is recommended to set "**conda-forge**" as the primary channel for searching and downloading packages, since it provides considerably more updated software with respect to the default channels.

> **Note**
> Setting **conda-forge** as the primary channel is **mandatory** if you want to create an **R**

environment for JupyterLab!

To do so, follow these steps:

1. Change the working directory to your home:

   ```
   cd
   ```

2. Exit from any active conda environment by executing (multiple time if needed) the following command:

   ```
   conda deactivate
   ```

3. Adds conda-forge to the top of the channel list:

   ```
   conda config --add channels conda-forge
   ```

   ```
   conda config --set channel_priority strict
   ```

Note
Although is it possible to make use of conda-forge only when needed, on a per-package basis (i.e. by using the `--channel` option of the the `conda install` command), this may sometime lead to incompatibilities between the packages downloaded from conda-forge and the ones retrieved from the default channels. Refer to the following Web page for a detailed explanation of the problem and possible solutions: https://conda-forge.org/docs/user/tipsandtricks.html#using-multiple-channels

Creating a custom Python-based virtual environment

1. Change the working directory to your home:

   ```
   cd
   ```

2. Create a new Conda environment with the following command:

   ```
   conda create -n my_env_name
   ```

   where "*my_env_name*" must be replaced with the name of the new environment.

3. Activate the new environment:

   ```
   conda activate my_env_name
   ```

4. Install the packages **python**, **ipykernel** (the IPython kernel) and **ipywidgets** (Jupyter interactive widgets) :

   ```
   conda install python=3.x ipykernel ipywidgets
   ```

   where "*x*" must be replaced with the desired Python version.

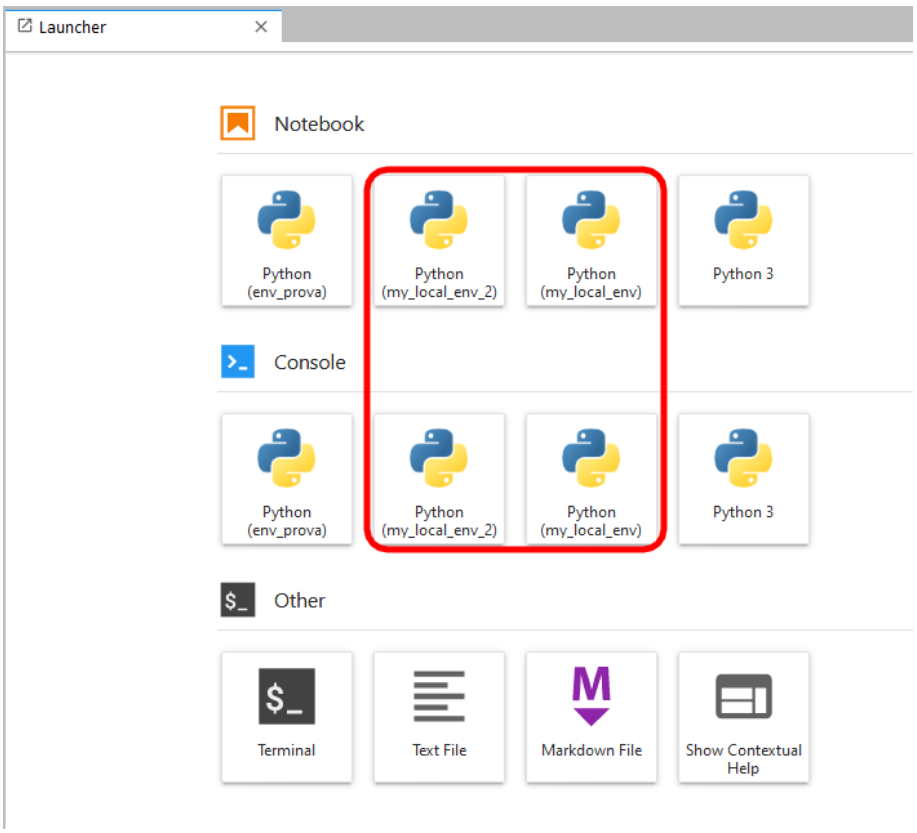
5. Expose the kernel of the new environment to JupyterLab:

   ```
   python -m ipykernel install --user --name 'my_env_name' --display-name "My Env Name"
   ```

6. Disable the conda environment:

   ```
   conda deactivate
   ```

7. Now, when starting a new notebook server, you should be able to see your local custom environments in the JupyterLab launcher (see figure below).



Creating a custom R-based virtual environment

0. Set conda-forge as the primary channel for searching and downloading packages, as previously explained. This operation must be performed just one time (there is no need to repeat this operation every time you create a new environment).

1. Change the working directory to your home:

   ```
   cd
   ```

2. Create a new Conda environment with the following command:

   ```
   conda create -n my_env_name
   ```

where “*my_env_name*” must be replaced with the name of the new environment.

3. Activate the new environment:

   ```
   conda activate my_env_name
   ```

4. Install the packages **r-base**, **r-essentials**, **xorg-libx11**, **xorg-libxext** **xorg-libxrender** and **xorg-libxt** :

   ```
   conda install r-base r-essentials xorg-libx11 xorg-libxext xorg-libxrender xorg-libxt
   ```

5. Find the name of the directory containing the current installation of JupyterLab. It is located under `/cluster/shared/software/` and its name follows the pattern “`jupyter_yyyymmdd`” (where “yyyy” is the year of installation, “mm” the month of installation and “dd” the day):

   ```
   ls /cluster/shared/software/
   ```

6. You now need to expose the kernel of the new environment to JupyterLab. To do so, starts the R shell in a modified environment where the “`/cluster/shared/software/jupyter_yyyymmdd/bin`” folder is prepended to the PATH environment variable:

   ```
   (PATH=/cluster/shared/software/jupyter_yyyymmdd/bin:$PATH R)
   ```

7. Within the R shell execute the following command:

```
IRkernel::installspec(name="my_env_name", displayname="My Env Name")
```

8. Exit from the R shell:

   ```
   q()
   ```

9. The shell will ask if you want to save the workspace. Answer ‘**n**’ :

   ```
   Save workspace image? [y/n/c]: n
   ```

10. Disable the conda environment:

    ```
    conda deactivate
    ```

11. Now, when starting a new notebook server you should be able to see your local custom environments in the JupyterLab launcher.

Modifying a custom virtual environment

1. Change the working directory to your home:

   ```
   cd
   ```

2. Activate the environment you wish to modify (if needed):

   `conda activate my_env_name`

3. Install or remove packages according to your needs:

   `conda install package_name_1 package_name_2 …`

   `or`

   `conda remove package_name_1 package_name_2 …`

4. If you do not need to work with this environment anymore, disable it:

   `conda deactivate`

Tip
In the case of **R** libraries, the package name usually starts with the "r-" prefix.

Tip
At the following Web page you can search among the packages available for the conda tool, provided by both the ufficial and third-party channels: https://anaconda.org/anaconda/repo

Installing Plotly

In order to to use the Plotly graphing libraries (https://plotly.com/graphing-libraries/) to produce graphs and charts in Jupyterlab, install one of the following packages (or both) :

- **plotly=4.9** - for Python
- **r-plotly=4.9** - for R

Please note that version matters! Different versions of these packages may lead to incompatibility problems with the current installation of Jupyterlab.

Removing a custom virtual environment

1. Change the working directory to your home:

   `cd`

2. Find the name of the directory containing the current installation of JupyterLab. It is located under `/cluster/shared/software/` and its name follows the pattern "`jupyter_yyyymmdd`" (where "yyyy" is the year of installation, "mm" the month of installation and "dd" the day):

   `ls /cluster/shared/software/`

3. Activate the conda environment containing the current installation of JupyterLab:

   ```
   conda activate /cluster/shared/software/jupyter_yyyymmdd
   ```

4. Remove the “link” between Jupyter and the environment we want to delete:

   ```
   jupyter kernelspec uninstall my_env_name
   ```

5. Delete the unwanted environment:

   ```
   conda remove --name my_env_name --all
   ```

6. Print the list of the local conda environments to confirm the removal:

   ```
   conda env list
   ```

7. Disable the conda environment:

   ```
   conda deactivate
   ```

9. Logs

The logs generated by each execution of the notebook server are stored in the **jupyter_logs** folder in your home directory. Please note that the log file for a given server will be available only **after the termination of the related job**.
Remember to clean up the jupyter_logs folder from time to time, since its content is accounted for in your disk space quota.

10. Best practices

- Be sure to **frequently save to file both the notebook and the results**, so as to preserve your work in the case of software crashes, loss of connection between the hub and the notebook server and some types of hardware failures. Longer is the computation, more important is to save intermediate and final results to file!

- Jupyter notebooks are designed primarily for algorithms prototyping and for interactive data analysis. Although one of the main benefits of running Jupyter notebooks on a compute node is to allow the execution of long-running computations directly from within the notebook, their execution time should be in the order of a few days, at most. For longer computations it is highly recommended to develop a dedicated python program and to manually submit it to the cluster by using the *qsub*[4] command.

- Perform any significant I/O operation on the scratch area.

- By the end of the computation, always copy any relevant file from the scratch area to your home directory. Then, clean up your scratch folder.

[4] See https://hpccenter.sns.it/page/wiki for a brief introduction to PBS.

- Although JupyterLab's user interface allows files uploading, this feature should be used only for small files (up to a few megabytes). In the case of large data files, please use tools like rsync, scp or sftp.

- When running multiple notebook servers, they can potentially operate on the same files. It should be noted, however, that **manipulating a file from more than one server is extremely dangerous, since it may lead to data loss or corruption!**
  Our suggestion is to organize your work in such a way that each server operates on a different folder, so as to avoid opening the same files from multiple servers.

- Upon server termination, double check the actual termination of the corresponding job by means of the *qstat* command.

- Remember to clean up the *jupyter_logs* folder from time to time, since its content is accounted for in your disk space quota.

11. Known bugs

- In the Job submission form (see Figure 2), the start of a new server may fail with the following error message "***Failed to recv data from background qsub***". This is due to an open bug of PBS. Just return to the home page and try again.

- Whenever a notebook server is started, JupyterHub generates a random authentication token which is passed to the notebook server. The list of the active tokens can be accessed by clicking on the "Token" link of the navigation bar (see Figure 4).

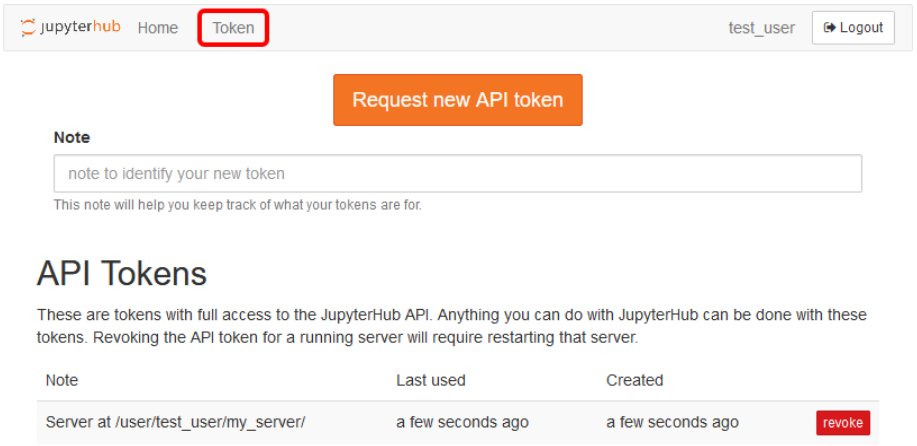


Figure 4: Tokens management page

In normal conditions, these tokens are automatically revoked at the termination of the related notebook servers. However, in case of errors when starting or stopping the server (e.g. due to the bug described above), the associated token may not be automatically removed.

It is a good practice to periodically check the Tokens page and to manually revoke any pending token. Keep in mind, however, that **this procedure must be performed**

only when no notebook server is running!
In fact, **revoking the token of a running server may lead to malfunctions** that require the re-start of the server.

- The download buttons in the main and contextual menus of JupyterLab do not work in Chrome.
- Sometimes the R Kernel (IRkernel) hangs when restarted. If this happens, shut down the kernel and then re-assign the kernel to the notebook.

If you experience a malfunction or find a new bug, please send a report to hpcstaff@sns.it.