FASTR Documentation

Release 3.0.0

Fastr contributors

Mar 05, 2019

Contents

FASTR is a framework that helps creating workflows of different tools. The workflows created in FASTR are automatically enhanced with flexible data input/output, execution options (local, cluster, etc) and solid provenance.

We chose to create tools by creating wrappers around executables and connecting everything with Python.

Fastr is open-source (licensed under the Apache 2.0 license) and hosted on bitbucket at https://bitbucket.org/bigr_erasmusmc/fastr

For support, go to https://groups.google.com/d/forum/fastr-users

To get yourself a copy, see the *Installation*

The official documentation can be found at fastr.readthedocs.io

The Fastr workflow system is presented in the following article:

> Hakim Achterberg, Marcel Koek, and Wiro Niessen. "Fastr: a workflow engine for advanced data flows in medical image analysis." Frontiers in ICT 3 (2016): 15.

Fastr is made possible by contributions from the following people: Hakim Achterberg, Marcel Koek, Adriaan Versteeg, Thomas Phil, Mattias Hansson, Baldur van Lew, Marcel Zwiers, and Coert Metz

FASTR Documentation

1.1 Introduction

Fastr is a system for creating workflows for automated processing of large scale data. A processing workflow might also be called a processing pipeline, however we feel that a pipeline suggests a linear flow of data. Fastr is designed to handle complex flows of data, so we prefer to use the term network. We see the workflow as a network of processing tools, through which the data will flow.

The original authors work in a medical image analysis group at Erasmus MC. They often had to run analysis that used multiple programs written in different languages. Every time a experiment was set up, the programs had to be glued together by scripts (often in bash or python).

At some point the authors got fed up by doing these things again and again, and so decided to create a flexible, powerful scripting base to easily create these scripts. The idea evolved to a framework in which the building blocks could be defined in XML and the networks could be constructed in very simple scripts (similar to creating a GUI).

1.1.1 Philosophy

Researchers spend a lot of time processing data. In image analysis, this often includes using multiple tools in succession and feeding the output of one tool to the next. A significant amount of time is spent either executing these tools by hand or writing scripts to automate this process. This process is time consuming and error-prone. Considering all these tasks are very similar, we wanted to write one elaborate framework that makes it easy to create pipelines, reduces the risk of errors, generates extensive logs, and guarantees reproducibility.

The Fastr framework is applicable to multiple levels of usage: from a single researcher who wants to design a processing pipeline and needs to get reproducible results for publishing; to applying a consolidated image processing pipeline to a large population imaging study. On all levels of application the pipeline provenance and managed execution of the pipeline enables you to get reliable results.

1.1.2 System overview

There are a few key requirements for the design of the system:

- Any tool that your computer can run using the command line (without user interaction) should be usable by the system without modifying the tool.

- The creation of a workflow should be simple, conceptual and require no real programming.

- Networks, once created, should be usable by anyone like a simple program. All processing should be done automatically.

- All processing of the network should be logged extensively, allowing for complete reproducibility of the system (guaranteeing data provenance).

Using these requirements we define a few key elements in our system:

- A `fastr.Tool` is a definition of any program that can be used as part of a pipeline (e.g. a segmentation tool)

- A `fastr.Node` is a single operational step in the workflow. This represents the execution of a `fastr.Tool`.

- A `fastr.Link` indicates how the data flows between nodes.

- A `fastr.Network` is an object containing a collection of `fastr.Node` and `fastr.Link` that form a workflow.

With these building blocks, the creation of a pipeline will boil down to just specifying the steps in the pipeline and the flow of the data between them. For example a simple neuro-imaging pipeline could look like:
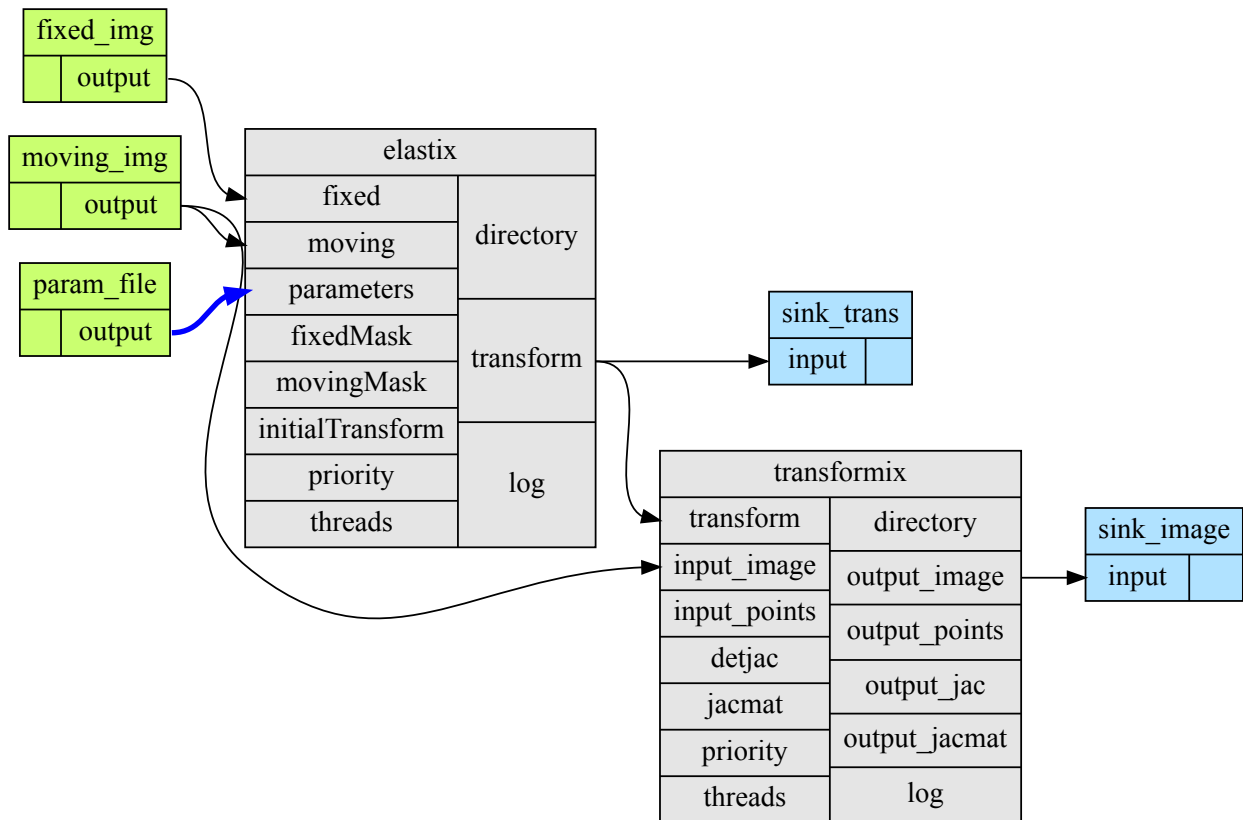

Fig. 1.1: A simple workflow that registers two images and uses the resulting transform to resample the moving image.

In Fastr this translates to:

- Create a `fastr.Network` for your pipeline

- Create a `fastr.SourceNode` for the fixed image

- Create a `fastr.SourceNode` for the moving image
- Create a `fastr.SourceNode` for the registration parameters
- Create a `fastr.Node` for the registration (in this case elastix)
- Create a `fastr.Node` for the resampling of the image (in this case transformix)
- Create a `fastr.SinkNode` to save the transformations
- Create a `fastr.SinkNode` to save the transformed images
- `fastr.Link` the output of the fixed image source node to the fixed image input of the registration node
- `fastr.Link` the output of the moving image source node to the moving image input of the registration node
- `fastr.Link` the output of the registration parameters source node to the registration parameters input of the registration node
- `fastr.Link` the output transform of the registration node to the transform input of the resampling node
- `fastr.Link` the output transform of the registration node to the input of transformation SinkNode
- `fastr.Link` the output image of the resampling node to the input of image SinkNode
- Run the `fastr.Network` for subjects X

This might seem like a lot of work for a registration, but the Fastr framework manages all other things, executes the pipeline and builds a complete paper trail of all executed operations. The execution can be on any of the supported execution environments (local, cluster, etc). The data can be imported from and exported to any of the supported data connections (file, XNAT, etc). It is also important to keep in mind that this is a simple example, but for more complex pipelines, managing the workflow with Fastr will be easier and less error-prone than writing your own scripts.

1.2 Quick start guide

This manual will show users how to install Fastr, configure Fastr, construct and run simple networks, and add tool definitions.

1.2.1 Installation

You can install Fastr either using pip, or from the source code.

Installing via pip

You can simply install fastr using `pip`:

```
pip install fastr
```

Note: You might want to consider installing `fastr` in a [virtualenv](#)

Installing from source code

To install from source code, use Mercurial via the command-line:

```
hg clone https://<yourusername>@bitbucket.org/bigr_erasmusmc/fastr  # for http
hg clone ssh://hg@bitbucket.org/bigr_erasmusmc/fastr # for ssh
```

If you prefer a GUI you can try TortoiseHG (Windows, Linux and Mac OS X) or SourceTree (Windows and Mac OS X). The address of the repository is (given for both http and ssh):

```
https://<yourusername>@bitbucket.org/bigr_erasmusmc/fastr
ssh://hg@bitbucket.org/bigr_erasmusmc/fastr
```

To install to your current Python environment, run:

```
cd fastr/
pip install .
```

This installs the scripts and packages in the default system folders. For windows this is the python `site-packages` directory for the fastr python library and `Scripts` directory for the executable scripts. For Ubuntu this is in the `/usr/local/lib/python2.7/dist-packages/` and `/usr/local/bin/` respectively.

Note: If you want to develop fastr, you might want to use `pip install -e .` to get an editable install

---

Note: You might want to consider installing `fastr` in a virtualenv

---

Note:

- On windows `python` and the `Scripts` directory are not on the system PATH by default. You can add these by going to `System -> Advanced Options -> Environment variables`.
- On mac you need the Xcode Command Line Tools. These can be installed using the command `xcode-select --install`.

1.2.2 Configuration

Fastr has defaults for all settings so it can be run out of the box to test the examples. However, when you want to create your own Networks, use your own data, or use your own Tools, it is required to edit your config file.

Fastr will search for a config file named `config.py` in the `$FASTRHOME` directory (which defaults to `~/.fastr/` if it is not set). So if `$FASTRHOME` is set the `~/.fastr/` will be ignored.

For a sample configuration file and a complete overview of the options in `config.py` see the *Config file* section.

1.2.3 Creating a simple network

If Fastr is properly installed and configured, we can start creating networks. Creating a network is very simple:

```
>>> import fastr
>>> network = fastr.create_network(id='example', version='1.0')
```

Now we have an empty network, the next step is to create some nodes and links. Imagine we want to create the following network:
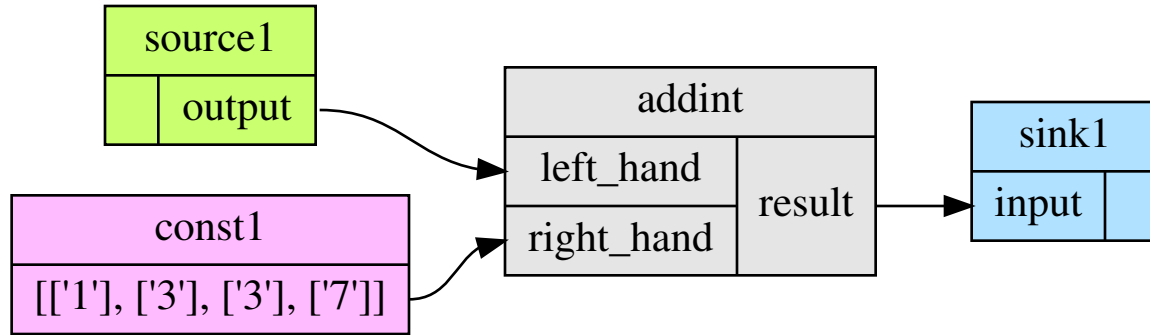
Creating nodes

We will create the nodes and add them to the network. This is done via the network `create_` methods. Let's create two source nodes, one normal node, and one sink:

```
>>> source1 = network.create_source('Int', id='source1')
>>> sink1 = network.create_sink('Int', id='sink1')
>>> addint = network.create_node('fastr/math/AddInt:1.0', tool_version='1.0', id=
↪'addint')
```

The functions `Network.create_source`, `Network.create_sink` and `Network.create_node` create the desired node and add it into the Network.

A SourceNode and SinkNode only require the datatype to be specified. A Node requires a Tool to be instantiated from. The `id` option is optional for all four, but makes it easier to identify the nodes and read the logs. The tool is defined by a namespace, the id and the version of the command. Many packages have multiple version which are available. The `tool_version` argument reflects the version of the Fastr wrapper which describes how the command can be called. For reproducibility also these are checked as they might be updated as well.

There is an easy way to add a constant to an input, by using a shortcut method. If you assign a `list` or `tuple` to an item in the input list, it will automatically create a ConstantNode and a `Link` between the ContantNode and the given Input:

```
>>> [1, 3, 3, 7] >> addint.inputs['right_hand']
Link link_0 (network: example):
   fastr:///networks/example/1.0/nodelist/const_addint_right_hand_0/outputs/output ==>
↪ fastr:///networks/example/1.0/nodelist/addint/inputs/right_hand/0
```

The created constant would have the id `const_addint__right_hand_0` as it automatically names the new constant `const_$nodeid__$inputid_$number`.

Note: The use of the >>, <<, and = operators for linking is discussed bellow in section *Creating links*.

In an interactive python session we can simply look at the basic layout of the node using the `repr` function. Just type the name of the variable holding the node and it will print a human readable representation:

```
>>> source1
SourceNode source1 (tool: Source:1.0 v1.0)
     Inputs          |        Outputs
------------------------------------------
                     |  output    (Int)
>>> addint
```

```
Node addint (tool: AddInt:1.0 v1.0)
       Inputs          |        Outputs
--------------------------------------------
left_hand  (Int)       |   result   (Int)
right_hand (Int)       |
```

This tool has inputs of type Int, so the sources and sinks need to have a matching datatype.

The tools and datatypes available are stored in *fastr.tools* and *fastr.types*. These variables are created when `fastr` is imported for the first time. They contain all the datatype and tools specified by the json or xml files in the search paths. To get an overview of the tools and datatypes loaded by fastr:

```
>>> fastr.tools
ToolManager
...
fastr/math/Add:1.0      1.0 :  .../fastr/resources/tools/fastr/math/1.0/add.xml
fastr/math/AddInt:1.0   1.0 :  .../fastr/resources/tools/fastr/math/1.0/addint.xml
...


>>> fastr.types
DataTypeManager
...
Directory               :   <URLType: Directory>
...
Float                   :   <ValueType: Float>
...
Int                     :   <ValueType: Int>
...
String                  :   <ValueType: String>
...
```

The `fastr.tools` variable contains all tools that Fastr could find during initalization. Tools can be chosen in two tways:

- `tools[id]` which returns the newest version of the tool
- `tools[id, version]` which returns the specified version of the tool

Creating links

So now we have a network with 4 nodes defined, however there is no relation between the nodes yet. For this we have to create some links.

```
>>> link1 = source1.output >> addint.inputs['left_hand']
>>> link2 = sink1.inputs['input'] <<  addint.outputs['result']
```

This asks the network to create links and immediately store them inside the network. A link always points from an Output to an Input (note that SubOutput or SubInputs are also valid). A SourceNode has only 1 output which is fixed, so it is easy to find. However, addImage has two inputs and one output, this requires us to specify which output we need. A normal node has a mapping with Inputs and one with Outputs. They can be indexed with the appropriate id's. The function returns the links, but you only need that if you are planning to change the properties of a link.

The operators with >> and << clearly indicate the direction of the desired data flow. Also they return the created link, which is easy if you want to change the flow in a link later on. The last short hand uses the assignment, but it cannot return the created link and changing the link later on is more difficult.

Create an image of the Network

For checking your Network it is very useful to have a graphical representation of the network. This can be achieved using the *Network.draw* method.

```
>>> network.draw()
'example.svg'
```

This will create a figure in the path returned by the function that looks like:
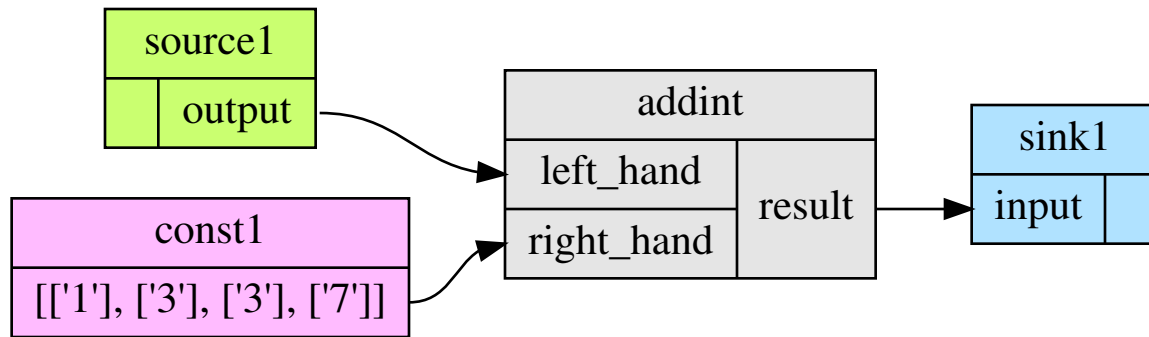
Note: for this to work you need to have graphviz installed

1.2.4 Running a Network

Running a network locally is almost as simple as calling the *Network.execute* method:

```
>>> source_data = {'source1': {'s1': 4, 's2': 5, 's3': 6, 's4': 7}}
>>> sink_data = {'sink1': 'vfs://tmp/fastr_result_{sample_id}.txt'}
>>> run = network.execute(source_data, sink_data)
>>> # Lots output will appear on the stdout while running
>>> run.result  # Show if the run was successful or if errors were encountered
True
```

As you can see the execute method needs data for the sources and sinks. This has to be supplied in two `dict` that have keys matching every source/sink `id` in the network. Not supplying data for every source and sink will result in an error, although it is possible to pass an empty `list` to a source.

Note: The values of the source data have to be simple values or urls and values of the sink data have to be url templates. To see what url schemes are available and how they work see *IOPlugin Reference*. For the sink url templates see `SinkeNode.set_data`

For source nodes you can supply a `list` or a `dict` with values. If you supply a `dict` the keys will be interpreted as sample ids and the values as the corresponding values. If you supply a `list`, keys will be generated in the form of `id_{N}` where N will be index of the value in the list.

Warning: As a `dict` does not have a fixed order, when a `dict` is supplied the samples are ordered by key to get a fixed order! For a `list` the original order is retained.

For the sink data, an url template has to be supplied that governs how the data is stored. The mini-lanuage (the replacement fields) are described in the `SinkNode.set_data` method.

To rerun a stopped/crashed pipeline check the user manual on *Continuing a Network*

1.3 User Manual

In this chapter we will discuss the parts of Fastr in more detail. We will give a more complete overview of the system and describe the more advanced features.

1.3.1 Tools

The Tool in Fastr are the building blocks of each workflow. A tool represents a program/script/binary that can be called by Fastr and can be seens as a template. A *Node* can be created based on a Tool. A Node will be one processing step in a workflow, and the tool defines what the step does.

On the import of Fastr, all available Tools will be loaded in a default ToolManager that can be accessed via `fastr.tools`. To get an overview of the tools in the system, just print the `repr()` of the ToolManager:

```
>>> import fastr
>>> fastr.tools
ToolManager
...
fastr.math.Add          v0.1 :  .../fastr/resources/tools/fastr/math/0.1/add.xml
fastr.math.AddInt       v0.1 :  .../fastr/resources/tools/fastr/math/0.1/addint.xml
...
```

As you can see it gives the tool id, version and the file from which it was loaded for each tool in the system. To view the layout of a tool, just print the `repr()` of the Tool itself.

```
>>> fastr.tools['AddInt']
Tool AddInt v0.1 (Add two integers)
       Inputs            |        Outputs
---------------------------------------------
left_hand  (Int)         |  result    (Int)
right_hand (Int)         |
```

To add a Tool to the system a file should be added to one of the path in `fastr.config.tools_path`. The structure of a tool file is described in *Tool description*

Create your own tool

There are 4 steps in creating a tool:

1. **Create folders.** We will call the tool ThrowDie. Create the folder throw_die in the folder fastr-tools. In this folder create another folder called bin.

2. **Place executable in correct place.** In this example we will use a snippet of executable python code:

```python
#!/usr/bin/env python
import sys
import random
import json
```

(continues on next page)

```python
if (len(sys.argv) > 1):
    sides = int(sys.argv[1])
else:
    sides = 6
result = [int(random.randint(1, sides ))]


print('RESULT={}'.format(json.dumps(result)))
```

Save this text in a file called `throw_die.py`

Place the executable python script in the folder `throw_die/bin`

3. **Create and edit xml file for tool.** See *tool definition reference* for all the fields that can be defined in a tool.

Put the following text in file called `throw_die.xml`.

```xml
<tool id="ThrowDie" description="Simulates a throw of a die. Number of sides of␣
↪the die is provided by user"
    name="throw_die" version="1.0">
  <authors>
    <author name="John Doe" />
  </authors>
  <command version="1.0" >
    <authors>
      <author name="John Doe" url="http://a.b/c" />
    </authors>
    <targets>
      <target arch="*" bin="throw_die.py" interpreter="python" os="*" paths='bin/
↪'/>
    </targets>
    <description>
        throw_die.py number_of_sides
        output = simulated die throw
    </description>
  </command>
  <interface>
    <inputs>
      <input cardinality="1" datatype="Int" description="Number of die sides" id=
↪"die_sides" name="die sides" nospace="False" order="0" required="True"/>
    </inputs>
    <outputs>
      <output id="output" name="output value" datatype="Int" automatic="True"␣
↪cardinality="1" method="json" location="^RESULT=(.*)$" />
    </outputs>
  </interface>
</tool>
```

Put throw_die.xml in the folder example_tool. All Attributes in the example above are required. For a complete overview of the xml Attributes that can be used to define a tool, check the *Tool description*. The most important Attributes in this xml are:

```
id      : The id is used in in FASTR to create an instance of your tool, this␣
↪name will appear in the tools when you type fastr.tools.
targets : This defines where the executables are located and on which platform␣
↪they are available.
inputs  : This defines the inputs that you want to be used in FASTR, how FASTR␣
↪should use them and what data is allowed to be put in there.
```

More xml examples can be found in the fastr-tools folder.

4) **Edit configuration file.** Append the line `[PATH TO LOCATION OF FASTR-TOOLS]/fastr-tools/ throw_die/` to the the `config.py` (located in ~/.fastr/ directory) to the `tools_path`. See *Config file* for more information on configuration.

You should now have a working tool. To test that everything is ok do the following in python:

```
>>> import fastr
>>> fastr.tools
...
```

Now a list of available tools should be produced, including the tool ThrowDie

To test the tool create the script test_throwdie.py:

```python
import fastr

# Create network
network = fastr.create_network('ThrowDie')

# Create nodes
source1 = network.create_source('Int', id='source1')
sink1 = network.create_sink('Int', id='sink1')
throwdie = network.create_node('ThrowDie', id='throwdie')

# Create links
link1 = source1.output >> throwdie.inputs['die_sides']
link2 = throwdie.outputs['output'] >> sink1.inputs['input']

# Draw and execute
source_data = {'source1': {'s1': 4, 's2': 5, 's3': 6, 's4': 7}}
sink_data = {'sink1': 'vfs://tmp/fastr_result_{sample_id}.txt'}
network.draw()
network.execute(source_data, sink_data)
```

Call the script from commandline by

```
$ python test_throwdie.py
```

An image of the network will be created in the current directory and result files will be put in the tmp directory. The result files are called `fastr_result_s1.txt`, `fastr_result_s2.txt`, `fastr_result_s3.txt`, and `fastr_result_s4.txt`

Note: If you have code which is operating system depend you will have to edit the xml file. The following gives and example of how the elastix tool does this:

```xml
<targets>
    <target os="windows" arch="*" bin="elastix.exe">
      <paths>
        <path type="bin" value="vfs://apps/elastix/4.7/install/" />
        <path type="lib" value="vfs://apps/elastix/4.7/install/lib" />
      </paths>
    </target>
    <target os="linux" arch="*" modules="elastix/4.7" bin="elastix">
      <paths>
        <path type="bin" value="vfs://apps/elastix/4.7/install/" />
        <path type="lib" value="vfs://apps/elastix/4.7/install/lib" />
```

(continues on next page)

```
        </paths>
      </target>
      <target os="darwin" arch="*" modules="elastix/4.7" bin="elastix">
        <paths>
          <path type="bin" value="vfs://apps/elastix/4.7/install/" />
          <path type="lib" value="vfs://apps/elastix/4.7/install/lib" />
        </paths>
      </target>
  </targets>
```

`vfs` is the virtual file system path, more information can be found at *VirtualFileSystem*.

1.3.2 Network

A *Network* represented an entire workflow. It hold all *Nodes*, *Links* and other information required to execute the workflow. Networks can be visualized as a number of building blocks (the Nodes) and links between them:
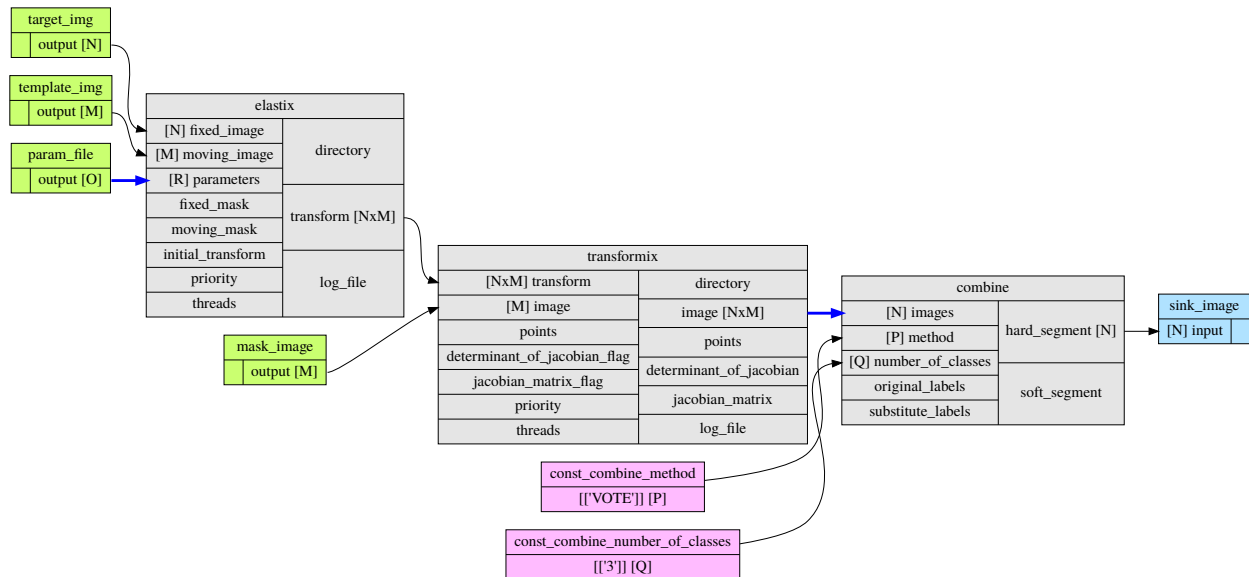
An empty network is easy to create, all you need is to name it:

```
>>> network = fastr.create_network(id="network_name")
```

the *network* is the main interface to fastr, from it you can create all elements to create a workflow. in the following sections the different elements of a *network* will be described in more detail.

Node

Nodes are the point in the *Network* where the processing happens. A *Node* takes the input data and executes jobs as specified by the underlying Tool. A *Nodes* can be created easily:

```
>>> node2 = network.create_node(tool, id='node1', step_id='step1')
```

We tell the `Network` to create a Node using the *create_node* method. Optionally you can add define a `step_id` for the node which is a logical grouping of Nodes that is mostly used for visualization.

Note: For a Node, the tool can be given both as the Tool class or the id of the tool. This id can be just the id or a tuple with the id and version.

A *Node* contains *Inputs* and *Outputs*. To see the layout of the *Node* one can simply look at the `repr()`.

```
>>> addint = network.create_node('AddInt', id='addint')
>>> addint
Node addint (tool: AddInt v1.0)
      Inputs          |        Outputs
-------------------------------------------
left_hand  (Int)      |   result   (Int)
right_hand (Int)      |
```

The inputs and outputs are located in mappings with the same name:

```
>>> addint.inputs
<Input map, items: ['left_hand', 'right_hand']>

>>> addint.outputs
<Output map, items: ['result']>
```

The InputMap and OutputMap are classes that behave like mappings. The InputMap also facilitates the linking shorthand. By assigning an *Output* to an existing key, the InputMap will create a *Link* between the Input and Output.

SourceNode

A SourceNode is a special kind of node that is the start of a workflow. The SourceNodes are given data at run-time that fetched via IOPlugins. On create, only the datatype of the data that the SourceNode supplied needs to be known. Creating a SourceNode is very similar to an ordinary node:

```
>>> source1 = network.create_source('Int', id='source1', step_id='step1', node_group=
↪'subject')
```

The first argument is the type of data the source supplies. The other optional arguments are for naming and grouping of the nodes. A SourceNode only has a single output which has a short-cut access via `source.output`.

Note: For a source or constant node, the datatype can be given both as the *BaseDataType* class or the id of the datatype.

ConstantNode

A ConstantNode is another special node. It is a subclass of the SourceNode and has a similar function. However, instead of setting the data at run-time, the data of a constant is given at creation and saved in the object. Creating a ConstantNode is similar as creating a source, but with supplying data:

```
>>> constant1 = network.create_constant('Int', [42], id='constant1', step_id='step1',
↪node_group='subject)
```

The first argument is the datatype the node supplies, similar to a SourceNode. The second argument is the data that is contained in the ConstantNode. Often, when a ConstantNode is created, it is created specifically for one input and will not be reused. In this case there is a shorthand to create and link a constant to an input:

```
>>> link = addint.inputs['value1'] << [42]
>>> link = [42] >> addint.inputs['value1']
>>> addint.inputs['value1'] = [42]
```

are three methods that will create a constant node with the value 42 and create a link between the output and input `addint.value1`.

SinkNode

The SinkNode is the counter-part of the source node. Instead of get data into the workflow, it saves the data resulting from the workflow. For this a rule has to be given at run-time that determines where to store the data. The information about how to create such a rule is described at `SinkNode.set_data`. At creation time, only the datatype has to be specified:

```
>>> sink2 = network.create_sink('Int', id='sink2', step_id='step1', node_group=
→'subject')
```

Link

Links indicate how the data flows between *Nodes*. Links can be created explicitly using on of the following:

```
>>> link = network.create_link(node1.outputs['image'], node2.inputs['image'])
```

or can be create implicitly by a short hand (there are three options):

```
# This style of assignment will create a Link similar to above
>>> link = node1.outputs['image'] >> node2.inputs['image']
>>> link = node2.inputs['image'] << node1.outputs['image']
>>> node2.inputs['image'] = node1.outputs['image']
```

Note that a *Link* is also create automatically when using the short-hand for the ConstantNode <fastr.planning.node.ConstantNode>'.

1.3.3 Data Flow

The data enters the Network via SourceNodes flows via other Node and leaves the Network via SinkNodes.The flow between Nodes goes from an Output via a Link to an Input. In the following image it is simple to track the data from the SourceNodes at the left to the SinkNodes at right side:
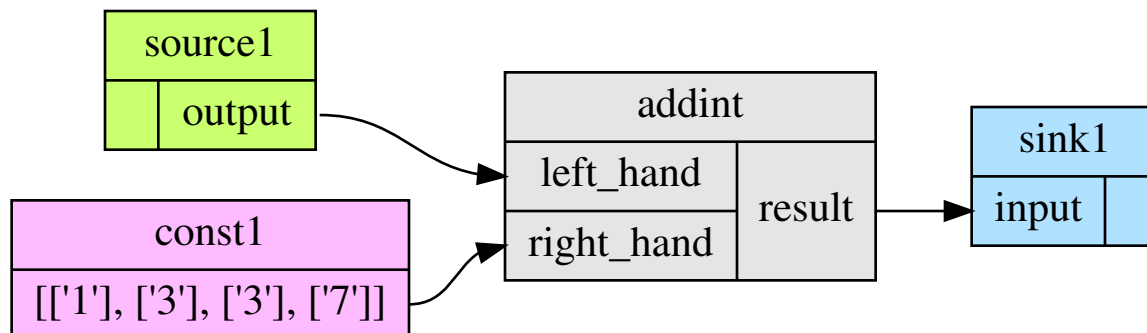
Note that the data in Fastr is stored in the Output and the Link and Input just give access to it (possible while transforming the data).

Data flow inside a Node

In a Node all data from the Inputs will be combined and the jobs will be generated. There are strict rules to how this combination is performed. In the default case all inputs will be used pair-wise, and if there is only a single value for an input, it it will be considered as a constant.

To illustrate this we will consider the following Tool (note this is a simplified version of the real tool):

```
>>> fastr.tools['Elastix']
Tool Elastix v4.8 (Elastix Registration)
                        Inputs                                    |              Outputs
--------------------------------------------------------------------------------------------
↪--------
fixed_image        (ITKImageFile)                                 |  transform␣
↪(ElastixTransformFile)
moving_image       (ITKImageFile)                                 |
parameters         (ElastixParameterFile)                         |
```

Also it is important to know that for this tool (by definition) the cardinality of the `transform` Output will match the cardinality of the `parameters` Input.

If we supply a Node based on this Tool with a single sample on each Input there will be one single matching Output sample created:
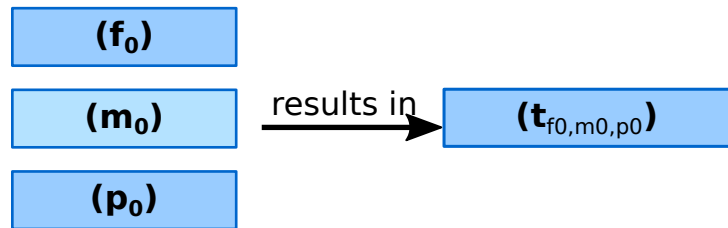
If the cardinality of the `parameters` sample would be increased to 2, the resulting `transform` sample would also become 2:
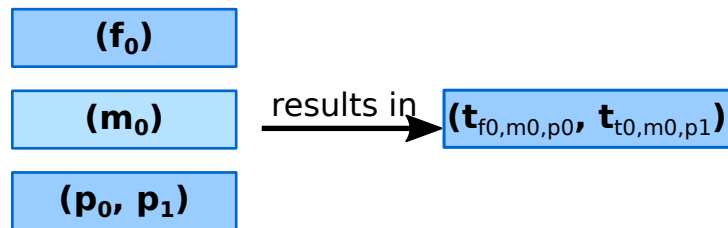
Now if the number of samples on `fixed_image` would be increased to 3, the `moving_image` and `parameters` will be considered constant and be repeated, resulting in 3 `transform` samples.
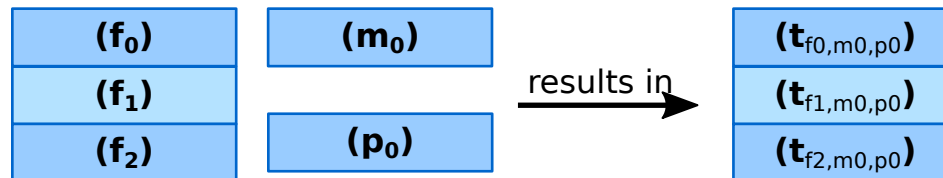
Then if the amount of samples for `moving_image` is also increased to 3, the `moving_image` and `fixed_image` will be used pairwise and the `parameters` will be constant.

(f$_0$)	(m$_0$)	results in	(t$_{f0,m0,p0}$, t$_{f0,m0,p1}$)
(f$_1$)	(m$_1$)		(t$_{f1,m1,p0}$, t$_{f1,m1,p1}$)
(f$_2$)	(m$_2$)		(t$_{f2,m2,p0}$, t$_{f2,m2,p1}$)

(p$_0$, p$_1$)
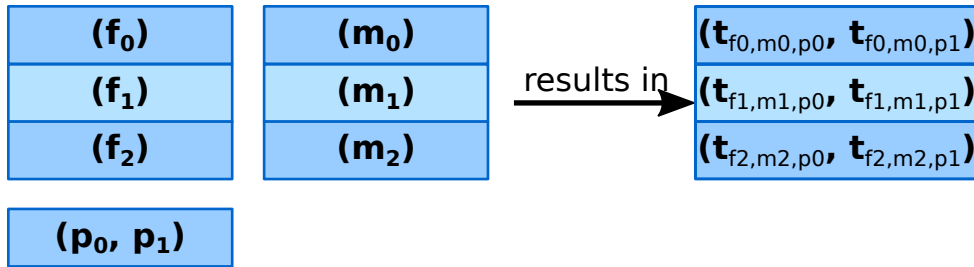
Advanced flows in a Node

Sometimes the default pairwise behaviour is not desirable. For example if you want to test all combinations of certain input samples. To achieve this we can change the *input_group* of *Inputs* to set them apart from the rest. By default all Inputs are assigned to the `default` input group. Now let us change that:

```
>>> node = network.create_node('Elastix', id='elastix')
>>> node.inputs['moving_image'].input_group = 'moving'
```

This will result in `moving_image` to be put in a different input group. Now if we would supply `fixed_image` with 3 samples and `moving_image` with 4 samples, instead of an error we would get the following result:

(f$_0$)		(m$_0$)	results in	(t$_{f0,m0,p0}$)	(t$_{f0,m1,p0}$)	(t$_{f0,m2,p0}$)	(t$_{f0,m3,p0}$)
(f$_1$)	⊗	(m$_1$)		(t$_{f1,m0,p0}$)	(t$_{f1,m1,p0}$)	(t$_{f1,m2,p0}$)	(t$_{f1,m3,p0}$)
(f$_2$)		(m$_2$)		(t$_{f2,m0,p0}$)	(t$_{f2,m1,p0}$)	(t$_{f2,m2,p0}$)	(t$_{f2,m3,p0}$)
(p$_0$)		(m$_3$)					
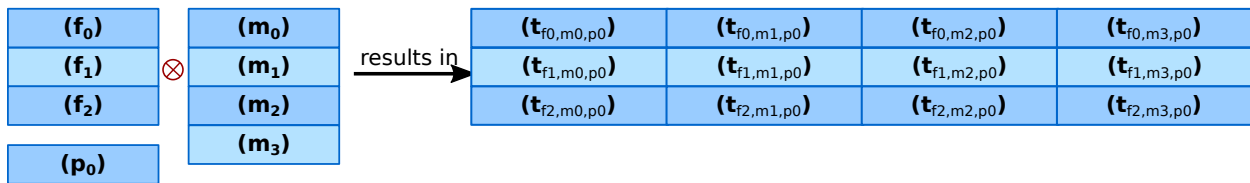
> **Warning:** TODO: Expand this section with the merging dimensions

Data flows in a Link

As mentioned before the data flows from an Output to an Input through a Link. By default the Link passed the data as is, however there are two special directives that change the shape of the data:

1. Collapsing flow, this collapses certain dimensions from the sample array into the cardinality. As a user you have to specify the dimension or tuple of dimensions you want to collapse.

	dim2	
(a$_0$)		
(b$_0$)	(b$_1$)	
(c$_0$)	(c$_1$)	

collapse (dim1) →
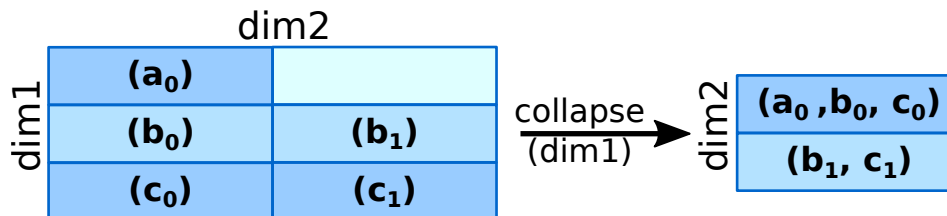
dim2	
(a$_0$, b$_0$, c$_0$)
(b$_1$, c$_1$)

This is useful in situation where you want to use a tool that aggregates over a number of samples (e.g. take a mean or sum).

To achieve this you can set the *collapse* property of the *Link* as follows:

```
>>> link.collapse = 'dim1'
>>> link.collapse = ('dim1', 'dim2')  # In case you want to collapse multiple␣
→dimensions
```

2. Expanding flow, this turns the cardinality into a new dimension. The new dimension will be named after the `Output` from which the link originates. It will be in the form of `{nodeid}__{outputid}`
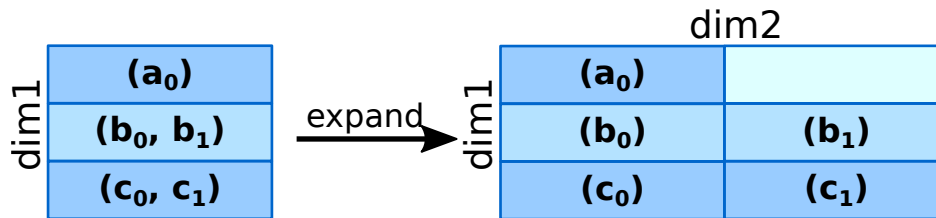
This flow directive is useful if you want to split a large sample in multiple smaller samples. This could be because processing the whole sample is not feasible because of resource constraints. An example would be splitting a 3D image into slices to process separately to avoid high memory use or to achieve parallelism.

To achieve this you can set the `expand` property of the `Link` to `True`:

```
>>> link.expand = True
```

Note: both collapsing and expanding can be used on the same link, it will executes similar to a expand-collapse sequence, but the newly created expand dimension is ignored in the collapse.
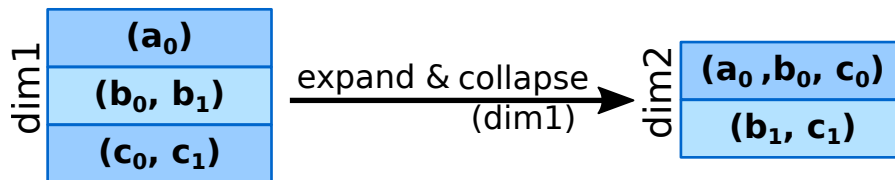

```
>>> link.collapse = 'dim1'
>>> link.expand = True
```

Data flows in an Input

If an Input has multiple Links attached to it, the data will be combined by concatenating the values for each corresponding sample in the cardinality.

Broadcasting (matching data of different dimensions)

Sometimes you might want to combine data that does not have the same number of dimensions. As long as all dimensions of the lower dimensional datasets match a dimension in the higher dimensional dataset, this can be achieved using *broadcasting*. The term *broadcasting* is borrowed from NumPy and described as:

> "The term broadcasting describes how numpy treats arrays with different shapes during arithmetic operations. Subject to certain constraints, the smaller array is "broadcast" across the larger array so that they have compatible shapes."
>
> —NumPy manual on broadcasting

In fastr it works similar, but to combined different Inputs in an InputGroup. To illustrate broadcasting it is best to use an example, the following network uses broadcasting in the `transformix` Node:
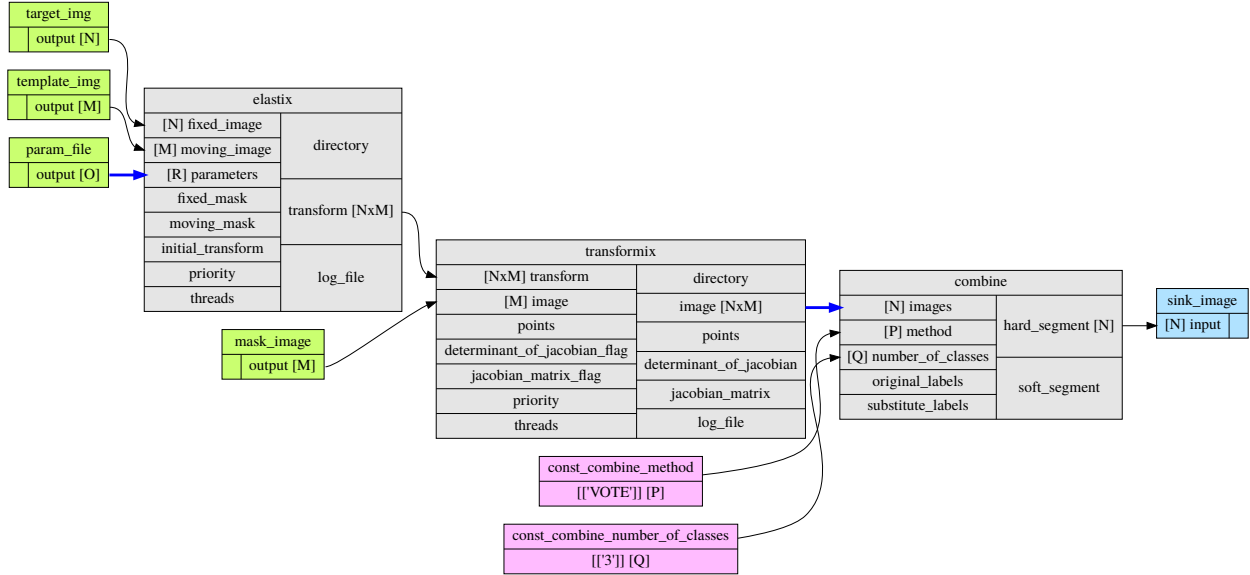
As you can see this visualization prints the dimensions for each Input and Output (e.g. the `elastix.fixed_image` Input has dimensions `[N]`). To explain what happens in more detail, we present an image illustrating the details for the samples in `elastix` and `transformix`:
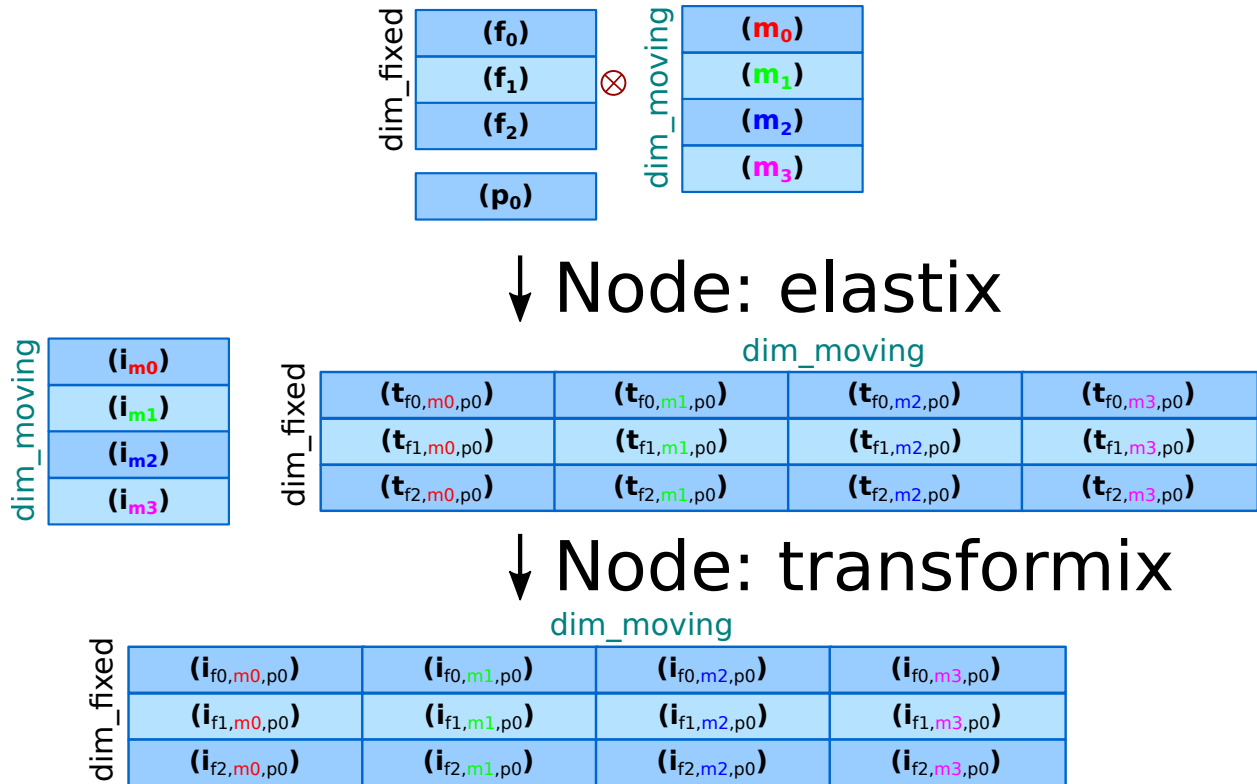
In the figure the `moving_image` (and references to it) are identified with different colors, so they are easy to track across the different steps.

At the top the Inputs for the `elastix` Node are illustrated. Because the input groups a set differently, output samples are generated for all combinations of `fixed_image` and `moving_image` (see *Advanced flows in a Node* for details).

In the `transformix` Node, we want to combine a list of samples that is related to the `moving_image` (it has the same dimension name and sizes) with the resulting `transform` samples from the `elastix` Node. As you can see the sizes of the sample collections do not match (`[N]` vs `[N x M]`). This is where *broadcasting* comes into play, it allows the system to match these related sample collections. Because all the dimensions in `[N]` are known in `[N x M]`, it is possible to match them uniquely. This is done automatically and the result is a new `[N xM]` sample collection. To create a matching sample collections, the samples in the `transformix.image` Input are reused as indicated by the colors.

> **Warning:** Note that this might fail when there are data-blocks with non-unique dimension names, as it will be not be clear which of the dimensions with identical names should be matched!

1.3.4 DataTypes

In Fastr all data is contained in object of a specific type. The types in Fastr are represented by classes that subclass *BaseDataType*. There are a few different other classes under *BaseDataType* that are each a base class for a family of types:

- *DataType* – The base class for all types that hold data
  - *ValueType* – The base class for types that contain simple data (e.g. Int, String) that can be represented as a str
  - *EnumType* – The base class for all types that are a choice from a `set` of options
  - *URLType* – The base class for all types that have their data stored in files (which are referenced by URL)
- *TypeGroup* – The base class for all types that actually represent a group of types
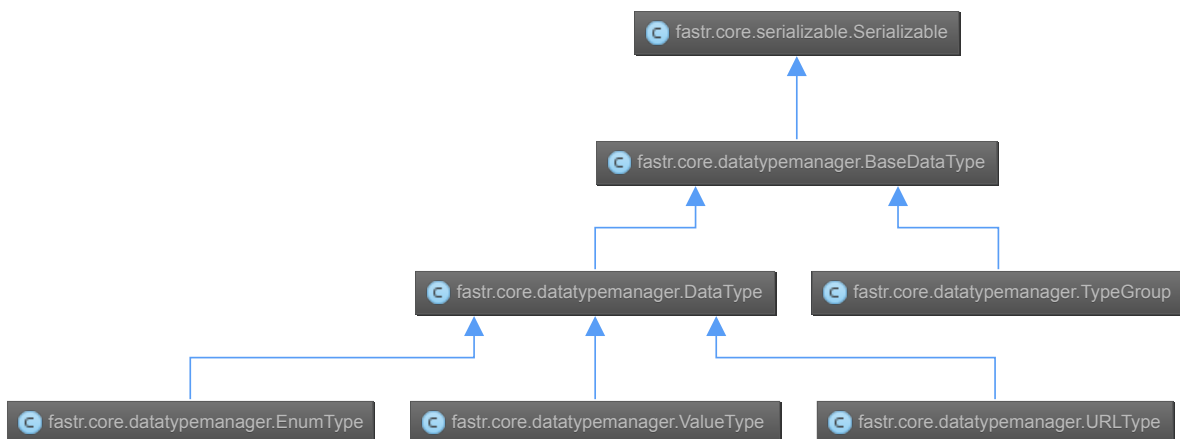
Fig. 1.2: The relation between the different DataType classes

The types are defined in xml files and created by the `DataTypeManager`. The `DataTypeManager` acts as a container containing all Fastr types. It is automatically instantiated as `fastr.types`. In fastr the created DataTypes classes are also automatically place in the *fastr.datatypes* module once created.

Resolving Datatypes

Outputs in fastr can have a *TypeGroup* or a number of *DataTypes* associated with them. The final *DataType*

used will depend on the linked *Inputs*. The *DataType* resolving works as a two-step procedure.

1. All possible *DataTypes* are determined and considered as *options*.

2. The best possible *DataType* from *options* is selected for non-automatic Outputs

The *options* are defined as the intersection of the set of possible values for the *Output* and each separate *Input* connected to the *Output*. Given the resulting *options* there are three scenarios:

- If there are no valid *DataTypes* (*options* is empty) the result will be None.

- If there is a single valid *DataType*, then this is automatically the result (even if it is not a preferred *DataType*).

- If there are multiple valid *DataTypes*, then the preferred *DataTypes* are used to resolve conflicts.

There are a number of places where the preferred *DataTypes* can be set, these are used in the order as given:

1. The *preferred* keyword argument to `match_types`

2. The preferred types specified in the *fastr.config*

1.3.5 Execution

Executing a Network is very simple:

```
>>> source_data = {'source_id1': ['val1', 'val2'],
                   'source_id2': {'id3': 'val3', 'id4': 'val4'}}
>>> sink_data = {'sink_id1': 'vfs://some_output_location/{sample_id}/file.txt'}
>>> network.execute(source_data, sink_data)
```

The *Network.execute* method takes a `dict` of source data and a `dict` sink data as arguments. The dictionaries should have a key for each SourceNode or SinkNode.

The execution of a Network uses a layered model:

- *Network.execute* will analyze the Network and call all Nodes.

- `Node.execute` will create jobs and fill their payload

- *execute_job* will execute the job on the execute machine and resolve any deferred values (`val://` urls).

- *Tool.execute* will find the correct target and call the interface and if required resolve `vfs://` urls

- *Interface.execute* will actually run the required command(s)

The `ExecutionPlugin` will call call the *executionscript.py* for each job, passing the job as a gzipped pickle file. The *executionscript.py* will resolve deferred values and then call *Tool.execute* which analyses the required target and executes the underlying *Interface*. The Interface actually executes the job and collect the results. The result is returned (via the Tool) to the *executionscript.py*. There we save the result, provenance and profiling in a new gzipped pickle file. The execution system will use a callback to load the data back into the Network.

The selection and settings of the `ExecutionPlugin` are defined in the *fastr config*.

Continuing a Network

Normally a random temporary directory is created for each run. To continue a previously stopped/crashed network, you should call the *Network.execute* method using the same temporary directory(tmp dir). You can set the temporary directory to a fixed value using the following code:

```
>>> tmpdir = '/tmp/example_network_rerun'
>>> network.execute(source_data, sink_data, tmpdir=tmpdir)
```

> **Warning:** Be aware that at this moment, Fastr will rerun only the jobs where not all output files are present or if the job/tool parameters have been changed. It will not rerun if the input data of the node has changed or the actual tools have been adjusted. In these cases you should remove the output files of these nodes, to force a rerun.

1.3.6 IOPlugins

Sources and sink are used to get data in and out of a `Network` during execution. To make the data retrieval and storage easier, a plugin system was created that selects different plugins based on the URL scheme used. So for example, a url starting with `vfs://` will be handles by the `VirtualFileSystem plugin`. A list of all the `IOPlugins` known by the system and their use can be found at *IOPlugin Reference*.

1.3.7 Secrets

Fastr uses a secrets system for storing and retrieving login credentials. Currently the following keyrings are supported:

- Python keyring and keyrings.alt lib: - Mac OS X Keychain - Freedesktop Secret Service (requires secretstorage) - KWallet (requires dbus) - Windows Credential Vault - Gnome Keyring - Google Keyring (stores keyring on Google Docs) - Windows Crypto API (File-based keyring secured by Windows Crypto API) - Windows Registry Keyring (registry-based keyring secured by Windows Crypto API) - PyCrypto File Keyring - Plaintext File Keyring (not recommended)

- Netrc (not recommended)

When a password is retrieved trough the fastr SecretService it loops trough all of the available SecretProviders (currently keyring and netrc) until a match is found.

The Python keyring library automatically picks the best available keyring backend. If you wish to choose your own python keyring backend it is possible to do so by make a keyring configuration file according to the keyring library documentation. The python keyring library connects to one keyring. Currently it cannot loop trough all available keyrings until a match is found.

1.3.8 Debugging

This section is about debugging Fastr tools wrappers, Fastr Networks (when building a Network) and Fastr Network Runs.

Debugging a Fastr tool

When wrapping a Tool in Fastr sometimes it will not work as expected or not load properly. Fastr is shipped with a command that helps checking Tools. The *fastr verify* command can try to load a Tool in steps to make it more easy to understand where the loading went wrong.

The `fastr verify` command will use the following steps:

- Try to load the tool with and without compression

- Try to find the correct serializer and make sure the format is correct

- Try to validate the Tool content against the json_schema of a proper Tool

---

- Try to create a Tool object

- If available, execute the tool test

An example of the use of `fastr verify`:

```
$ fastr verify tool fastr/resources/tools/fastr/math/0.1/add.xml
[INFO]    verify:0020 >> Trying to read file with compression OFF
[INFO]    verify:0036 >> Read data from file successfully
[INFO]    verify:0040 >> Trying to load file using serializer "xml"
[INFO]    verify:0070 >> Validating data against Tool schema
[INFO]    verify:0080 >> Instantiating Tool object
[INFO]    verify:0088 >> Loaded tool <Tool: Add version: 1.0> successfully
[INFO]    verify:0090 >> Testing tool...
```

If your Tool is loading but not functioning as expected you might want to easily test your Tool without building an entire Network around it that can obscure errors. It is possible to run a tool from the Python prompt directly using `tool.execute`:

```
>>> tool.execute(left_hand=40, right_hand=2)
[INFO] localbinarytarget:0090 >> Changing ./bin
[INFO]    tool:0311 >> Target is <Plugin: LocalBinaryTarget>
[INFO]    tool:0318 >> Using payload: {'inputs': {'right_hand': (2,), 'left_hand':
→(40,)}, 'outputs': {}}
[INFO] localbinarytarget:0135 >> Adding extra PATH: ['/home/hachterberg/dev/fastr-
→develop/fastr/fastr/resources/tools/fastr/math/0.1/bin']
[INFO] fastrinterface:0393 >> Execution payload: {'inputs': {'right_hand': (2,),
→'left_hand': (40,)}, 'outputs': {}}
[INFO] fastrinterface:0496 >> Adding (40,) to argument list based on <fastrinterface.
→InputParameterDescription object at 0x7fc950fa8850>
[INFO] fastrinterface:0496 >> Adding (2,) to argument list based on <fastrinterface.
→InputParameterDescription object at 0x7fc950fa87d0>
[INFO] localbinarytarget:0287 >> Options: ['/home/hachterberg/dev/fastr-develop/fastr/
→fastr/resources/tools/fastr/math/0.1/bin']
[INFO] localbinarytarget:0201 >> Calling command arguments: ['python', '/home/
→hachterberg/dev/fastr-develop/fastr/fastr/resources/tools/fastr/math/0.1/bin/addint.
→py', '--in1', '40', '--in2', '2']
[INFO] localbinarytarget:0205 >> Calling command: "'python' '/home/hachterberg/dev/
→fastr-develop/fastr/fastr/resources/tools/fastr/math/0.1/bin/addint.py' '--in1' '40
→' '--in2' '2'"
[INFO] fastrinterface:0400 >> Collecting results
[INFO] executionpluginmanager:0467 >> Callback processing thread ended!
[INFO] executionpluginmanager:0467 >> Callback processing thread ended!
[INFO] executionpluginmanager:0467 >> Callback processing thread ended!
[INFO] jsoncollector:0076 >> Setting data for result with [42]
<fastr.core.interface.InterfaceResult at 0x7fc9661ccfd0>
```

In this case an AddInt was ran from the python shell. As you can see it shows the payload it created based on the call, followed by the options for the directories that contain the binary. Then the command that is called is given both as a list and string (for easy copying to the prompt yourself). Finally the collected results is displayed.

Note: You can give input and outputs as keyword arguments for execute. If an input and output have the same name, you can disambiguate them by prefixing them with `in_` or `out_` (e.g. `in_image` and `out_image`)

Debugging an invalid Network

The simplest command to check if your Network is considered valid is to use the `Network.is_valid` method. It will simply check if the Network is valid:

```
>>> network.is_valid()
True
```

It will return a boolean that only indicates the validity of the Network, but it will print any errors it found to the console/log with the ERROR log level, for example when datatypes on a link do not match:

```
>>> invalid_network.is_valid()
[WARNING] datatypemanager:0388 >> No matching DataType available (args (<ValueType:␣
→Float class [Loaded]>, <ValueType: Int class [Loaded]>))
[WARNING]      link:0546 >> Cannot match datatypes <ValueType: Float class [Loaded]>␣
→and <ValueType: Int class [Loaded]> or not preferred datatype is set! Abort linking␣
→fastr:///networks/add_ints/0.0/nodelist/source/outputs/output to fastr:///networks/
→add_ints/0.0/nodelist/add/inputs/left_hand!
[WARNING] datatypemanager:0388 >> No matching DataType available (args (<ValueType:␣
→Float class [Loaded]>, <ValueType: Int class [Loaded]>))
[ERROR]   network:0571 >> [add] Input left_hand is not valid: SubInput fastr:///
→networks/add_ints/0.0/nodelist/add/inputs/left_hand/0 is not valid: SubInput source␣
→(link_0) is not valid
[ERROR]   network:0571 >> [add] Input left_hand is not valid: SubInput fastr:///
→networks/add_ints/0.0/nodelist/add/inputs/left_hand/0 is not valid: [link_0] source␣
→and target have non-matching datatypes: source Float and Int
[ERROR]   network:0571 >> [link_0] source and target have non-matching datatypes:␣
→source Float and Int
False
```

Because the messages might not always be enough to understand errors in the more complex Networks, we would advice you to create a plot of the network using the `network.draw_network` method:

```
>>> network.draw_network(network.id, draw_dimensions=True, expand_macro=True)
'add_ints.svg'
```

The value returned is the path of the output image generated (it will be placed in the current working directory. The `draw_dimensions=True` will make the drawing add indications about the sample dimensions in each Input and Output, whereas `expand_macro=True` causes the draw to expand MacroNodes and draw the content of them. If you have many nested MacroNodes, you can set `expand_macro` to an integer and that is the depth until which the MacroNodes will be draw in detail.

An example of a simple multi-atlas segmentation Network nicely shows the use of drawing the dimensions, the dimensions vary in certain Nodes due to the use of input_groups and a collapsing link (drawn in blue):
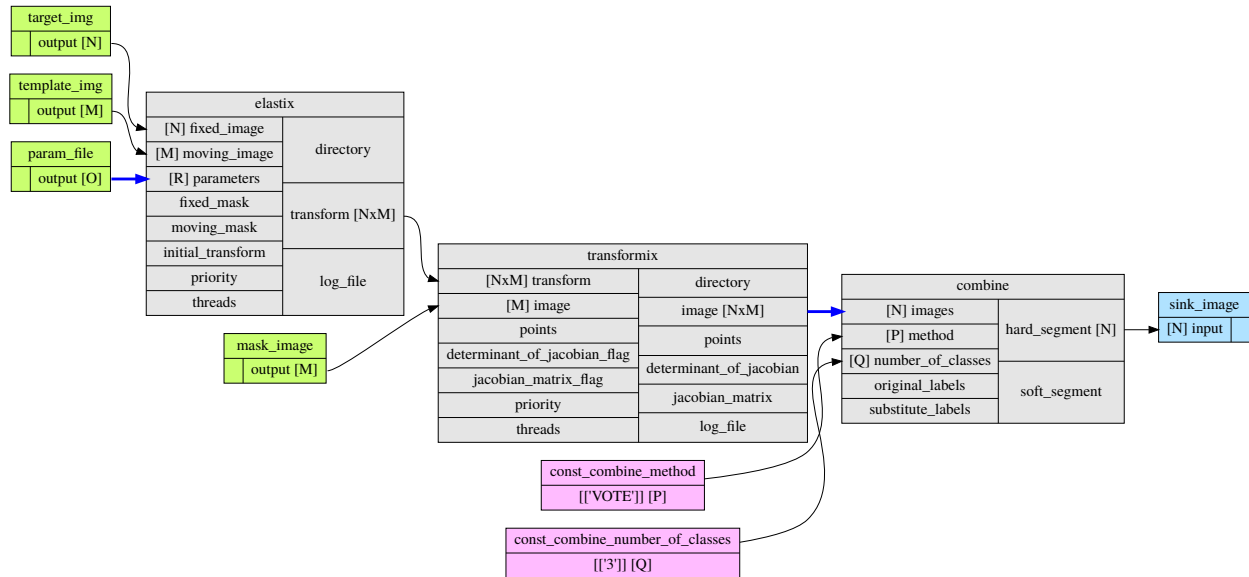
Debugging a Network run with errors

If a Network run did finish but there were errors detected, Fastr will report those at the end of the execution. We included an example of a Network that has failing samples in `fastr/examples/failing_network.py` which can be used to test debugging. An example of the output of a Network run with failures:

```
[INFO] networkrun:0604 >> #################################
[INFO] networkrun:0605 >> #    network execution FINISHED    #
[INFO] networkrun:0606 >> #################################
[INFO] networkrun:0618 >> ===== RESULTS =====
[INFO] networkrun:0627 >> sink_1: 2 success / 2 failed
[INFO] networkrun:0627 >> sink_2: 2 success / 2 failed
[INFO] networkrun:0627 >> sink_3: 1 success / 3 failed
[INFO] networkrun:0627 >> sink_4: 1 success / 3 failed
[INFO] networkrun:0627 >> sink_5: 1 success / 3 failed
[INFO] networkrun:0628 >> ===================
[WARNING] networkrun:0651 >> There were failed samples in the run, to start debugging␣
→you can run:

    fastr trace $RUNDIR/__sink_data__.json --sinks

see the debug section in the manual at https://fastr.readthedocs.io/en/default/static/
→user_manual.html#debugging for more information.
```

As you can see, there were failed samples in every sink. Also you already get the suggestion to use *fastr trace*. This command helps you inspect the staging directory of the Network run and pinpoint the errors.

The suggested command will print a similar summary as given by the network execution:

```
$ fastr trace $RUNDIR/__sink_data__.json --sinks
sink_1 -- 2 failed -- 2 succeeded
sink_2 -- 2 failed -- 2 succeeded
sink_3 -- 3 failed -- 1 succeeded
sink_4 -- 3 failed -- 1 succeeded
sink_5 -- 3 failed -- 1 succeeded
```

Since this is not given us new information we can add the `-v` flag for more output and limit the output to one sink, in this case `sink_5`:

```
$ fastr trace $RUNDIR/__sink_data__.json --sinks sink_5
sink_5 -- 3 failed -- 1 succeeded
  sample_1_1: Encountered error: [FastrOutputValidationError] Could not find result
→for output out_2 (/home/hachterberg/dev/fastr-develop/fastr/fastr/execution/job.
→py:970)
  sample_1_2: Encountered error: [FastrOutputValidationError] Could not find result
→for output out_1 (/home/hachterberg/dev/fastr-develop/fastr/fastr/execution/job.
→py:970)
  sample_1_3: Encountered error: [FastrOutputValidationError] Could not find result
→for output out_1 (/home/hachterberg/dev/fastr-develop/fastr/fastr/execution/job.
→py:970)
  sample_1_3: Encountered error: [FastrOutputValidationError] Could not find result
→for output out_2 (/home/hachterberg/dev/fastr-develop/fastr/fastr/execution/job.
→py:970)
```

Now we are given one error per sample, but this does not yet give us that much information. To get a very detailed report we have to specify one sink and one sample. This will make the `fastr trace` command print a complete error report for that sample:

```
$ fastr trace $RUNDIR/__sink_data__.json --sinks sink_5 --sample sample_1_1 -v
Tracing errors for sample sample_1_1 from sink sink_5
Located result pickle: /home/hachterberg/FastrTemp/fastr_failing_network_2017-09-
→04T10-44-58_uMWeMV/step_1/sample_1_1/__fastr_result__.pickle.gz


===== JOB failing_network___step_1___sample_1_1 =====
Network: failing_network
Run: failing_network_2017-09-04T10-44-58
Node: step_1
Sample index: (1)
Sample id: sample_1_1
Status: JobState.execution_failed
Timestamp: 2017-09-04 08:45:19.238192
Job file: /home/hachterberg/FastrTemp/fastr_failing_network_2017-09-04T10-44-58_
→uMWeMV/step_1/sample_1_1/__fastr_result__.pickle.gz

Command:
List representation: [u'python', u'/home/hachterberg/dev/fastr-develop/fastr/fastr/
→resources/tools/fastr/util/0.1/bin/fail.py', u'--in_1', u'1', u'--in_2', u'1', u'--
→fail_2']
String representation: 'python' '/home/hachterberg/dev/fastr-develop/fastr/fastr/
→resources/tools/fastr/util/0.1/bin/fail.py' '--in_1' '1' '--in_2' '1' '--fail_2'

Output data:
{'out_1': [<Int: 2>]}

Status history:
2017-09-04 08:45:19.238212: JobState.created
2017-09-04 08:45:21.537417: JobState.running
2017-09-04 08:45:31.578864: JobState.execution_failed

----- ERRORS -----
- FastrOutputValidationError: Could not find result for output out_2 (/home/
→hachterberg/dev/fastr-develop/fastr/fastr/execution/job.py:970)
- FastrValueError: [failing_network___step_1___sample_1_1] Output values are not
→valid! (/home/hachterberg/dev/fastr-develop/fastr/fastr/execution/job.py:1421)
```

```
-----------------

----- STDOUT -----
Namespace(fail_1=False, fail_2=True, in_1=1, in_2=1)
in 1  : 1
in 2  : 1
fail_1: False
fail_2: True
RESULT_1=[2]


-----------------

----- STDERR -----

-----------------
```

As shown above, it finds the result files of the failed job(s) and prints the most important information. The first paragraph shows the information about the Job that was involved. The second paragraph shows the command used both as a list (which is clearer and internally used in Python) and as a string (which you can copy/paste to the shell to test the command). Then there is the output data as determined by Fastr. The next section shows the status history of the Job which can give an indication about wait and run times. Then there are the errors that Fastr encounted during the execution of the Job. In this case it could not find the output for the Tool. Finally the stdout and stderr of the subprocess are printed. In this case we can see that RESULT_2=[. . .] was not in the stdout, and so the result could not be located.

Note: Sometimes there are no Job results in a directory, this usually means the process got killed before the Job could finished. On cluster environments, this often means that the process was killed due to memory constraints.

Asking for help with debugging

If you would like help with debugging, you can contact us via the fastr-users google group. To enable us to track the errors please include the following:

- The entire log of the fastr run (can be copied from console or from the end of `~/.fastr/logs/info.log`.
- A dump of the network run, which can be created that by using the *fastr dump* command like:

```
$ fastr dump $RUNDIR fastr_run_dump.zip
```

    This will create a zip file including all the job files, logs, etc but not the actual data files.

These should be enough information to trace most errors. In some cases we might need to ask for additional information (e.g. tool files, datatype files) or actions from your side.

1.3.9 Naming Convention

For the naming convention of the tools we tried to stay close to the Python **PEP 8** coding style. In short, we defined toolnames as classes so they should be UpperCamelCased. The inputs and outputs of a tool we considered as functions or method arguments, these should we named lower_case_with_underscores.

An overview of the mapping of Fastr to **PEP 8**:

Fastr construct	Python **PEP8** equivalent	Examples
Network.id	**module**	brain_tissue_segmentation
Tool.id	**class**	BrainExtractionTool, ThresholdImage
Node.id	**variable name**	brain_extraction, threshold_mask
Input/Output.id	**method**	image, number_of_classes, probability_image

Furthermore there are some small guidelines:

- No input or output in the input or output names. This is already specified when setting or getting the data.

- Add the type of the output that is named. i.e. enum, string, flag, image,

   - No File in the input/output name (Passing files around is what Fastr was developed for).

   - No type necessary where type is implied i.e. lower_threshold, number_of_levels, max_threads.

- Where possible/useful use the fullname instead of an abbreviation.

1.3.10 Provenance

For every data derived data object, Fastr records the Provenance. The `SinkNode` write provenance records next to every data object it writes out. The records contain information on what operations were performed to obtain the resulting data object.

W3C Prov

The provenance is recorded using the W3C Prov Data Model (PROV-DM). Behind the scences we are using the python prov implementation.

The PROV-DM defines 3 Starting Point Classes and and their relating properties. See Fig. 1.3 for a graphic representation of the classes and the relations.

Fig. 1.3: The three Starting Point classes and the properties that relate them. The diagrams in this document depict Entities as yellow ovals, Activities as blue rectangles, and Agents as orange pentagons. The responsibility properties are shown in pink.*[0]

*

Implementation

In the workflow document the provenance classes map to fastr concepts in the following way:

   Agent Fastr, *Networks*, *Tools*, *Nodes*

   Activity *Jobs*

   Entities Data

---

[0] This picture and caption is taken from http://www.w3.org/TR/prov-o/ . "Copyright © 2011-2013 World Wide Web Consortium, (MIT, ERCIM, Keio, Beihang). http://www.w3.org/Consortium/Legal/2015/doc-license"

Usage

The provenance is stored in ProvDocument objects in pickles. The convenience command line tool `fastr prov` can be used to extract the provenance in the PROV-N notation and can be serialized to PROV-JSON and PROV-XML. The provenance document can also be vizualized using the `fastr prov` command line tool.

1.4 Command Line Tools

Fastr is shipped with a number of command line tools to perform common tasks and greatly simplify things such as debugging. The list of command line tools that is included in Fastr:

command	description
cat	Print information from a job file
dump	Dump the contents of a network run tempdir into a zip for remote assistance
execute	Execute a fastr job file
extract_argparse	Create a stub for a Tool based on a python script using argparse
provenance	Get PROV information from the result pickle.
pylint	Dump the contents of a network run tempdir into a zip for remote assistance
run	Run a Network from the commandline
sink	Command line access to the IOPlugin sink
source	Command line access to the IOPlugin source
test	Run the tests of a tool to verify the proper function
trace	Trace samples/sinks from a run
upgrade	Print information from a job file
verify	Verify fastr resources, at the moment only tool definitions are supported.
webapp	Start the fastr webapp and open in a new browser tab

1.4.1 fastr cat

Extract selected information from the extra job info. The path is the selection of the data to retrieve. Every parts of the path (separated by a /) is seen as the index for the previous object. So for example to get the stdout of a job, you could use 'fastr cat __fastr_extra_job_info__.json process/stdout'.

```
usage: fastr cat [-h] __fastr_extra_job_info__.json path
```

Positional Arguments

> **__fastr_extra_job_info__.json** result file to cat
>
> **path** path of the data to print

1.4.2 fastr dump

Create a dump of a network run directory that contains the most important information for debugging. This includes a serialization of the network, all the job command and result files, the extra job information files and the provenance files. No data files will be included, but note that if jobs get sensitive information passed via the command line this will be included in the job files.

```
usage: fastr dump [-h] RUNDIR DUMP.zip
```

Positional Arguments

RUNDIR	The run directory to dump
DUMP.zip	The file to place the dump in

1.4.3 fastr execute

Execute a job from commandline.

```
usage: fastr execute [-h] [JOBFILE]
```

Positional Arguments

JOBFILE	File of the job to execute (default ./__fastr_command__.pickle.gz)

1.4.4 fastr extract_argparse

Extract basic information from argparse.

```
usage: fastr extract_argparse [-h] SCRIPT.py TOOL.xml
```

Positional Arguments

SCRIPT.py	Python script to inspect
TOOL.xml	created Tool stub

1.4.5 fastr provenance

Export the provenance information from JSON to other formats or plot the provenance data as a graph.

```
usage: fastr provenance [-h] [-so SYNTAX_OUT_FILE] [-sf SYNTAX_FORMAT]
                        [-i INDENT] [-vo VISUALIZE_OUT_FILE]
                        [RESULTFILE]
```

Positional Arguments

RESULTFILE	File of the job to execute (default ./__fastr_prov__.json)

Named Arguments

-so, --syntax-out-file   Write the syntax to file.

-sf, --syntax-format   Choices are: [json], provn or xml

Default: "json"

-i, --indent   Indent size of the serialized documents.

Default: 2

-vo, --visualize-out-file   Visualize the provenance. The most preferred format is svg. You can specify any format pydot supports. Specify the format by postfixing the filename with an extension.

1.4.6 fastr pylint

Run pylint in such a way that the output is written to a file

```
usage: fastr pylint [-h] --output_file PYLINT.OUT
```

Named Arguments

--output_file   The file to result in

1.4.7 fastr run

Execute a job or network from commandline.

```
usage: fastr run [-h] NETWORKFILE
```

Positional Arguments

NETWORKFILE   File of the network to execute

1.4.8 fastr sink

executes an ioplugin

```
usage: fastr sink [-h] -i INPUT [INPUT ...] -o OUTPUT [OUTPUT ...]
                  [-d DATATYPE [DATATYPE ...]]
```

Named Arguments

-i, --input   The url to process (can also be a list)

-o, --output   The output urls in vfs scheme (can also be a list and should be the same size as –inurl)

-d, --datatype   The datatype of the source/sink data to handle

1.4.9 fastr source

Executes an source command

```
usage: fastr source [-h] -i INPUT [INPUT ...] -o OUTPUT [-d DATATYPE]
                    [-s SAMPLE_ID]
```

Named Arguments

> **-i, --input**   The url to process (can also be a list)
>
> **-o, --output**   The output url in vfs scheme
>
> **-d, --datatype**   The datatype of the source/sink data to handle
>
> **-s, --sample_id** The sample_id of the source/sink data to handle

1.4.10 fastr test

Run a tests for a fastr resource.

```
usage: fastr test [-h] {tool,tools,network,networks} ...
```

Sub-commands:

tool

Test a single tool

```
fastr test tool [-h] TOOL
```

Positional Arguments

> **TOOL**   Tool to test or directory with tool reference data

tools

Test all tools known to fastr

```
fastr test tools [-h]
```

network

Test a single network

```
fastr test network [-h] NETWORK
```

Positional Arguments

> NETWORK   The reference data to test the Network

networks

Test all network references inside subdirectories

```
fastr test networks [-h] [--result RESULT.json] REFERENCE
```

Positional Arguments

> REFERENCE   path of the directory containing subdirectories with reference data

Named Arguments

> --result   Write the results of the test to a JSON file

1.4.11 fastr trace

Fastr trace helps you inspect the staging directory of the Network run and pinpoint the errors.

```
usage: fastr trace [-h] [--verbose] [--sinks [SINKS [SINKS ...]]]
                   [--samples [SAMPLES [SAMPLES ...]]]
                   [__sink_data__.json]
```

Positional Arguments

> __sink_data__.json   result file to cat
>
>   Default: "/home/docs/checkouts/readthedocs.org/user_builds/fastr/checkouts/3.0.0/fastr/doc/__sink_data_

Named Arguments

> --verbose, -v   set verbose output for more details
>
>   Default: False
>
> --sinks   list results for specified sinks
>
> --samples   list result for all samples

1.4.12 fastr upgrade

Extract selected information from the extra job info. The path is the selection of the data to retrieve. Every parts of the path (separated by a /) is seen as the index for the previous object. So for example to get the stdout of a job, you could use 'fastr cat __fastr_extra_job_info__.json process/stdout'.

```
usage: fastr upgrade [-h] NETWORK.py NEW.py
```

Positional Arguments

NETWORK.py	Network creation file (in python) to upgrade
NEW.py	location of the result file

1.4.13 fastr verify

Verify fastr resources, at the moment only tool definitionsare supported.

```
usage: fastr verify [-h] TYPE path
```

Positional Arguments

TYPE	Possible choices: tool
	Type of resource to verify (e.g. tool)
path	path of the resource to verify

1.4.14 fastr webapp

Starts the fastr web client.

```
usage: fastr webapp [-h] [-d] [-o]
```

Named Arguments

-d, --debug	Debug mode.
	Default: False
-o, --openpage	Open web page after start.
	Default: False

1.5 Resource File Formats

This chapter describes the various files fastr uses. The function and format of the files is described allowing the user to configure fastr and add DataTypes and Tools.

1.5.1 Config file

Fastr reads the config files from the following locations by default (in order):

- `$FASTRHOME/config.py`
- `~/.fastr/config.py`

Reading a new config file change or override settings, making the last config file read have the highest priority. All settings have a default value, making config files and all settings within optional.

Example config file

Here is a minimal config file:

```python
# Enable debugging output
debug = False

# Define the path to the tool definitions
tools_path = ['/path/to/tools',
              '/path/to/other/tools'] + tools_path
types_path = ['/path/to/datatypes',
              '/path/to/other/datatypes'] + types_path


# Specify what your preferred output types are.
preferred_types += ["NiftiImageFileCompressed",
                    "NiftiImageFile"]

# Set the tmp mount
mounts['tmp'] = '/path/to/tmpdir'
```

Format

The config file is actually a python source file. The next syntax applies to setting configuration values:

```python
# Simple values
float_value = 1.0
int_value = 1
str_value = "Some value"
other_str_value = 'name'.capitalize()

# List-like values
list_value = ['over', 'ride', 'values']
other_list_value.prepend('first')
other_list_value.append('list')

# Dict-like values
dict_value = {'this': 1, 'is': 2, 'fixed': 3}
other_dict_value['added'] = 'this key'
```

Note: Dictionaries and list always have a default, so you can always append or assign elements to them and do not have to create them in a config file. Best practice is to only edit them unless you really want to block out the earliers config files.

Most operations will be assigning values, but for list and dict values a special wrapper object is used that allows manipulations from the default. This limits the operations allowed.

List values in the `config.py` have the following supported operators/methods:

- +, __add__ and __radd__
- += or __iadd__

- `append`

- `prepend`

- `extend`

Mapping (dict-like) values in the `config.py` have the following supported operators/methods:

- `update`

- `[]` or `__getitem__`, `__setitem__` and `__delitem__`

Configuration fields

This is a table the known config fields on the system:

1.5.2 `Tool` description

Tools are the building blocks in the fastr network. To add new *Tools* to fastr, XML/json files containing a *Tool* definition can be added. These files have the following layout:

Attribute			Description
`id`			The id of this Tool (used internally in fastr)
`name`			The name of the Tool, for human readability
`version`			The version of the Tool wrapper (not the binary)
`url`			The url of the Tool wrapper
`authors[]`			List of authors of the Tools wrapper
	`name`		Name of the author
	`email`		Email address of the author
	`url`		URL of the website of the author
`tags`	`tag[]`		List of tags describing the Tool
`command`			Description of the underlying command
	`version`		Version of the tool that is wrapped
	`url`		Website where the tools that is wrapped can be obtained
	`targets[]`		Description of the target binaries/script of this Tool
		`os`	OS targeted (windows, linux, macos or * (for any)
		`arch`	Architecture targeted 32, 64 or * (for any)
		`...`	Extra variables based on the target used, see *Targets*
	`description`		Description of the Tool
	`license`		License of the Tool, either full license or a clear name (e.g. LGPL, GPL v2)
	`authors[]`		List of authors of the Tool (not the wrapper!)
		`name`	Name of the authors
		`email`	Email address of the author
		`url`	URL of the website of the author
`interface`			The interface definition see *Interfaces*
`help`			Help text explaining the use of the Tool
`cite`			Bibtext of the Citation(s) to reference when using this Tool for a publication

1.6 Resource Reference

In this chapter we describe the different plugins bundled with Fastr (e.g. IOPlugins, ExecutionPlugins). The reference is build automatically from code, so after installing a new plugin the documentation has to be rebuild for it to be included in the docs.

1.6.1 CollectorPlugin Reference

`CollectorPlugins` are used for finding and collecting the output data of outputs part of a `FastrInterface`

scheme	CollectorPlugin
JsonCollector	*JsonCollector*
PathCollector	*PathCollector*
StdoutCollector	*StdoutCollector*

JsonCollector

The JsonCollector plugin allows a program to print out the result in a pre-defined JSON format. It is then used as values for fastr.

The working is as follows:

1. The location of the output is taken

2. If the location is `None`, go to step 5

3. The substitutions are performed on the location field (see below)

4. The location is used as a regular expression and matched to the stdout line by line

5. The matched string (or entire stdout if location is `None`) is `loaded as a json`

6. The data is parsed by `set_result`

The structure of the JSON has to follow the a predefined format. For normal `Nodes` the format is in the form:

```
[value1, value2, value3]
```

where the multiple values represent the cardinality.

For a `FlowNodes` the format is the form:

```
{
    'sample_id1': [value1, value2, value3],
    'sample_id2': [value4, value5, value6]
}
```

This allows the tool to create multiple output samples in a single run.

PathCollector

The PathCollector plugin for the FastrInterface. This plugin uses the location fields to find data on the filesystem. To use this plugin the method of the output has to be set to `path`

The general working is as follows:

1. The location field is taken from the output

2. The substitutions are performed on the location field (see below)

3. The updated location field will be used as a regular expression filter

4. The filesystem is scanned for all matching files/directory

The special substitutions performed on the location use the Format Specification Mini-Language Format Specification Mini-Language. The predefined fields that can be used are:

- `inputs`, an objet with the input values (use like `{inputs.image[0]}`) The input contains the following attributes that you can access:

  - `.directory` for the directory name (use like `input.image[0].directory`) The directory is the same as the result of `os.path.dirname`

  - `.filename` is the result of `os.path.basename` on the path

  - `.basename` for the basename name (use like `input.image[0].basename`) The basename is the same as the result of `os.path.basename` and the extension stripped. The extension is considered to be everything after the first dot in the filename.

  - `.extension` for the extension name (use like `input.image[0].extension`)

- `output`, an object with the output values (use like `{outputs.result[0]}`) It contains the same attributes as the input

  - `special.cardinality`, the index of the current cardinality

  - `special.extension`, is the extension for the output DataType

Example use:

```
<output ... method="path" location="{output.directory[0]}/TransformParameters.
↪{special.cardinality}.{special.extension}"/>
```

Given the output directory `./nodeid/sampleid/result`, the second sample in the output and filetype with a `txt` extension, this would be translated into:

```
<output ... method="path" location="./nodeid/sampleid/result/TransformParameters.1.
↪txt>
```

StdoutCollector

The StdoutCollector can collect data from the stdout stream of a program. It filters the `stdout` line by line matching a predefined regular expression.

The general working is as follows:

1. The location field is taken from the output

2. The substitutions are performed on the location field (see below)

3. The updated location field will be used as a regular expression filter

4. The `stdout` is scanned line by line and the regular expression filter is applied

The special substitutions performed on the location use the Format Specification Mini-Language Format Specification Mini-Language. The predefined fields that can be used are:

- `inputs`, an objet with the input values (use like `{inputs.image[0]}`)

- `outputs`, an object with the output values (use like `{outputs.result[0]}`)

- `special` which has two subfields:

- – `special.cardinality`, the index of the current cardinality

- – `special.extension`, is the extension for the output DataType

Note: because the plugin scans line by line, it is impossible to catch multi-line output into a single value

---

1.6.2 ExecutionPlugin Reference

This class is the base for all Plugins to execute jobs somewhere. There are many methods already in place for taking care of stuff.

There are fall-backs for certain features, but if a system already implements those it is usually preferred to skip the fall-back and let the external system handle it. There are a few flags to enable disable these features:

- `cls.SUPPORTS_CANCEL` indicates that the plugin can cancel queued jobs

- `cls.SUPPORTS_HOLD_RELEASE` indicates that the plugin can queue jobs in a hold state and can release them again (if not, the base plugin will create a hidden queue for held jobs). The plugin should respect the `Job.status == JobState.hold` when queueing jobs.

- `cls.SUPPORTS_DEPENDENCY` indicate that the plugin can manage job dependencies, if not the base plugin job dependency system will be used and jobs with only be submitted when all dependencies are met.

- `cls.CANCELS_DEPENDENCIES` indicates that if a job is cancelled it will automatically cancel all jobs depending on that job. If not the plugin traverse the dependency graph and kill each job manual.

Note: If a plugin supports dependencies it is assumed that when a job gets cancelled, the depending job also get cancelled automatically!

Most plugins should only need to redefine a few abstract methods:

- `__init__` the constructor

- `cleanup` a clean up function that frees resources, closes connections, etc

- `_queue_job` the method that queues the job for execution

Optionally an extra job finished callback could be added:

- `_job_finished` extra callback for when a job finishes

If `SUPPORTS_CANCEL` is set to True, the plugin should also implement:

- `_cancel_job` cancels a previously queued job

If `SUPPORTS_HOLD_RELEASE` is set to True, the plugin should also implement:

- `_hold_job` hold_job a job that is currently held

- `_release_job` releases a job that is currently held

If `SUPPORTED_DEPENDENCY` is set to True, the plugin should:

- Make sure to use the `Job.hold_jobs` as a list of its dependencies

Not all of the functions need to actually do anything for a plugin. There are examples of plugins that do not really need a `cleanup`, but for safety you need to implement it. Just using a `pass` for the method could be fine in such a case.

> **Warning:** When overwriting other functions, extreme care must be taken not to break the plugins working, as there is a lot of bookkeeping that can go wrong.

scheme	ExecutionPlugin
BlockingExecution	*BlockingExecution*
DRMAAExecution	*DRMAAExecution*
LinearExecution	*LinearExecution*
ProcessPoolExecution	*ProcessPoolExecution*
RQExecution	*RQExecution*
SlurmExecution	*SlurmExecution*
StrongrExecution	*StrongrExecution*

BlockingExecution

The blocking execution plugin is a special plugin which is meant for debug purposes. It will not queue jobs but immediately execute them inline, effectively blocking fastr until the Job is finished. It is the simplest execution plugin and can be used as a template for new plugins or for testing purposes.

DRMAAExecution

A DRMAA execution plugin to execute Jobs on a Grid Engine cluster. It uses a configuration option for selecting the queue to submit to. It uses the python `drmaa` package.

> **Note:** To use this plugin, make sure the `drmaa` package is installed and that the execution is started on an SGE submit host with DRMAA libraries installed.

> **Note:** This plugin is at the moment tailored to SGE, but it should be fairly easy to make different subclasses for different DRMAA supporting systems.

Configuration fields

name	type	description	default
drmaa_queue	str	The default queue to use for jobs send to the scheduler	'week'
drmaa_max_jobs	int	The maximum jobs that can be send to the scheduler at the same time (0 for no limit)	0
drmaa_engine	str	The engine to use (options: grid_engine, torque	'grid_engine'
drmaa_job_check_interval	int	The interval in which the job checker will startto check for stale jobs	900

LinearExecution

An execution engine that has a background thread that executes the jobs in order. The queue is a simple FIFO queue and there is one worker thread that operates in the background. This plugin is meant as a fallback when other plugins do not function properly. It does not multi-processing so it is safe to use in environments that do no support that.

ProcessPoolExecution

A local execution plugin that uses multiprocessing to create a pool of worker processes. This allows fastr to execute jobs in parallel with true concurrency. The number of workers can be specified in the fastr configuration, but the default amount is the `number of cores - 1` with a minimum of `1`.

> **Warning:** The ProcessPoolExecution does not check memory requirements of jobs and running many workers might lead to memory starvation and thus an unresponsive system.

Configuration fields

name	type	description	default
process_pool_worker_number	int	Number of workers to use in a process pool	3

RQExecution

A execution plugin based on Redis Queue. Fastr will submit jobs to the redis queue and workers will peel the jobs from the queue and process them.

This system requires a running redis database and the database url has to be set in the fastr configuration.

> **Note:** This execution plugin required the `redis` and `rq` packages to be installed before it can be loaded properly.

Configuration fields

name	type	description	default
rq_host	str	The url of the redis serving the redis queue	'redis://localhost:6379/0'
rq_queue	str	The redis queue to use	'default'

SlurmExecution

Configuration fields

name	type	description	default
slurm_job_check_interval	int	The interval in which the job checker will startto check for stale jobs	30
slurm_partition	str	The slurm partition to use	''

StrongrExecution

A execution plugin based on Redis Queue. Fastr will submit jobs to the redis queue and workers will peel the jobs from the queue and process them.

This system requires a running redis database and the database url has to be set in the fastr configuration.

> **Note:** This execution plugin required the `redis` and `rq` packages to be installed before it can be loaded properly.

1.6.3 FlowPlugin Reference

Plugin that can manage an advanced data flow. The plugins override the execution of node. The execution receives all data of a node in one go, so not split per sample combination, but all data on all inputs in one large payload. The flow plugin can then re-order the data and create resulting samples as it sees fits. This can be used for all kinds of specialized data flows, e.g. cross validation.

To create a new FlowPlugin there is only one method that needs to be implemented: `execute`.

scheme	FlowPlugin
CrossValidation	*CrossValidation*

CrossValidation

Advanced flow plugin that generated a cross-validation data flow. The node need an input with data and an input number of folds. Based on that the outputs test and train will be supplied with a number of data sets.

1.6.4 IOPlugin Reference

IOPlugins are used for data import and export for the sources and sinks. The main use of the *IOPlugins* is during execution (see *Execution*). The *IOPlugins* can be accessed via `fastr.ioplugins`, but generally there should be no need for direct interaction with these objects. The use of is mainly via the URL used to specify source and sink data.

scheme	IOPlugin
CommaSeperatedValueFile	*CommaSeperatedValueFile*
FileSystem	*FileSystem*
HTTPPlugin	*HTTPPlugin*
Null	*Null*
Reference	*Reference*
S3Filesystem	*S3Filesystem*
VirtualFileSystem	*VirtualFileSystem*
VirtualFileSystemRegularExpression	*VirtualFileSystemRegularExpression*
VirtualFileSystemValueList	*VirtualFileSystemValueList*
XNATStorage	*XNATStorage*

CommaSeperatedValueFile

The CommaSeperatedValueFile an expand-only type of IOPlugin. No URLs can actually be fetched, but it can expand a single URL into a larger amount of URLs.

The `csv://` URL is a `vfs://` URL with a number of query variables available. The URL mount and path should point to a valid CSV file. The query variable then specify what column(s) of the file should be used.

The following variable can be set in the query:

variable	usage
value	the column containing the value of interest, can be int for index or string for key
id	the column containing the sample id (optional)
header	indicates if the first row is considered the header, can be `true` or `false` (optional)
delimiter	the delimiter used in the csv file (optional)
quote	the quote character used in the csv file (optional)
reformat	a reformatting string so that `value = reformat.format(value)` (used before relative_path)
relative_path	indicates the entries are relative paths (for files), can be `true` or `false` (optional)

The header is by default `false` if the neither the `value` and `id` are set as a string. If either of these are a string, the header is required to define the column names and it automatically is assumed `true`

The delimiter and quota characters of the file should be detected automatically using the `Sniffer`, but can be forced by setting them in the URL.

Example of valid `csv` URLs:

```
# Use the first column in the file (no header row assumed)
csv://mount/some/dir/file.csv?value=0

# Use the images column in the file (first row is assumed header row)
csv://mount/some/dir/file.csv?value=images

# Use the segmentations column in the file (first row is assumed header row)
# and use the id column as the sample id
csv://mount/some/dir/file.csv?value=segmentations&id=id

# Use the first column as the id and the second column as the value
# and skip the first row (considered the header)
csv://mount/some/dir/file.csv?value=1&id=0&header=true

# Use the first column and force the delimiter to be a comma
csv://mount/some/dir/file.csv?value=0&delimiter=,
```

FileSystem

The FileSystem plugin is create to handle `file://` type or URLs. This is generally not a good practice, as this is not portable over between machines. However, for test purposes it might be useful.

The URL scheme is rather simple: `file://host/path` (see wikipedia for details)

We do not make use of the `host` part and at the moment only support localhost (just leave the host empty) leading to `file:///` URLs.

> **Warning:** This plugin ignores the hostname in the URL and does only accept driver letters on Windows in the form `c:/`

HTTPPlugin

> **Warning:** This Plugin is still under development and has not been tested at all. example url: https://server.io/path/to/resource

Null

The Null plugin is create to handle `null://` type or URLs. These URLs are indicating the sink should not do anything. The data is not written to anywhere. Besides the scheme, the rest of the URL is ignored.

Reference

The Reference plugin is create to handle `ref://` type or URLs. These URLs are to make the sink just write a simple reference file to the data. The reference file contains the DataType and the value so the result can be reconstructed. It for files just leaves the data on disk by reference. This plugin is not useful for production, but is used for testing purposes.

S3Filesystem

> **Warning:** As this IOPlugin is under development, it has not been thoroughly tested.
>
> example url: s3://bucket.server/path/to/resource

VirtualFileSystem

The virtual file system class. This is an IOPlugin, but also heavily used internally in fastr for working with directories. The VirtualFileSystem uses the `vfs://` url scheme.

A typical virtual filesystem url is formatted as `vfs://mountpoint/relative/dir/from/mount.ext`

Where the `mountpoint` is defined in the *Config file*. A list of the currently known mountpoints can be found in the `fastr.config` object

```
>>> fastr.config.mounts
{'example_data': '/home/username/fastr-feature-documentation/fastr/fastr/examples/data
→',
 'home': '/home/username/',
 'tmp': '/home/username/FastrTemp'}
```

This shows that a url with the mount `home` such as `vfs://home/tempdir/testfile.txt` would be translated into `/home/username/tempdir/testfile.txt`.

There are a few default mount points defined by Fastr (that can be changed via the config file).

mountpoint	default location
home	the users home directory (`expanduser('~/')`)
tmp	the fastr temprorary dir, defaults to `tempfile.gettempdir()`
example_data	the fastr example data directory, defaults `$FASTRDIR/example/data`

VirtualFileSystemRegularExpression

The VirtualFileSystemValueList an expand-only type of IOPlugin. No URLs can actually be fetched, but it can expand a single URL into a larger amount of URLs.

A `vfsregex://` URL is a vfs URL that can contain regular expressions on every level of the path. The regular expressions follow the `re module` definitions.

An example of a valid URLs would be:

```
vfsregex://tmp/network_dir/.*/.*/__fastr_result__.pickle.gz
vfsregex://tmp/network_dir/nodeX/(?P<id>.*)/__fastr_result__.pickle.gz
```

The first URL would result in all the `__fastr_result__.pickle.gz` in the working directory of a Network. The second URL would only result in the file for a specific node (nodeX), but by adding the named group `id` using `(?P<id>.*)` the sample id of the data is automatically set to that group (see Regular Expression Syntax under the special characters for more info on named groups in regular expression).

Concretely if we would have a directory `vfs://mount/somedir` containing:

```
image_1/Image.nii
image_2/image.nii
image_3/anotherimage.nii
image_5/inconsistentnamingftw.nii
```

we could match these files using `vfsregex://mount/somedir/(?P<id>image_\d+)/.*\.nii` which would result in the following source data after expanding the URL:

```
{'image_1': 'vfs://mount/somedir/image_1/Image.nii',
 'image_2': 'vfs://mount/somedir/image_2/image.nii',
 'image_3': 'vfs://mount/somedir/image_3/anotherimage.nii',
 'image_5': 'vfs://mount/somedir/image_5/inconsistentnamingftw.nii'}
```

Showing the power of this regular expression filtering. Also it shows how the ID group from the URL can be used to have sensible sample ids.

> **Warning:** due to the nature of regexp on multiple levels, this method can be slow when having many matches on the lower level of the path (because the tree of potential matches grows) or when directories that are parts of the path are very large.

VirtualFileSystemValueList

The VirtualFileSystemValueList an expand-only type of IOPlugin. No URLs can actually be fetched, but it can expand a single URL into a larger amount of URLs. A `vfslist://` URL basically is a url that points to a file using vfs. This file then contains a number lines each containing another URL.

If the contents of a file `vfs://mount/some/path/contents` would be:

```
vfs://mount/some/path/file1.txt
vfs://mount/some/path/file2.txt
vfs://mount/some/path/file3.txt
vfs://mount/some/path/file4.txt
```

Then using the URL `vfslist://mount/some/path/contents` as source data would result in the four files being pulled.

Note: The URLs in a vfslist file do not have to use the `vfs` scheme, but can use any scheme known to the Fastr system.

XNATStorage

Warning: As this IOPlugin is under development, it has not been thoroughly tested.

The XNATStorage plugin is an IOPlugin that can download data from and upload data to an XNAT server. It uses its own `xnat://` URL scheme. This is a scheme specific for this plugin and though it looks somewhat like the XNAT rest interface, a different type or URL.

Data resources can be access directly by a data url:

```
xnat://xnat.example.com/data/archive/projects/sandbox/subjects/subject001/experiments/
↪experiment001/scans/T1/resources/DICOM
xnat://xnat.example.com/data/archive/projects/sandbox/subjects/subject001/experiments/
↪*_BRAIN/scans/T1/resources/DICOM
```

In the second URL you can see a wildcard being used. This is possible at long as it resolves to exactly one item.

The `id` query element will change the field from the default experiment to subject and the `label` query element sets the use of the label as the fastr id (instead of the XNAT id) to `True` (the default is `False`)

To disable `https` transport and use `http` instead the query string can be modified to add `insecure=true`. This will make the plugin send requests over `http`:

```
xnat://xnat.example.com/data/archive/projects/sandbox/subjects/subject001/experiments/
↪*_BRAIN/scans/T1/resources/DICOM?insecure=true
```

For sinks it is import to know where to save the data. Sometimes you want to save data in a new assessor/resource and it needs to be created. To allow the Fastr sink to create an object in XNAT, you have to supply the type as a query parameter:

```
xnat://xnat.bmia.nl/data/archive/projects/sandbox/subjects/S01/experiments/_BRAIN/
↪assessors/test_assessor/resources/IMAGE/files/image.nii.gz?resource_
↪type=xnat:resourceCatalog&assessor_type=xnat:qcAssessmentData
```

Valid options are: subject_type, experiment_type, assessor_type, scan_type, and resource_type.

If you want to do a search where multiple resources are returned, it is possible to use a search url:

```
xnat://xnat.example.com/search?projects=sandbox&subjects=subject[0-9][0-9][0-9]&
↪experiments=*_BRAIN&scans=T1&resources=DICOM
```

This will return all DICOMs for the T1 scans for experiments that end with _BRAIN that belong to a subject-tXXX where XXX is a 3 digit number. By default the ID for the samples will be the experiment XNAT ID (e.g. XNAT_E00123). The wildcards that can be the used are the same UNIX shell-style wildcards as provided by the module `fnmatch`.

It is possible to change the id to a different fields id or label. Valid fields are project, subject, experiment, scan, and resource:

```
xnat://xnat.example.com/search?projects=sandbox&subjects=subject[0-9][0-9][0-9]&
↪experiments=*_BRAIN&scans=T1&resources=DICOM&id=subject&label=true
```

The following variables can be set in the search query:

variable	default	usage
projects	`*`	The project(s) to select, can contain wildcards (see `fnmatch`)
subjects	`*`	The subject(s) to select, can contain wildcards (see `fnmatch`)
experiments	`*`	The experiment(s) to select, can contain wildcards (see `fnmatch`)
scans	`*`	The scan(s) to select, can contain wildcards (see `fnmatch`)
resources	`*`	The resource(s) to select, can contain wildcards (see `fnmatch`)
id	`experiment`	What field to use a the id, can be: project, subject, experiment, scan, or resource
label	`false`	Indicate the XNAT label should be used as fastr id, options `true` or `false`
insecure	`false`	Change the url scheme to be used to http instead of https
verify	`true`	(Dis)able the verification of SSL certificates
regex	`false`	Change search to use regex `re.match()` instead of fnmatch for matching
overwrite	`false`	Tell XNAT to overwrite existing files if a file with the name is already present

For storing credentials the `.netrc` file can be used. This is a common way to store credentials on UNIX systems. It is required that the file is only accessible by the owner only or a `NetrcParseError` will be raised. A netrc file is really easy to create, as its entries look like:

```
machine xnat.example.com
        login username
        password secret123
```

See the `netrc module` or the GNU inet utils website for more information about the `.netrc` file.

Note: On windows the location of the netrc file is assumed to be `os.path.expanduser('~/_netrc')`. The leading underscore is because windows does not like filename starting with a dot.

Note: For scan the label will be the scan type (this is initially the same as the series description, but can be updated manually or the XNAT scan type cleanup).

Warning: labels in XNAT are not guaranteed to be unique, so be careful when using them as the sample ID.

For background on XNAT, see the XNAT API DIRECTORY for the REST API of XNAT.

1.6.5 Interface Reference

Abstract base class of all Interfaces. Defines the minimal requirements for all Interface implementations.

scheme	`Interface`
FastrInterface	*FastrInterface*
FlowInterface	*FlowInterface*
NipypeInterface	*NipypeInterface*

FastrInterface

The default Interface for fastr. For the command-line Tools as used by fastr. It build a commandline call based on the input/output specification.

The fields that can be set in the interface:

Attribute		Description
`id`		The id of this Tool (used internally in fastr)
`inputs[]`		List of Inputs that can are accepted by the Tool
	`id`	ID of the Input
	`name`	Longer name of the Input (more human readable)
	`datatype`	The ID of the DataType of the Input[1]
	`enum[]`	List of possible values for an Enum-Type (created on the fly by fastr)[1]
	`prefix`	Commandline prefix of the Input (e.g. –in, -i)
	`cardinality`	Cardinality of the Input
	`repeat_prefix`	Flag indicating if for every value of the Input the prefix is repeated
	`required`	Flag indicating if the input is required
	`nospace`	Flag indicating if there is no space between prefix and value (e.g. –in=val)
	`format`	For DataTypes that have multiple representations, indicate which one to use
	`default`	Default value for the Input
	`description`	Long description for an input
`outputs[]`		List of Outputs that are generated by the Tool (and accessible to fastr)
	`id`	ID of the Output
	`name`	Longer name of the Output (more human readable)
	`datatype`	The ID of the DataType of the Output[1]
	`enum[]`	List of possible values for an Enum-Type (created on the fly by fastr)[1]
	`prefix`	Commandline prefix of the Output (e.g. –out, -o)
	`cardinality`	Cardinality of the Output
	`repeat_prefix`	Flag indicating if for every value of the Output the prefix is repeated
	`required`	Flag indicating if the input is required
	`nospace`	Flag indicating if there is no space between prefix and value (e.g. –out=val)

Table 1.1 – continued from previous page

Attribute		Description
	`format`	For DataTypes that have multiple representations, indicate which one to use
	`description`	Long description for an input
	`action`	Special action (defined per DataType) that needs to be performed before creating output value (e.g. 'ensure' will make sure an output directory exists)
	`automatic`	Indicate that output doesn't require commandline argument, but is created automatically by a Tool[2]
	`method`	The collector plugin to use for the gathering automatic output, see the *Collector plugins*
	`location`	Definition where to an automatically, usage depends on the `method`[2]

FlowInterface

The Interface use for AdvancedFlowNodes to create the advanced data flows that are not implemented in the fastr. This allows nodes to implement new data flows using the plugin system.

The definition of `FlowInterfaces` are very similar to the default `FastrInterfaces`.

Note: A flow interface should be using a specific FlowPlugin

NipypeInterface

Experimental interfaces to using `nipype` interfaces directly in fastr tools, only using a simple reference.

To create a tool using a nipype interface just create an interface with the correct type and set the `nipype` argument to the correct class. For example in an xml tool this would become:

```
<interface class="NipypeInterface">
  <nipype_class>nipype.interfaces.elastix.Registration</nipype_class>
</interface>
```

Note: To use these interfaces `nipype` should be installed on the system.

Warning: This interface plugin is basically functional, but highly experimental!

---

[1] `datatype` and `enum` are conflicting entries, if both specified `datatype` has presedence
[2] More details on defining automatica output are given in [TODO]

1.6.6 ReportingPlugin Reference

Base class for all reporting plugins. The plugin has a number of methods that can be implemented that will be called on certain events. On these events the plugin can inspect the presented data and take reporting actions.

scheme	ReportingPlugin
ElasticsearchReporter	*ElasticsearchReporter*
PimReporter	*PimReporter*
SimpleReport	*SimpleReport*

ElasticsearchReporter

NOT DOCUMENTED!

Configuration fields

name	type	description	default
elasticsearch_host	str	The elasticsearch host to report to	''
elasticsearch_index	str	The elasticsearch index to store data in	'fastr'
elasticsearch_debug	bool	Setup elasticsearch debug mode to send stdout stderr on job succes	False

PimReporter

NOT DOCUMENTED!

Configuration fields

name	type	description	default
pim_host	str	The PIM host to report to	''
pim_update_interval	float	The interval in which to send jobs to PIM	2.5
pim_batch_size	int	Maximum number of jobs that can be send to PIM in a single interval	100
pim_debug	bool	Setup PIM debug mode to send stdout stderr on job succes	False

SimpleReport

NOT DOCUMENTED!

1.6.7 Target Reference

The abstract base class for all targets. Execution with a target should follow the following pattern:

```
>>> with Target() as target:
...     target.run_commmand(['sleep', '10'])
```

The Target context operator will set the correct paths/initialization. Within the context command can be ran and when leaving the context the target reverts the state before.

scheme	Target
DockerTarget	*DockerTarget*
LocalBinaryTarget	*LocalBinaryTarget*
MacroTarget	*MacroTarget*
SingularityTarget	*SingularityTarget*

DockerTarget

A tool target that is located in a Docker images. Can be run using docker-py. A docker target only need two variables: the binary to call within the docker container, and the docker container to use.

```
{
  "arch": "*",
  "os": "*",
  "binary": "bin/test.py",
  "docker_image": "fastr/test"
}
```

```
<target os="*" arch="*" binary="bin/test.py" docker_image="fastr/test">
```

LocalBinaryTarget

A tool target that is a local binary on the system. Can be found using environmentmodules or a path on the executing machine. A local binary target has a number of fields that can be supplied:

- `binary (required)`: the name of the binary/script to call, can also be called `bin` for backwards compatibility.

- `modules`: list of modules to load, this can be environmentmodules or lmod modules. If modules are given, the `paths`, `environment_variables` and `initscripts` are ignored.

- `paths`: a list of paths to add following the structure `{"value": "/path/to/dir", "type": "bin"}`. The types can be `bin` if the it should be added to $PATH or `lib` if it should be added to te library path (e.g. $LD_LIBRARY_PATH for linux).

- `environment_variables`: a dictionary of environment variables to set.

- `initscript`: a list of script to run before running the main tool

- `interpreter`: the interpreter to use to call the binary e.g. `python`

The LocalBinaryTarget will first check if there are modules given and the module subsystem is loaded. If that is the case it will simply unload all current modules and load the given modules. If not it will try to set up the environment itself by using the following steps:

1. Prepend the bin paths to $PATH

2. Prepend the lib paths to the correct environment variable

3. Setting the other environment variables given ($PATH and the system library path are ignored and cannot be set that way)

4. Call the initscripts one by one

The definition of the target in JSON is very straightforward:

```json
{
  "binary": "bin/test.py",
  "interpreter": "python",
  "paths": [
    {
      "type": "bin",
      "value": "vfs://apps/test/bin"
    },
    {
      "type": "lib",
      "value": "./lib"
    }
  ],
  "environment_variables": {
    "othervar": 42,
    "short_var": 1,
    "testvar": "value1"
  },
  "initscripts": [
    "bin/init.sh"
  ],
  "modules": ["elastix/4.8"]
}
```

In XML the definition would be in the form of:

```xml
<target os="linux" arch="*" modules="elastix/4.8" bin="bin/test.py" interpreter=
↪"python">
  <paths>
    <path type="bin" value="vfs://apps/test/bin" />
    <path type="lib" value="./lib" />
  </paths>
  <environment_variables short_var="1">
    <testvar>value1</testvar>
    <othervar>42</othervar>
  </environment_variables>
  <initscripts>
    <initscript>bin/init.sh</initscript>
  </initscripts>
</target>
```

MacroTarget

A target for MacroNodes. This target cannot be executed as the MacroNode handles execution differently. But this contains the information for the MacroNode to find the internal Network.

SingularityTarget

A tool target that is run using a singularity container, see the singulary website

- `binary (required):` the name of the binary/script to call, can also be called `bin` for backwards compatibility.

- **`container (required):` the singularity container to run, this can be in url form for singularity** pull or as a path to a local container

- `interpreter`: the interpreter to use to call the binary e.g. `python`

1.7 Development and Design Documentation

In this chapter we will discuss the design of Fastr in more detail. We give pointers for development and add the design documents as we currently envision Fastr. This is both for people who are interested in the Fastr develop and for current developers to have an archive of the design decision agreed upon.

1.7.1 Sample flow in Fastr

The current Sample flow is the following:

Output	**ContainsSamples**	
SubOutput	**ForwardsSamples**	selects cardinality
Link	**ForwardsSamples**	collapse + expand (changes cardinality and dimensions)
SubInput	**ForwardsSamples**	direct forward
Input	**ForwardsSamples**	broadcast matching (combine samples in cardinality)
InputGroup	**ForwardsSamples**	broadcast matching (combine samples in payload)
NodeRun	**ForwardsSamples**	combines payloads (plugin based, e.g. cross product)

The idea is that we make a common interface for all classes that are related to the flow of Samples. For this we propose the following mixin classes that provide the interface and allow for better code sharing. The basic structure

of the classes is given in the following diagram:

```
           ┌─────────────────────┐
           │   HasDimensions     │
           ├─────────────────────┤
           │    dimensions       │
           ├─────────────────────┤
           │ + size              │
           │ + dimnames          │
           └─────────────────────┘
                     △
                     │
           ┌─────────────────────┐
           │    HasSamples       │
           ├─────────────────────┤
           │   __getitem__()     │
           ├─────────────────────┤
           │ + __contains__      │
           │ + __iter__          │
           │ + iteritems()       │
           │ + items()           │
           │ + indexes           │
           │ + ids               │
           └─────────────────────┘
              △              △
             /                \
┌─────────────────────┐   ┌─────────────────────┐
│  ContainsSamples    │   │  ForwardsSamples    │
├─────────────────────┤   ├─────────────────────┤
│     samples         │   │ source              │
├─────────────────────┤   │ index_to_target     │
│ + __getitem__()     │   │ index_to_source     │
│ + __setitem__()     │   │ combine_samples     │
│ + dimensions        │   │ combine_dimensions  │
└─────────────────────┘   ├─────────────────────┤
                          │ + __getitem__       │
                          │ + dimensions        │
                          └─────────────────────┘
```
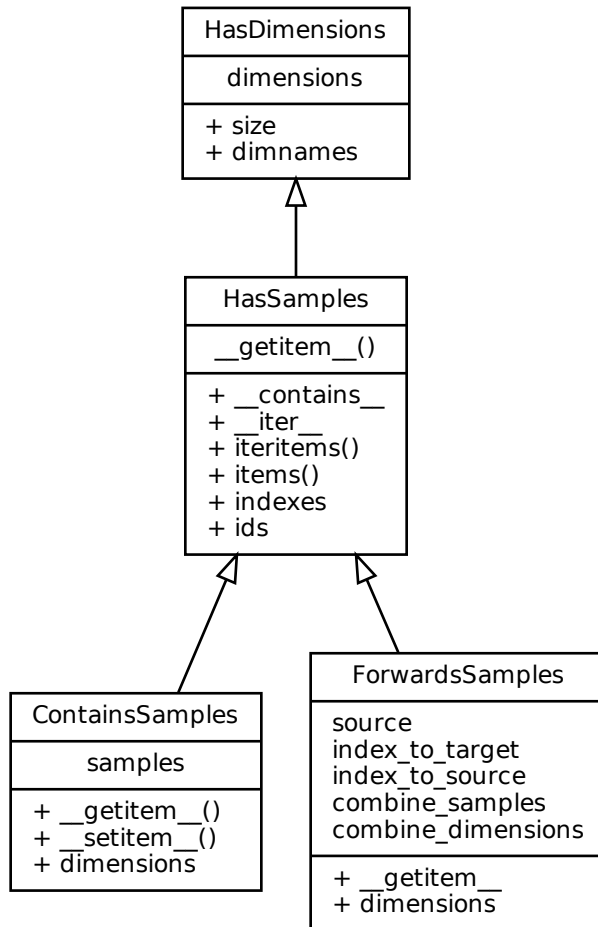
The abstract and mixin methods are as follows:

ABC	Inherits from	Abstract Methods	Mixin methods
`HasDimensions`		`dimensions`	`size` `dimnames`

`HasSamples`	`HasDimensions`	`__getitem__`	`__contains__`
			`__iter__`
			`iteritems`
			`items`
			`indexes`
			`ids`

`ContainsSamples`	`HasSamples`	`samples`	`__getitem__`
			`__setitem__`
			`dimensions`

`ForwardsSamples`	`HasSamples`	`source`
		`index_to_target`
		`index_to_source`
		`combine_samples`
		
		`combine_dimensions`
			`__getitem__`
			`dimensions`

Note: Though the flow is currently working like this, the mixins are not yet created.

1.7.2 Network Execution

The network execution should contain a number of steps:

- `Network`
  - Creates a `NetworkRun` based on the current layout
- `NetworkRun`
  - Transform the `Network` (possibly joining Nodes of certain interface into a combined NodeRun etc)
  - Start generation of the Job Direct Acyclic Graph (DAG)
- `SchedulingPlugin`
  - Prioritize Jobs based on some predefined rules
  - Combine certain `Jobs` to improve efficiency (e.g. minimize i/o on a grid)
- `ExecutionPlugin`

- Run a (list of) `Jobs`. If there is more than one jobs, run them sequentially on same execution host using a local temp for intermediate files.

- On finished callback: Updated DAG with newly ready jobs, or remove cancelled jobs

This could be visualized as the following loop:

```
                    ┌──────────────┐
                    │   Network    │
                    └──────────────┘
                           │
                         creates
                           ▼
                    ┌──────────────┐
                    │  NetworkRun  │◄──────┐
                    └──────────────┘       │
                     │          │          │
                  executes    creates      │
                     ▼          │          │
              ┌──────────┐      │          │
              │ NodeRun  │      │       callback
              └──────────┘      │          │
                   │            │          │
                adds jobs       ▼          │
                   ▼     ┌──────────┐      │
                         │  JobDAG  │      │
                         └──────────┘      │
                              │            │
                     analyzes and selects jobs
                              ▼            │
                  ┌──────────────────┐     │
                  │ SchedulingPlugin │     │
                  └──────────────────┘     │
                              │            │
                      (list of) Jobs to execute
                              ▼            │
                  ┌──────────────────┐     │
                  │ ExecutionPlugin  │─────┘
                  └──────────────────┘
```
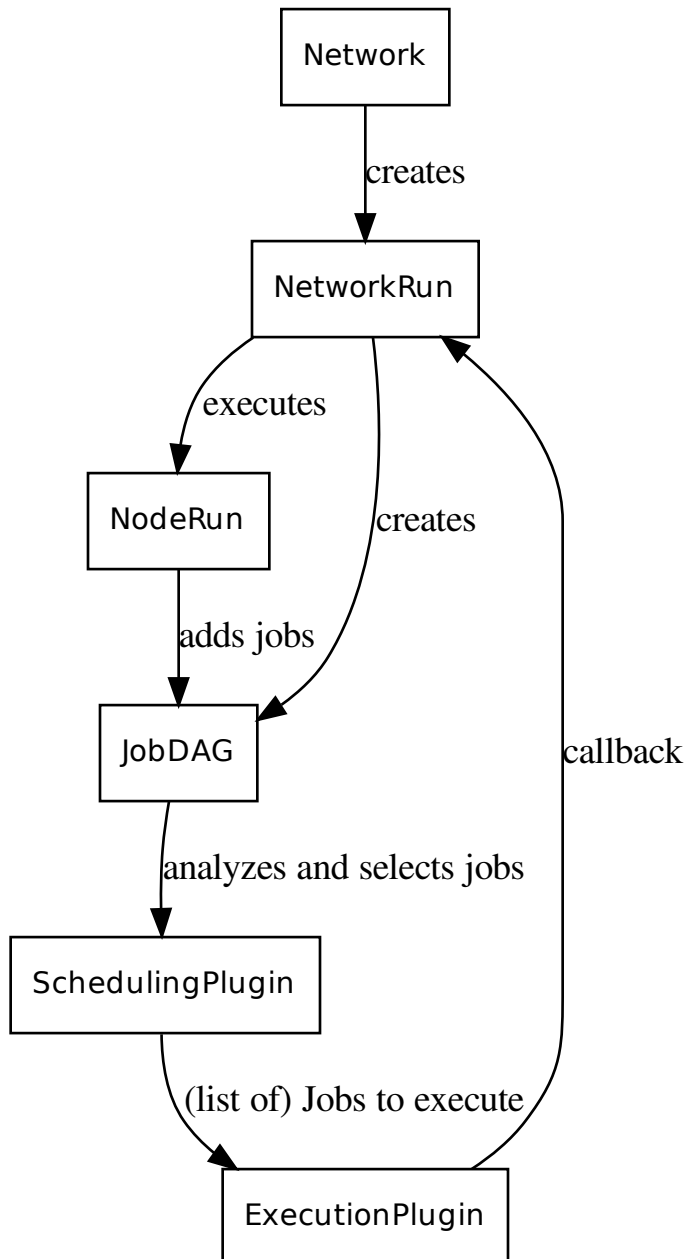
The callback of the `ExecutionPlugin` to the `NetworkRun` would trigger the execution of the relevant `NodeRuns` and the addition of more `Jobs` to the `JobDAG`.

Note: The Job DAG should be thread-safe as it could be both read and extended at the same time.

Note: If a list of jobs is send to the `ExecutionPlugin` to be run as on Job on an external execution platform, the resources should be combined as follows: memory=max, cores=max, runtime=sum

---

Note: If there are execution hosts that have mutliple cores the `ExecutionPlugin` should manage this (for example by using pilot jobs). The `SchedulingPlugin` creates units that should be run sequentially on the resources noted and will not attempt parallelization

---

A `NetworkRun` would be contain similar information as the `Network` but not have functionality for editing/changing it. It would contain the functionality to execute the Network and track the status and samples. This would allow `Network.execute` to create multiple concurrent runs that operate indepent of each other. Also editing a `Network` after the run started would have no effect on that run.

Note: This is a plan, not yet implemented

Note: For this to work, it would be important for a Jobs to have forward and backward dependency links.

SchedulingPlugins

The idea of the plugin is that it would give a priority on Jobs created by a `Network`. This could be done based on different strategies:

- Based on (sorted) sample id's, so that one sample is always prioritized over others. The idea is that samples are process as much as possible in order, finishing the first sample first. Only processing other samples if there is left-over capacity.

- Based on distance to a (particular) `Sink`. This is to generate specific results as quick as possible. It would not focus on specific samples, but give priority to whatever sample is closest to being finished.

- Based on the distance to from a `Souce`. Based on the sign of the weight it would either keep all samples on the same stage as much as possible, only progressing to a new `NodeRun` when all samples are done with the previous `NodeRun`, or it would push samples with accelerated rates.

Additionally it will group `Jobs` to be executed on a single host. This could reduce i/o and limited the number of jobs an external scheduler has to track.

Note: The interface for such a plugin has not yet been established.

1.7.3 Secrets

"Something that is kept or meant to be kept unknown or unseen by others."

Using secrets

Fastr IOPlugins that need authentication data should use the Fastr SecretService for retrieving such data. The SecretService can be used as follows.

```python
from fastr.utils.secrets import SecretService
from fastr.utils.secrets.exceptions import CouldNotRetrieveCredentials

secret_service = SecretService()

try:
  password = secret_service.find_password_for_user('testserver.lan:9000', 'john-doe')
except CouldNotRetrieveCredentials:
  # the password was not found
  pass
```

Implementing a SecretProvider

A SecretProvider is implemented as follows:

1. Create a file in fastr/utils/secrets/providers/<yourprovidername>.py

2. Use the template below to write your SecretProvider

3. Add the secret provider to fastr/utils/secrets/providers/__init__.py

4. Add the secret provider to fastr/utils/secrets/secretservice.py: import it and add it to the array in function _init_providers

```python
from fastr.utils.secrets.secretprovider import SecretProvider
from fastr.utils.secrets.exceptions import CouldNotRetrieveCredentials,␣
↪CouldNotSetCredentials, CouldNotDeleteCredentials, NotImplemented


try:
  # this is where libraries can be imported
  # we don't want fastr to crash if a specific
  # library is unavailable
  # import my-libary
except (ImportError, ValueError) as e:
  pass

class KeyringProvider(SecretProvider):
  def __init__(self):
    # if libraries are imported in the code above
    # we need to check if import was succesfull
    # if it was not, raise a RuntimeError
    # so that FASTR ignores this SecretProvider
    # if 'my-library' not in globals():
    #   raise RuntimeError("my-library module required")
    pass

  def get_password_for_user(self, machine, username):
    # This function should return the password as a string
    # or raise a CouldNotRetrieveCredentials error if the password
    # is not found.
    # In the event that this function is unsupported a
    # NotImplemented exception should be thrown
```

(continues on next page)

```python
        raise NotImplemented()

    def set_password_for_user(self, machine, username, password):
        # This function should set the password for a specified
        # machine + user. If anything goes wrong while setting
        # the password a CouldNotSetCredentials error should be raised.
        # In the event that this function is unsupported a
        # NotImplemented exception should be thrown
        raise NotImplemented()

    def del_password_for_user(self, machine, username):
        # This function should delete the password for a specified
        # machine + user. If anything goes wrong while setting
        # the password a CouldNotDeleteCredentials error should be raised.
        # In the event that this function is unsupported a
        # NotImplemented exception should be thrown
        raise NotImplemented()
```

1.8 Changelog

All notable changes to this project will be documented in this file.

The format is based on Keep a Changelog and this project adheres to Semantic Versioning

1.8.1 3.0.0 - 2019-03-05

Changed

- Now ported to Python 3.6+ (Python 2 is no longer supported!)

- New public API which is not fully compatible with fastr 2.x, the changes are small. The new API will be guaranteed in next minor version upgrades and is considered to be stable.

- Clear way of defining resource limits for Nodes in a Network using the ResourceLimit class.

- The datatype and cardinality of inputs of a tool are now checked before the tool is to be executed as an extra safety.

- Dimensions are drawn by default in network.draw

- The api now accepts types other than Output, list, tuple when creating a link. When a single value is given it is assumedly a constant from the network definition.

- Drawing a network will not create temporary .dot files anymore

- Sinkdata can be a string, it that case it will be the same string for all sink nodes so a {node} substitution should be used in the template

- Make the xnat ioplugin use xnat+http:// and xnat+https:// url schemes in favour of xnat:// with ?insecure=... (old behaviour will also work for now)

- Complete rewrite of PIM plugin (PIMReporter) making use of the new Reporter plugin infrastructure. It also caches all communication with PIM to be resilient against connection interruptions.

Added

- `fastr upgrade` command to automatically upgrade a network creation file from fastr 2.x to fastr 3.x API.

- http(s) IOPlugin for downloading files via http(s)

- network.draw now has a flag to hide the unconnected inputs and output of a node. The unconnected inputs/outputs are hidden by default.

- Reporting plugins, Fastr now exposes a number of message hooks which can be listened to by Reporter plugins.

Fixed

- Fixed some bugs with drmaa communication (more safeties added)

- Fixed a bug in the MacroNode update function which could cause networks with MacroNodes to be invalid

- The margins and font size of the network.draw graph rendering are set a bit wider and smaller (resp.) to avoid excessive text overflow.

- Fixed bug in provenance which did not properly chain the provenance of subsequent jobs.

1.8.2 2.1.2 - 2018-10-24

Added

- Allow overriding the timestamp of the network execution

Changed

- Updated PIM publisher to support the new PIM API v2

- Updated XNAT IOPlugin to not crash when creating a resource failed because another process already did that (race condition)

- Make default resource limits for DRMAA configurable

- Add stack trace to FastrExceptions

1.8.3 2.1.1 - 2018-06-29

Fixed

- Fixed some issues with the type estimation of outputs of Jobs and update validation functions of NIFTI files

1.8.4 2.1.0 - 2018-04-13

Added

- SLURM execution plugin based on `sbatch`, `scancel`, `scontrol` and `squeue`. The plugin supports job dependencies and cancellation.

- Support for running tools in Docker containers using a DockerTarget

- Support for running tools in Singularity containers using a SingularityTarget

- Support for datatypes with multiple extensions (e.g. .tif and .tiff) by setting the extension to a tuple of options. The first extension is leading for deciding filenames in a sink.

Changed

- Source jobs now also validate the output (and do not only rely on the stderr of the tool)

- Added preferred_types attribute to TypeGroups that gives the order of preference of members, alternatively the order of _members is used (this should be given as tuple or list to be meaningful)

- In the config.py you can now access the USER_DIR and SYSTEM_DIR variables for use in setting other variables. These are only read and changing them will only change subsequent config reads but not the main config values.

- checksum for nii.gz now takes the md5 checksum of the decompressed data

- Serialization of MacroNodes now should function properly

Fixed

- BUG in XNAT plugin that made it impossible to download data from scans without an empty type string

- BUG where the order of OrderedDict in a source was not preserved

- BUG where newer Werkzeug version requires the web port to be an integer

1.8.5 2.0.1 - 2017-10-19

- Fix a bug in the validation of FilePrefix datatypes

1.8.6 2.0.0 - 2017-09-28

Added

- The default python logger can now be configured from the fastr config file under key logging_config

- Support for MacroNodes, a Network can be used as a Node inside of another Network. There is should be no limitation on the internal Network used, but currently the MacroNode ignores input_groups on its inputs.

- A sync helper was added to assist in slow file synchronisation over NFS

- Source and Sink can now handle S3 URL's

- FastrInterface can now forward errors from a subprocess if they are dumped to stdout or stderr in a json identified by `__FASTR_ERRORS__ = []`.

- A `specials.workdir` field in the location field of automatic outputs that gives the current working directory (e.g. job directory)

- Added support for Torque (using pbs-drmaa library) to DRMAAExecution

- Added option to set a limit for number of jobs submitted at same time be the DRMAAExecution

- Use of the ~/.fastr/config.d directory for adding additional config files. Any `.py` file in there will be parsed in alphabetical order.

- XNATStorage IOPlugin now has a retry scheme for uploads, if an uploaded file could not be found on the server, it is retried up to 3 times.

- Added `fastr dump` command to create a zip containing all important debugging information.

Changed

- FilePrefix type does not have an extension anymore (avoids ugly dot in middle of filename)

- Allow expanding of link where samples have a non-uniform cardinality. This will not result in a sparse array.

- The default for `required` for the automatic outputs is now `False`

- Removed `testtool` commandline subcommand in favour of the `test` subcommand which can test both Tools and Networks

- Moved nodegroup specification into the Node for speedup

Fixed

- Stop Jobs from failing when a non-required, non-requested output is invalid

- Bug in boolean value parsing in the Boolean datatype

- Bug in target that caused paths not to be expanded properly in some cases

- Made sure failed sources also create a sample so the failure becomes visible and traceable.

- Bug in XNAT IOPlugin that made download from XNAT seem to fail (while getting the correct data).

Removed

- `fastr.current_network` has been removed as it was deemed to "magical" and could change things out of the sight of the user.

1.8.7 1.2.2 - 2017-08-24

Fixed

- Fixed a bug breaking the XNAT IOPlugin due to an xnatpy version update.

1.8.8 1.2.1 - 2017-04-04

Added

- A FastrInterface can now specify a `negate` flag on an automatic output that also has a prefix, which will negate the flag. This is useful for flag the suppress the creation of an output (e.g. no_mask). An example is given in the Tool `fastr.util.AutoPrefixNegateTest`.

Changed

- The provenance and extra information of a Job now is not serialized in the Job, but exported to separate files next to the job file __fastr_prov__.json and __fastr_extra_job_info__.json which makes the information more accessible and reduces the memory footprint of the main process hugely as it will not read this information back anymore.

- Most execution plugin will not overwrite the executionscript stdout and stderr but rather append it. This is only relevant when continuing a run in the an existing temporary directory, but avoids loss of information.

Fixed

- Bug that stopped the `Link.append` function from returning the newly created link
- Bugs that caused some cardinality computations of the output to fail during execution
- Bug in the job.tmpurl that caused double slashes somewhere. Some tools chocked on this when it was used for parameters.

1.8.9 1.2.0 - 2017-03-15

Added

- Failed sample annotation: when a job fails, the result is annotated and forwarded until a SinkNode, where we can determine the status and possibly point of failure of the Sample.
- Commandline tool `fastr trace` that can inspect a workflow run and help trace errors and print debug information
- Supported for Lmod modules environment next to the old environmentmodules
- BaseDataType descendants are now (un)picklable (including EnumTypes)
- Option to use `{extension}` field in sink_data, which differs from `{ext}` in that it doesn't include a leading dot.
- Support for Docker targets. A Docker target will execute a command inside of a specified docker container, allowing Tools to use Docker for distribution
- Using the right and left shift operator (<< and >>) for creating links to `Inputs` using `input << output` or `output >> input`.
- In the FastrInterfaces, automatic outputs can have a prefix for a flag that should be set for the output to be actually generated.
- Fastr is now able to limit the amount of SourceJobs that are allowed to run concurrently.
- Ability to report progress to PIM (use the pim_host field in the config)

Changed

- Version can now also accept a format based on a date (e.g. 2017-02-17_bananas) which will be parsed the same way as 2017.02.17_bananas
- Work on the ExecutionPlugin and the corresponding API. Has better fall-backs and a mechanism to advertise plugin capabilities.
- The collector plugins have the `input` and `input_parts` fields merged, and the `output` and `output_parts` fields merged.

Fixed

- In some cases the log directory was not created properly, causing an handled exception
- A bug making the handling of Booleans incorrect for the FastrInterface, when a Boolean was given a flag would also appear when it was False
- Serialization of the namespace of a Network was not correct
- Check version of Fastr that creates and executes a Job against each other
- load_gpickle helper can handle data with Enums that use to cause an AttributeError
- Output validation of Jobs did not work correctly for automatic outputs

1.8.10 1.1.2 - 2016-12-22

Fixed

- The example network in resources/networks/add_ints.json was using an old serialization format making it non-functions. Replaced by a new network file.

1.8.11 1.1.1 - 2016-12-22

Fixed

- Network runs called from an interpreter (and not file) caused a crash because the network tried to report the file used. Better handling of these situations.

1.8.12 1.1.0 - 2016-12-08

Added

- Namespaces for resources (tools and networks)
- Network manager located at `fastr.networklist`
- RQExecution plugin. This plugin uses python-rq to manage a job queue.
- LinearExecution plugin. This plugin uses a background thread for execution.
- BlockingExecution plugin. This plugin executes jobs in a blocking fashion.
- Automatic generation of documentation for all plugins, the configuration fields and all commandline tools.

Changed

- Provenance is updated with a network dump and used tool definitions.
- New configuration system that uses python files
- New plugin system that integrates with the new configuration system and enables automatic importing of plugins
- The fastr command line tools now use an entrypoint which is located in `fastr.utils.cmd`. This code also dispatches the sub commands.

Removed

- fastr.config file. This is replaced by the config.py file. Go to the docs!

Fixed

- Adds explicit tool namespace and version to the provenance document.

FASTR User reference

2.1 Fastr User Reference

`fastr.`**`tools`**

> A ToolManager containing all versions of all Tools loaded into the FASTR environment. The ToolManager can be indexed using the Tool id string or a tool id string and a version. For example if you have two versions (4.5 and 4.8) of a tool called *Elastix*:

```
>>> fastr.tools['elastix.Elastix']
Tool Elastix v4.8 (Elastix Registration)
                        Inputs                                    |            ␣
↪Outputs
-------------------------------------------------------------------------------
↪----------------
fixed_image       (ITKImageFile)                                  | directory␣
↪(Directory)
moving_image      (ITKImageFile)                                  | transform␣
↪(ElastixTransformFile)
parameters        (ElastixParameterFile)                          | log_file ␣
↪(ElastixLogFile)
fixed_mask        (ITKImageFile)                                  |
moving_mask       (ITKImageFile)                                  |
initial_transform (ElastixTransformFile)                          |
priority          (__Elastix_4.8_interface__priority__Enum__)  |
threads           (Int)                                           |

>>> fastr.tools['elastix.Elastix', '4.5']
Tool Elastix v4.5 (Elastix Registration)
                        Inputs                                    |            ␣
↪Outputs
-------------------------------------------------------------------------------
↪----------------
fixed_image       (ITKImageFile)                                  | directory␣
↪(Directory)
```

(continues on next page)

```
moving_image     (ITKImageFile)                            | transform␣
↪(ElastixTransformFile)
parameters       (ElastixParameterFile)                    | log_file ␣
↪(ElastixLogFile)
fixed_mask       (ITKImageFile)                            |
moving_mask      (ITKImageFile)                            |
initial_transform (ElastixTransformFile)                   |
priority         (__Elastix_4.5_interface__priority__Enum__) |
threads          (Int)                                     |
```

fastr.**types**
> A dictionary containing all types loaded into the FASTR environment. The keys are the typenames and the values are the classes.

fastr.**networks**
> A dictionary containing all networks loaded in fastr

api.**create_network**(*version=None*)
> Create a new Network object
>
> > **Parameters**
> >
> > - **id** (`str`) – id of the network
> >
> > - **version** (`Union`[*Version*, `str`, `None`]) – version of the network
> >
> > **Return type** *Network*
> >
> > **Returns**

api.**create_network_copy**()
> Create a network based on another Network state. The network state can be a Network or the state gotten from a Network with __getstate__.
>
> > **Parameters** **network_state** (`Union`[*Network*, *Network*, `dict`]) – Network (state) to create a copy of
> >
> > **Return type** *Network*
> >
> > **Returns** The rebuilt network

class fastr.api.**Network**(*id*, *version=None*)
> Representation of a Network for the creating and adapting Networks
>
> **create_constant**(*datatype*, *data*, *id=None*, *step_id=None*, *resources=None*, *node_group=None*)
> > Create a ConstantNode in this Network. The Node will be automatically added to the Network.
> >
> > > **Parameters**
> > >
> > > - **datatype** (`Union`[*BaseDataType*, `str`]) – The DataType of the constant node
> > >
> > > - **data** (`Dict`[`str`, `Union`[`List`[`Union`[`str`, `Tuple`[`str`, …]]], `Dict`[`str`, `Union`[`str`, `Tuple`[`str`, …]]]]]) – The data to hold in the constant node
> > >
> > > - **id** (`Optional`[`str`]) – The id of the constant node to be created
> > >
> > > - **step_id** (`Optional`[`str`]) – The step to add the created constant node to
> > >
> > > - **resources** (`Optional`[*ResourceLimit*]) – The resources required to run this node
> > >
> > > - **node_group** (`Optional`[`str`]) – The group the node belongs to, this can be important for FlowNodes and such, as they will have matching dimension names.
> > >
> > > **Return type** *Node*

Chapter 2. FASTR User reference

Returns the newly created constant node

create_link (*source*, *target*, *id=None*, *collapse=None*, *expand=False*)

Create a link between two Nodes and add it to the current Network.

> **Parameters**
>
> - **source** (Union[*Input*, *BaseInput*]) – the output that is the source of the link
> - **target** (Union[*Output*, *BaseOutput*]) – the input that is the target of the link
> - **id** (Optional[str]) – the id of the link
> - **collapse** (Optional[Tuple[Union[int, str], . . .]]) – The dimensions to collapse in this link.
> - **expand** (bool) – Flag to expand cardinality into a new dimension
>
> **Return type** *Link*
>
> **Returns** the created link

create_macro (*network*, *id=None*)

Create macro node (a node which actually contains a network used as node inside another network).

> **Parameters**
>
> - **network** (Union[*Network*, *Network*, dict, *Tool*, str]) – The network to use, this can be a network (state), a macro tool, or the path to a python file that contains a function create_network which returns the desired network.
> - **id** (Optional[str]) – The id of the node to be created
>
> **Return type** *Node*
>
> **Returns** the newly created node

create_node (*tool*, *tool_version*, *id=None*, *step_id=None*, *resources=None*, *node_group=None*)

Create a Node in this Network. The Node will be automatically added to the Network.

> **Parameters**
>
> - **tool** (Union[*Tool*, str]) – The Tool to base the Node on
> - **id** (Optional[str]) – The id of the node to be created
> - **step_id** (Optional[str]) – The step to add the created node to
> - **resources** (Optional[*ResourceLimit*]) – The resources required to run this node
> - **nodegroup** – The group the node belongs to, this can be important for FlowNodes and such, as they will have matching dimension names.
>
> **Return type** *Node*
>
> **Returns** the newly created node

create_sink (*datatype*, *id=None*, *step_id=None*, *resources=None*, *node_group=None*)

Create a SinkNode in this Network. The Node will be automatically added to the Network.

> **Parameters**
>
> - **datatype** (Union[*BaseDataType*, str]) – The DataType of the sink node
> - **id** (Optional[str]) – The id of the sink node to be created
> - **step_id** (Optional[str]) – The step to add the created sink node to
> - **resources** (Optional[*ResourceLimit*]) – The resources required to run this node

- **node_group** (`str`) – The group the node belongs to, this can be important for FlowNodes and such, as they will have matching dimension names.

> **Return type** `Node`

> **Returns** the newly created sink node

create_source(*datatype*, *id=None*, *step_id=None*, *resources=None*, *node_group=None*)
Create a SourceNode in this Network. The Node will be automatically added to the Network.

> **Parameters**
>
> - **datatype** (`BaseDataType`) – The DataType of the source source_node
> - **id** (`str`) – The id of the source source_node to be created
> - **step_id** (`str`) – The step to add the created source source_node to
> - **resources** (`Optional[ResourceLimit]`) – The resources required to run this node
> - **node_group** (`str`) – The group the node belongs to, this can be important for FlowNodes and such, as they will have matching dimension names.

> **Returns** the newly created source source_node

> **Return type** `SourceNode`

draw(*file_path=None*, *draw_dimensions=True*, *hide_unconnected=True*, *expand_macros=1*)
Draw a graphical representation of the Network

> **Parameters**
>
> - **file_path** (`str`) – The path of the file to create, the extension will control the image type
> - **draw_dimensions** (`bool`) – Flag to control if the dimension sizes should be drawn in the figure, default is true
> - **expand_macros** (`bool`) – Flag to control if and how macro nodes should be expanded, by default 1 level is expanded

> **Return type** `Optional[str]`

> **Returns** path of the image created or None if failed

execute(*source_data*, *sink_data*, *tmpdir=None*, *timestamp=None*, *blocking=True*)
Execute the network with the given source and sink data.

> **Parameters**
>
> - **source_data** (`Dict[str, Union[List[Union[str, Tuple[str, ...]]], Dict[str, Union[str, Tuple[str, ...]]]]]`) – Source data to use as an input
> - **sink_data** (`Union[str, Dict[str, str]]`) – Sink rules to use for determining the outputs
> - **tmpdir** (`Optional[str]`) – The scratch directory to use for this network run, if an existing directory is given, fastr will try to resume a network run (see *Continuing a Network*)
> - **timestamp** (`Union[datetime, str, None]`) – The timestamp of the network run (useful for retrying or continuing previous runs)
> - **blocking** (`bool`) – Flag to indicate if the execution should be blocking or launched in a background thread

> **Return type** `NetworkRun`

> > **Returns** The network run object for the started execution

id
> The unique id describing this resource

> > **Return type** `str`

classmethod load(*filename*)
> Load Network froma file

> > **Parameters filename** (`str`) –

> > **Returns** loaded network

> > **Return type** *Network*

save(*filename*, *indent=2*)
> Save the Network to a JSON file

> > **Parameters**

> > - **filename** (`str`) – Path of the file to save to

> > - **indent** (`int`) – Indentation to use (None for no indentation)

version
> Version of the Network (so users can keep track of their version)

> > **Return type** *Version*

class `fastr.api.`**Link**(*parent*)
> Representation of a link for editing the Network

collapse
> The dimensions which the link will collapse into the cardinality

> > **Return type** `Tuple[Union[int, str], ...]`

expand
> Flag that indicates if the Link will expand the cardinality into a new dimension.

> > **Return type** `bool`

id
> The unique id describing this resource

> > **Return type** `str`

class `fastr.api.`**Node**(*parent*)
> Representation of Node for editing the Network

id
> The unique id describing this resource

> > **Return type** `str`

input
> In case there is only a single Inputs in a Node, this can be used as a short hand. In that case it is basically the same as `list(node.inputs.values()[0])`.

> > **Return type** *Input*

inputs
> Mapping object containing all Inputs of a Node

> > **Return type** `InputMap`

---

output
    In case there is only a single Outputs in a Node, this can be used as a short hand. In that case it is basically
    the same as `list(node.outputs.values()[0])`.

        Return type *Output*

outputs
    Mapping object containing all Outputs of a Node

        Return type SubObjectMap[*Output*]

class fastr.api.**Input**(*parent*)
    Representation of an Input of a Node

    __lshift__(*other*)
        This operator allows the easy creation of Links to this Input using the << operator. Creating links can be
        done by:

```
# Generic form
>> link = input << output
>> link = input << ['some', 'data']  # Create a constant node

# Examples
>> link1 = addint.inputs['left_hand'] << source1.input
>> link2 = addint.inputs['right_hand'] << [1, 2, 3]

# Mutliple links
>> links = addints.inputs['left_hand'] << (source1.output, source2.output,
↪source3.output)
```

        The last example would return a tuple with three links.

            Parameters other (Union[*Output*, *BaseOutput*, list, dict, tuple]) – the target to
                create the link from, this can be an Output, a tuple of Outputs, or a data structure that can be
                used as the data for a ConstantNode

            Return type Union[*Link*, Tuple[*Link*, ...]]

            Returns Newly created link(s)

    __rrshift__(*other*)
        This operator allows to use the >> operator as alternative to using the << operator. See the *__lshift__*
        operator for details.

            Parameters other (Union[*Output*, *BaseOutput*, list, dict, tuple]) – the target to
                create the link from

            Return type Union[*Link*, Tuple[*Link*, ...]]

            Returns Newly created link(s)

    append(*value*)
        Create a link from give resource to a new SubInput.

            Parameters value (Union[*Output*, *BaseOutput*, list, dict, tuple]) – The source
                for the link to be created

            Return type *Link*

            Returns The newly created link

    id
        The unique id describing this resource

> **Return type** `str`

input_group

> The input group of this Input. This property can be read and changed. Changing the input group of an Input will influence the data flow in a Node (see *Advanced flows in a Node* for details).
>
> **Return type** `str`

class `fastr.api.Output`(*parent*)

> Representation of an Output of a Node

__getitem__(*item*)

> Get a SubOuput of this Ouput. The SubOutput selects some data from the parent Output based on an index or slice of the cardinalty.
>
> **Parameters** `key` – the key of the requested item, can be an index or slice
>
> **Return type** *Output*
>
> **Returns** the requested SubOutput with a view of the data in this Output

id

> The unique id describing this resource
>
> **Return type** `str`

FASTR Developer Module reference

3.1 fastr Package

3.1.1 `fastr` Package

Initialize self. See help(type(self)) for accurate signature.

fastr.__init__.__**dir**__() → list
> default dir() implementation

fastr.__init__.__**format**__()
> default object formatter

fastr.__init__.__**init_subclass**__()
> This method is called when a class is subclassed.

> The default implementation does nothing. It may be overridden to extend subclasses.

fastr.__init__.__**new**__()
> Create and return a new object. See help(type) for accurate signature.

fastr.__init__.__**reduce**__()
> helper for pickle

fastr.__init__.__**reduce_ex**__()
> helper for pickle

fastr.__init__.__**sizeof**__() → int
> size of object in memory, in bytes

fastr.__init__.__**subclasshook**__()
> Abstract classes can override this to customize issubclass().

> This is invoked early on by abc.ABCMeta.__subclasscheck__(). It should return True, False or NotImplemented. If it returns NotImplemented, the normal algorithm is used. Otherwise, it overrides the normal algorithm (and the outcome is cached).

3.1.2 `exceptions` Module

This module contains all Fastr-related Exceptions

exception fastr.exceptions.**FastrAttributeError**(*args*, ***kwargs*)
Bases: *fastr.exceptions.FastrError*, AttributeError

AttributeError in the fastr system

__module__ **= 'fastr.exceptions'**

exception fastr.exceptions.**FastrCannotChangeAttributeError**(*args*, ***kwargs*)
Bases: *fastr.exceptions.FastrError*

Attempting to change an attribute of an object that can be set only once.

__module__ **= 'fastr.exceptions'**

exception fastr.exceptions.**FastrCardinalityError**(*args*, ***kwargs*)
Bases: *fastr.exceptions.FastrError*

The description of the cardinality is not valid.

__module__ **= 'fastr.exceptions'**

exception fastr.exceptions.**FastrCollectorError**(*args*, ***kwargs*)
Bases: *fastr.exceptions.FastrError*

Cannot collect the results from a Job because of an error

__module__ **= 'fastr.exceptions'**

exception fastr.exceptions.**FastrDataTypeFileNotReadable**(*args*, ***kwargs*)
Bases: *fastr.exceptions.FastrError*

Could not read the datatype file.

__module__ **= 'fastr.exceptions'**

exception fastr.exceptions.**FastrDataTypeMismatchError**(*args*, ***kwargs*)
Bases: *fastr.exceptions.FastrError*

When using a DataType as the key for the DataTypeManager, the DataTypeManager found another DataType with the same name already in the DataTypeManager. The means fastr has two version of the same DataType in the system, which should never happen!

__module__ **= 'fastr.exceptions'**

exception fastr.exceptions.**FastrDataTypeNotAvailableError**(*args*, ***kwargs*)
Bases: *fastr.exceptions.FastrError*

The DataType requested is not found by the fastr system. Typically this means that no matching DataType is found in the DataTypeManager.

__module__ **= 'fastr.exceptions'**

exception fastr.exceptions.**FastrDataTypeNotInstantiableError**(*args*, ***kwargs*)
Bases: *fastr.exceptions.FastrError*

The base classes for DataTypes cannot be instantiated and should always be sub-classed.

__module__ **= 'fastr.exceptions'**

exception fastr.exceptions.**FastrDataTypeValueError**(*args*, ***kwargs*)
Bases: *fastr.exceptions.FastrError*

This value in fastr did not pass the validation specificied for its DataType, typically means that the data is missing or corrupt.

`__module__` **`= 'fastr.exceptions'`**

exception `fastr.exceptions.`**`FastrError`**(**args*, ***kwargs*)

Bases: `Exception`

This is the base class for all fastr related exceptions. Catching this class of exceptions should ensure a proper execution of fastr.

`__init__`(**args*, ***kwargs*)

Constructor for all exceptions. Saves the caller object fullid (if found) and the file, function and line number where the object was created.

`__module__` **`= 'fastr.exceptions'`**

`__repr__`()

String representation of the error

> **Returns** error representation
>
> **Return type** str

`__str__`()

String value of the error

> **Returns** error string
>
> **Return type** str

`__weakref__`

list of weak references to the object (if defined)

`excerpt`()

Return a excerpt of the Error as a tuple.

exception `fastr.exceptions.`**`FastrErrorInSubprocess`**(**args*, ***kwargs*)

Bases: *fastr.exceptions.FastrExecutionError*

Encountered an error in the subprocess started by the execution script

`__module__` **`= 'fastr.exceptions'`**

exception `fastr.exceptions.`**`FastrExecutableNotFoundError`**(*executable=None*,   **args*,
                                                                                                     **kwargs)

Bases: *fastr.exceptions.FastrExecutionError*

The executable could not be found!

`__init__`(*executable=None*, **args*, ***kwargs*)

Constructor for all exceptions. Saves the caller object fullid (if found) and the file, function and line number where the object was created.

`__module__` **`= 'fastr.exceptions'`**

`__str__`()

String representation of the error

exception `fastr.exceptions.`**`FastrExecutionError`**(**args*, ***kwargs*)

Bases: *fastr.exceptions.FastrError*

Base class for all fastr execution related errors

`__module__` **`= 'fastr.exceptions'`**

exception fastr.exceptions.**FastrFileNotFound**(*filepath*, *message=None*)
    Bases: *fastr.exceptions.FastrError*

    Could not find an expected file

    __init__(*filepath*, *message=None*)
        Constructor for all exceptions. Saves the caller object fullid (if found) and the file, function and line number where the object was created.

    __module__ = **'fastr.exceptions'**

exception fastr.exceptions.**FastrIOError**(**args*, ***kwargs*)
    Bases: *fastr.exceptions.FastrError*, OSError

    IOError in the fastr system

    __module__ = **'fastr.exceptions'**

    __weakref__
        list of weak references to the object (if defined)

exception fastr.exceptions.**FastrImportError**(**args*, ***kwargs*)
    Bases: *fastr.exceptions.FastrError*, ImportError

    ImportError in the fastr system

    __module__ = **'fastr.exceptions'**

    __weakref__
        list of weak references to the object (if defined)

exception fastr.exceptions.**FastrIndexError**(**args*, ***kwargs*)
    Bases: *fastr.exceptions.FastrError*, IndexError

    IndexError in the fastr system

    __module__ = **'fastr.exceptions'**

exception fastr.exceptions.**FastrIndexNonexistent**(**args*, ***kwargs*)
    Bases: *fastr.exceptions.FastrIndexError*

    This is an IndexError for samples requested from a sparse data array. The sample is not there but is probably not there because of sparseness rather than being a missing sample (e.g. out of bounds).

    __module__ = **'fastr.exceptions'**

exception fastr.exceptions.**FastrKeyError**(**args*, ***kwargs*)
    Bases: *fastr.exceptions.FastrError*, KeyError

    KeyError in the fastr system

    __module__ = **'fastr.exceptions'**

exception fastr.exceptions.**FastrLockNotAcquired**(*directory*, *message=None*)
    Bases: *fastr.exceptions.FastrError*

    Could not lock a directory

    __init__(*directory*, *message=None*)
        Constructor for all exceptions. Saves the caller object fullid (if found) and the file, function and line number where the object was created.

    __module__ = **'fastr.exceptions'**

exception fastr.exceptions.**FastrLookupError**(**args*, ***kwargs*)
    Bases: *fastr.exceptions.FastrError*

Could not find specified object in the fastr environment.

`__module__ = 'fastr.exceptions'`

exception fastr.exceptions.**FastrMountUnknownError**(**args*, ***kwargs*)
    Bases: *fastr.exceptions.FastrKeyError*

Trying to access an undefined mount

`__module__ = 'fastr.exceptions'`

exception fastr.exceptions.**FastrNetworkMismatchError**(**args*, ***kwargs*)
    Bases: *fastr.exceptions.FastrError*

Two interacting objects belong to different fastr network.

`__module__ = 'fastr.exceptions'`

exception fastr.exceptions.**FastrNetworkUnknownError**(**args*, ***kwargs*)
    Bases: *fastr.exceptions.FastrKeyError*

Reference to a Tool that is not recognised by the fastr system. This typically means the specific id/version combination of the requested tool has not been loaded by the ToolManager.

`__module__ = 'fastr.exceptions'`

exception fastr.exceptions.**FastrNoValidTargetError**(**args*, ***kwargs*)
    Bases: *fastr.exceptions.FastrKeyError*

Cannot find a valid target for the tool

`__module__ = 'fastr.exceptions'`

exception fastr.exceptions.**FastrNodeAreadyPreparedError**(**args*, ***kwargs*)
    Bases: *fastr.exceptions.FastrStateError*

A attempt is made at preparing a NodeRun for the second time. This is not allowed as it would wipe the current execution data and cause data-loss.

`__module__ = 'fastr.exceptions'`

exception fastr.exceptions.**FastrNodeNotPreparedError**(**args*, ***kwargs*)
    Bases: *fastr.exceptions.FastrStateError*

When trying to access execution data of a NodeRun, the NodeRun must be prepare. The NodeRun has not been prepared by the execution, so the data is not available!

`__module__ = 'fastr.exceptions'`

exception fastr.exceptions.**FastrNodeNotValidError**(**args*, ***kwargs*)
    Bases: *fastr.exceptions.FastrStateError*

A NodeRun is not in a valid state where it should be, typically an invalid NodeRun is passed to the executor causing trouble.

`__module__ = 'fastr.exceptions'`

exception fastr.exceptions.**FastrNotExecutableError**(**args*, ***kwargs*)
    Bases: *fastr.exceptions.FastrExecutionError*

The command invoked by subprocess is not executable on the system

`__module__ = 'fastr.exceptions'`

exception fastr.exceptions.**FastrNotImplementedError**(**args*, ***kwargs*)
    Bases: *fastr.exceptions.FastrError*, NotImplementedError

This function/method has not been implemented on purpose (e.g. should be overwritten in a sub-class)

__module__ = 'fastr.exceptions'

exception fastr.exceptions.**FastrOSError**(*args*, ***kwargs*)
    Bases: *fastr.exceptions.FastrError*, OSError

OSError in the fastr system

__module__ = 'fastr.exceptions'

__weakref__
        list of weak references to the object (if defined)

exception fastr.exceptions.**FastrObjectUnknownError**(*args*, ***kwargs*)
    Bases: *fastr.exceptions.FastrKeyError*

Reference to a Tool that is not recognised by the fastr system. This typically means the specific id/version combination of the requested tool has not been loaded by the ToolManager.

__module__ = 'fastr.exceptions'

exception fastr.exceptions.**FastrOptionalModuleNotAvailableError**(*args*,
                                                                                **kwargs)
    Bases: *fastr.exceptions.FastrNotImplementedError*

A optional modules for Fastr is needed for this function, but is not available on the current python installation.

__module__ = 'fastr.exceptions'

exception fastr.exceptions.**FastrOutputValidationError**(*args*, ***kwargs*)
    Bases: *fastr.exceptions.FastrExecutionError*

An output of a Job does not pass validation

__module__ = 'fastr.exceptions'

exception fastr.exceptions.**FastrParentMismatchError**(*args*, ***kwargs*)
    Bases: *fastr.exceptions.FastrError*

Two interactive objects have different parent where they should be the same

__module__ = 'fastr.exceptions'

exception fastr.exceptions.**FastrPluginCapabilityNotImplemented**(*args*,
                                                                                **kwargs)
    Bases: *fastr.exceptions.FastrNotImplementedError*

A plugin did not implement a capability that it advertised.

__module__ = 'fastr.exceptions'

exception fastr.exceptions.**FastrPluginNotAvailable**(*args*, ***kwargs*)
    Bases: *fastr.exceptions.FastrKeyError*

Indicates that a requested Plugin was not found on the system.

__module__ = 'fastr.exceptions'

exception fastr.exceptions.**FastrPluginNotLoaded**(*args*, ***kwargs*)
    Bases: *fastr.exceptions.FastrStateError*

The plugin was not successfully loaded. This means the plugin class cannot be instantiated.

__module__ = 'fastr.exceptions'

exception fastr.exceptions.**FastrResultFileNotFound**(*filepath*, *message=None*)

    Bases: *fastr.exceptions.FastrFileNotFound*, *fastr.exceptions.FastrExecutionError*

    Could not found the result file of job that finished. This means the executionscript process was killed during interruption. Generally this means a scheduler killed it because of resource shortage.

    __module__ = 'fastr.exceptions'

exception fastr.exceptions.**FastrScriptNotFoundError**(*interpreter=None*, *script=None*, *paths=None*, **args*, ***kwargs*)

    Bases: *fastr.exceptions.FastrExecutionError*

    Script could not be found

    __init__(*interpreter=None*, *script=None*, *paths=None*, **args*, ***kwargs*)

        Constructor for all exceptions. Saves the caller object fullid (if found) and the file, function and line number where the object was created.

    __module__ = 'fastr.exceptions'

    __str__()

        String value of the error

            Returns error string

            Return type str

exception fastr.exceptions.**FastrSerializationError**(*message*, *serializer*, *original_exception=None*)

    Bases: *fastr.exceptions.FastrError*

    The serialization encountered a serious problem

    __init__(*message*, *serializer*, *original_exception=None*)

        Constructor for all exceptions. Saves the caller object fullid (if found) and the file, function and line number where the object was created.

    __module__ = 'fastr.exceptions'

    __repr__()

        Simple string representation of the exception

    __str__()

        Advanced string representation of the exception including the data about where in the schema things went wrong.

exception fastr.exceptions.**FastrSerializationIgnoreDefaultError**(*message*, *serializer*, *original_exception=None*)

    Bases: *fastr.exceptions.FastrSerializationError*

    The value and default are both None, so the value should not be serialized.

    __module__ = 'fastr.exceptions'

exception fastr.exceptions.**FastrSerializationInvalidDataError**(*message*, *serializer*, *original_exception=None*)

    Bases: *fastr.exceptions.FastrSerializationError*

    Encountered data to serialize that is invalid given the serialization schema.

    __module__ = 'fastr.exceptions'

exception fastr.exceptions.**FastrSerializationMethodError**(**args*, ***kwargs*)
    Bases: *fastr.exceptions.FastrKeyError*

    The desired serialization method does not exist.

    __module__ **=** **'fastr.exceptions'**

exception fastr.exceptions.**FastrSinkDataUnavailableError**(**args*, ***kwargs*)
    Bases: *fastr.exceptions.FastrKeyError*

    Could not find the Sink data for the desire sink.

    __module__ **=** **'fastr.exceptions'**

exception fastr.exceptions.**FastrSizeInvalidError**(**args*, ***kwargs*)
    Bases: *fastr.exceptions.FastrError*

    The given size cannot be valid.

    __module__ **=** **'fastr.exceptions'**

exception fastr.exceptions.**FastrSizeMismatchError**(**args*, ***kwargs*)
    Bases: *fastr.exceptions.FastrError*

    The size of two object in fastr is not matching where it should.

    __module__ **=** **'fastr.exceptions'**

exception fastr.exceptions.**FastrSizeUnknownError**(**args*, ***kwargs*)
    Bases: *fastr.exceptions.FastrError*

    The size of object is not (yet) known and only a theoretical estimate is available at the moment.

    __module__ **=** **'fastr.exceptions'**

exception fastr.exceptions.**FastrSourceDataUnavailableError**(**args*, ***kwargs*)
    Bases: *fastr.exceptions.FastrKeyError*

    Could not find the Source data for the desire source.

    __module__ **=** **'fastr.exceptions'**

exception fastr.exceptions.**FastrStateError**(**args*, ***kwargs*)
    Bases: *fastr.exceptions.FastrError*

    An object is in an invalid/unexpected state.

    __module__ **=** **'fastr.exceptions'**

exception fastr.exceptions.**FastrSubprocessNotFinished**(**args*, ***kwargs*)
    Bases: *fastr.exceptions.FastrExecutionError*

    Encountered an error before the subprocess call by the execution script was properly finished.

    __module__ **=** **'fastr.exceptions'**

exception fastr.exceptions.**FastrToolNotAvailableError**(**args*, ***kwargs*)
    Bases: *fastr.exceptions.FastrError*

    The tool used is not available on the current platform (OS and architecture combination) and cannot be
        used.

    __module__ **=** **'fastr.exceptions'**

exception fastr.exceptions.**FastrToolTargetNotFound**(**args*, ***kwargs*)
    Bases: *fastr.exceptions.FastrError*

    Could not determine the location of the tools target binary/script. The tool cannot be used.

       __module__ = 'fastr.exceptions'

exception fastr.exceptions.**FastrToolUnknownError**(**args*, ***kwargs*)
    Bases: *fastr.exceptions.FastrKeyError*

    Reference to a Tool that is not recognised by the fastr system. This typically means the specific id/version combination of the requested tool has not been loaded by the ToolManager.

    __module__ = 'fastr.exceptions'

exception fastr.exceptions.**FastrToolVersionError**(**args*, ***kwargs*)
    Bases: *fastr.exceptions.FastrError*

    Version mismatch, usually the installed tool version and version requested by the network mismatch.

    __module__ = 'fastr.exceptions'

exception fastr.exceptions.**FastrTypeError**(**args*, ***kwargs*)
    Bases: *fastr.exceptions.FastrError*, TypeError

    TypeError in the fastr system

    __module__ = 'fastr.exceptions'

exception fastr.exceptions.**FastrUnknownURLSchemeError**(**args*, ***kwargs*)
    Bases: *fastr.exceptions.FastrKeyError*

    Fastr encountered a data URL with a scheme that was not recognised by the IOPlugin manager.

    __module__ = 'fastr.exceptions'

exception fastr.exceptions.**FastrValueError**(**args*, ***kwargs*)
    Bases: *fastr.exceptions.FastrError*, ValueError

    ValueError in the fastr system

    __module__ = 'fastr.exceptions'

exception fastr.exceptions.**FastrVersionInvalidError**(**args*, ***kwargs*)
    Bases: *fastr.exceptions.FastrValueError*

    The string representation of the version is malformatted.

    __module__ = 'fastr.exceptions'

exception fastr.exceptions.**FastrVersionMismatchError**(**args*, ***kwargs*)
    Bases: *fastr.exceptions.FastrValueError*

    There is a mismatch between different parts of the Fastr environment and integrity is compromised.

    __module__ = 'fastr.exceptions'

fastr.exceptions.**get_message**(*exception*)
    Extract the message from an exception is a safe manner

        Parameters exception (*BaseException*) – exception to extract from

        Returns message string

        Return type str

3.1.3 `version` Module

This module keeps track of the version of the currently used Fastr framework. It can check its version from mercurial or a saved file

`fastr.version.`**`clear_version`**`()`
>   Remove the cached version info

`fastr.version.`**`get_base_version`**`()`
>   Get the version from the top-level version file

>>   **Return type** `Optional[str]`

>>   **Returns** the version

>>   **Rtype str**

`fastr.version.`**`get_hg_info`**`()`
>   Read information about the current mercurial branch and revision

>>   **Return type** `Tuple[Optional[str], Optional[str]]`

>>   **Returns** tuple containing head revision and branch

`fastr.version.`**`get_saved_version`**`()`
>   Get cached version from file

>>   **Return type** `Tuple[Optional[str], Optional[str], Optional[str]]`

>>   **Returns** tuple with version, head revision and branch

`fastr.version.`**`save_version`**`(`*current_version*, *current_hg_head*, *current_hg_branch*`)`
>   Cache the version information (useful for when installing)

>>   **Parameters**

>>>   • **`current_version`** (`str`) – version

>>>   • **`current_hg_head`** (`str`) – mercurial head revision

>>>   • **`current_hg_branch`** (`str`) – mercurial branch

>>   **Returns**

3.1.4 Subpackages

api Package

`api` Package

This module provides the API for fastr that users should use. This API will be considered stable between major versions. If users only interact via this API (and refrain from operating on `parent` attributes), their code should be compatible within major version of fastr.

`fastr.api.`**`create_network`**`(`*id*, *version=None*`)`
>   Create a new Network object

>>   **Parameters**

>>>   • **`id`** (`str`) – id of the network

>>>   • **`version`** (`Union[`*Version*`, str, None]`) – version of the network

>>   **Return type** *Network*

>>   **Returns**

fastr.api.**create_network_copy**(*network_state*)

> Create a network based on another Network state. The network state can be a Network or the state gotten from a Network with __getstate__.

> > **Parameters network_state** (Union[*Network*, *Network*, dict]) – Network (state) to create a copy of

> > **Return type** *Network*

> > **Returns** The rebuilt network

class fastr.api.**ResourceLimit**(*cores=1*, *memory='2G'*, *time='01:00:00'*)

> Bases: object

> **__eq__**(*other*)

> > Check if two resource limits are equal

> > > **Parameters other** – resource limit to test against

> > > **Return type** bool

> **__getstate__**()

> > > **Return type** dict

> **__hash__** = None

> **__init__**(*cores=1*, *memory='2G'*, *time='01:00:00'*)

> > An object describing resource requirements/limits for a node

> > **Parameters**

> > - **cores** (Optional[int]) – number of cores

> > - **memory** (Union[str, int, None]) – memory specification, can be int with number of megabytes or a string with numbers ending on M, G, T, P for megabytes, gigabytes, terrabytes or petabytes. Note that the number has to be an integer, e.g. 1500M would work, whereas 1.5G would be invalid

> > - **time** (Union[str, int, None]) – run time specification, this can be an int with the number of seconds or a string in the HH:MM:SS, MM:SS, or SS format. Where HH, MM, and SS are integers representing the number of hours, minutes and seconds.

> **__module__** = **'fastr.core.resourcelimit'**

> **__ne__**(*other*)

> > Check if two resource limits are not equal

> > > **Parameters other** – resource limit to test against

> > > **Return type** bool

> **__setstate__**(*state*)

> **__slots__** = **('_cores', '_memory', '_time')**

> **copy**()

> > Return a copy of current resource limit object

> > > **Return type** *ResourceLimit*

> **cores**

> > The required number of gpus

> > > **Return type** int

---

memory
> The required memory in megabytes

>> **Return type** int

time
> The required time in seconds

>> **Return type** int

core Package

`core` Package

This module contains all of the core components of fastr. It has the classes to create networks and work with them.

`cardinality` Module

class fastr.core.cardinality.**AnyCardinalitySpec**(*parent*)
> Bases: *fastr.core.cardinality.CardinalitySpec*

> **__eq__**(*other*)
>> Test for equality

> **__hash__** = None

> **__module__** = 'fastr.core.cardinality'

> **__str__**()
>> String version of the cardinality spec, should be parseable by create_cardinality

>>> **Return type** str

> **validate**(*payload*, *cardinality*)
>> Validate cardinality given a payload and cardinality

>>> **Parameters**

>>> • **payload** (dict) – Payload of the corresponding job

>>> • **cardinality** (int) – Cardinality to validate

>>> **Return type** bool

>>> **Returns** Validity of the cardinality given the spec and payload

class fastr.core.cardinality.**AsCardinalitySpec**(*parent*, *target*)
> Bases: *fastr.core.cardinality.CardinalitySpec*

> **__eq__**(*other*)
>> Test for equality

> **__hash__** = None

> **__init__**(*parent*, *target*)
>> Initialize self. See help(type(self)) for accurate signature.

> **__module__** = 'fastr.core.cardinality'

> **__str__**()
>> String version of the cardinality spec, should be parseable by create_cardinality

Return type `str`

calculate_execution_cardinality(*key=None*)

Calculate the cardinality given the node and spec, during execution this should be available and not give unknowns once the data is present and the key is given.

Parameters **key** – Key for which the cardinality is calculated

Return type `Optional`[`int`]

Returns calculated cardinality

calculate_job_cardinality(*payload*)

Calculate the actually cardinality when a job needs to know how many arguments to create for a non-automatic output.

Return type `Optional`[`int`]

calculate_planning_cardinality()

Calculate the cardinality given the node and spec, for cardinalities that only have validation and not a pre-calculable value, this return None.

Parameters **node** – Node for which the cardinality is calculated

Return type `Optional`[`int`]

Returns calculated cardinality

node

predefined

Indicate whether the cardinality is predefined or can only be calculated after execution

class fastr.core.cardinality.**CardinalitySpec**(*parent*)

Bases: `object`

__dict__ **= mappingproxy({'__module__':** **'fastr.core.cardinality',** **'__init__':** **<functi**

__eq__(*other*)

Test for equality

__hash__ **= None**

__init__(*parent*)

Initialize self. See help(type(self)) for accurate signature.

__module__ **= 'fastr.core.cardinality'**

__ne__(*other*)

Return self!=value.

__repr__()

Console representation of the cardinality spec

Return type `str`

__str__()

String version of the cardinality spec, should be parseable by create_cardinality

Return type `str`

__weakref__

list of weak references to the object (if defined)

calculate_execution_cardinality(*key=None*)
Calculate the cardinality given the node and spec, during execution this should be available and not give unknowns once the data is present and the key is given.

> **Parameters key** – Key for which the cardinality is calculated
>
> **Return type** Optional[int]
>
> **Returns** calculated cardinality

calculate_job_cardinality(*payload*)
Calculate the actually cardinality when a job needs to know how many arguments to create for a non-automatic output.

> **Return type** Optional[int]

calculate_planning_cardinality()
Calculate the cardinality given the node and spec, for cardinalities that only have validation and not a pre-calculable value, this return None.

> **Parameters node** – Node for which the cardinality is calculated
>
> **Return type** Optional[int]
>
> **Returns** calculated cardinality

predefined
Indicate whether the cardinality is predefined or can only be calculated after execution

validate(*payload*, *cardinality*, *planning=True*)
Validate cardinality given a payload and cardinality

> **Parameters**
>
> - **payload** (Optional[dict]) – Payload of the corresponding job
>
> - **cardinality** (int) – Cardinality to validate
>
> **Return type** bool
>
> **Returns** Validity of the cardinality given the spec and payload

class fastr.core.cardinality.**ChoiceCardinalitySpec**(*parent*, *options*)
Bases: *fastr.core.cardinality.CardinalitySpec*

__eq__(*other*)
Test for equality

__hash__ = None

__init__(*parent*, *options*)
Initialize self. See help(type(self)) for accurate signature.

__module__ = 'fastr.core.cardinality'

__str__()
String version of the cardinality spec, should be parseable by create_cardinality

> **Return type** str

class fastr.core.cardinality.**IntCardinalitySpec**(*parent*, *value*)
Bases: *fastr.core.cardinality.CardinalitySpec*

__eq__(*other*)
Test for equality

__hash__ = None

__init__(*parent*, *value*)
  Initialize self. See help(type(self)) for accurate signature.

__module__ = **'fastr.core.cardinality'**

__str__()
  String version of the cardinality spec, should be parseable by create_cardinality

  Return type str

calculate_execution_cardinality(*node*)
  Calculate the cardinality given the node and spec, during execution this should be available and not give unknowns once the data is present and the key is given.

  Parameters key – Key for which the cardinality is calculated

  Return type int

  Returns calculated cardinality

calculate_job_cardinality(*payload*)
  Calculate the actually cardinality when a job needs to know how many arguments to create for a non-automatic output.

  Return type Optional[int]

calculate_planning_cardinality()
  Calculate the cardinality given the node and spec, for cardinalities that only have validation and not a pre-calculable value, this return None.

  Parameters node – Node for which the cardinality is calculated

  Return type int

  Returns calculated cardinality

predefined
  Indicate whether the cardinality is predefined or can only be calculated after execution

class fastr.core.cardinality.**MaxCardinalitySpec**(*parent*, *value*)
  Bases: *fastr.core.cardinality.CardinalitySpec*

  __eq__(*other*)
    Test for equality

  __hash__ = **None**

  __init__(*parent*, *value*)
    Initialize self. See help(type(self)) for accurate signature.

  __module__ = **'fastr.core.cardinality'**

  __str__()
    String version of the cardinality spec, should be parseable by create_cardinality

    Return type str

class fastr.core.cardinality.**MinCardinalitySpec**(*parent*, *value*)
  Bases: *fastr.core.cardinality.CardinalitySpec*

  __eq__(*other*)
    Test for equality

  __hash__ = **None**

__init__ (*parent*, *value*)
   Initialize self. See help(type(self)) for accurate signature.

__module__ **= 'fastr.core.cardinality'**

__str__ ()
   String version of the cardinality spec, should be parseable by create_cardinality

   Return type str

class fastr.core.cardinality.**RangeCardinalitySpec** (*parent*, *min*, *max*)
   Bases: *fastr.core.cardinality.CardinalitySpec*

__eq__ (*other*)
   Test for equality

__hash__ **= None**

__init__ (*parent*, *min*, *max*)
   Initialize self. See help(type(self)) for accurate signature.

__module__ **= 'fastr.core.cardinality'**

__str__ ()
   String version of the cardinality spec, should be parseable by create_cardinality

   Return type str

class fastr.core.cardinality.**ValueCardinalitySpec** (*parent*, *target*)
   Bases: *fastr.core.cardinality.CardinalitySpec*

__eq__ (*other*)
   Test for equality

__hash__ **= None**

__init__ (*parent*, *target*)
   Initialize self. See help(type(self)) for accurate signature.

__module__ **= 'fastr.core.cardinality'**

__str__ ()
   String version of the cardinality spec, should be parseable by create_cardinality

   Return type str

calculate_execution_cardinality (*key=None*)
   Calculate the cardinality given the node and spec, during execution this should be available and not give
   unknowns once the data is present and the key is given.

   Parameters key – Key for which the cardinality is calculated

   Return type Optional[int]

   Returns calculated cardinality

calculate_job_cardinality (*payload*)
   Calculate the actually cardinality when a job needs to know how many arguments to create for a non-
   automatic output.

   Return type Optional[int]

node

`fastr.core.cardinality.`**`create_cardinality`**(*desc*, *parent*)
Create simplified description of the cardinality. This changes the string representation to a tuple that is easier to check at a later time.

> **Parameters** **`desc`** (`str`) – the string version of the cardinality
>
> **Parent** the parent input or output to which this cardinality spec belongs
>
> **Return type** *CardinalitySpec*
>
> **Returns** the simplified cardinality description
>
> **Raises** *FastrCardinalityError* – if the Input/Output has an incorrect cardinality description.

The translation works with the following table:

cardinality string	cardinality spec	description
`"*"`, `any`	"('any',)	Any cardinality is allowed
`"N"`	`('int', N)`	A cardinality of N is required
`"N-M"`	`('range', N, M)`	A cardinality between N and M is required
`"*-M"`	`('max', M)`	A cardinality of maximal M is required
`"N-*"`	`('min', N)`	A cardinality of minimal N is required
`"[M,N,...,O, P]"`	`('choice', [M,N,..., O,P])`	The cardinality should one of the given options
`"as:input_id"`	`('as', 'input_id')`	The cardinality should match the cardinality of the given Input
`"val:input_id"`	`('val', 'input_id')`	The cardinliaty should match the value of the given Input

> **Note:** The maximumu, minimum and range are inclusive

dimension Module

class `fastr.core.dimension.`**`Dimension`**(*name*, *size*)
Bases: `object`

A class representing a dimension. It contains the name and size of the dimension.

`__eq__`(*other*)
Dimension is the same if the name and size are the same

`__hash__` = None

`__init__`(*name*, *size*)
The constructor for the dimension.

> **Parameters**
>
> • **`name`** (`str`) – Name of the dimension
>
> • **`size`** (`int or sympy.Symbol`) – Size fo the dimension

`__module__` = 'fastr.core.dimension'

`__ne__`(*other*)
The not equal test is simply the inverse of the equal test

__repr__()
> String representation of a Dimension

__slots__ = ('_name', '_size')

copy()
> Get a copy object of a Dimension

name

size

update_size(*value*)

class fastr.core.dimension.**ForwardsDimensions**
> Bases: *fastr.core.dimension.HasDimensions*

> Class of objects that have dimensions not because they contain data with dimensions but forward them (optionally with changes via combine_dimensions)

> **__abstractmethods__ = frozenset({'source', 'combine_dimensions'})**

> **__module__ = 'fastr.core.dimension'**

> **combine_dimensions**(*dimensions*)
> > Method to combine/manipulate the dimensions

> > > **Parameters dimensions** – the input dimensions from the source

> > > **Returns** dimensions manipulated for this object

> > > **Return type** tuple of dimensions

> **dimensions**
> > The dimensions of the object based on the forwarding

> **source**
> > The source object from which the dimensions are forwarded

> > > **Returns** the object from which the dimensions are forwarded

> > > **Return type** *HasDimensions*

class fastr.core.dimension.**HasDimensions**
> Bases: object

> A Mixin class for any object that has a notion of dimensions and size. It uses the dimension property to expose the dimension name and size.

> **__abstractmethods__ = frozenset({'dimensions'})**

> **__dict__ = mappingproxy({'__module__': 'fastr.core.dimension', '__doc__': '\n A Mixin**

> **__module__ = 'fastr.core.dimension'**

> **__weakref__**
> > list of weak references to the object (if defined)

> **dimensions**
> > The dimensions has to be implemented by any subclass. It has to provide a tuple of Dimensions.

> > > **Returns** dimensions

> > > **Return type** tuple

> **dimnames**
> > A tuple containing the dimension names of this object. All items of the tuple are of type str.

---

ndims
> The number of dimensions in this object

size
> A tuple containing the size of this object. All items of the tuple are of type int or sympy.Symbol.

`interface` Module

A module that describes the interface of a Tool. It specifies how a set of input values will be translated to commands to be executed. This creates a generic interface to different ways of executing underlying software.

class fastr.core.interface.**InputSpec**
> Bases: *fastr.core.interface.InputSpec*
>
> Class containing the information about an Input Specification, this is essentially a data class (but
>
> **__dict__** = mappingproxy({'__module__': 'fastr.core.interface', '__doc__': '\n Class
>
> **__module__** = 'fastr.core.interface'
>
> **static __new__**(*cls*, *id_*, *cardinality*, *datatype*, *required=False*, *description=''*, *default=None*, *hidden=False*)
> > Create new instance of InputSpec(id, cardinality, datatype, required, description, default, hidden)

fastr.core.interface.**InputSpecBase**
> alias of *fastr.core.interface.InputSpec*

class fastr.core.interface.**Interface**
> Bases: fastr.abc.baseplugin.Plugin, fastr.abc.serializable.Serializable
>
> Abstract base class of all Interfaces. Defines the minimal requirements for all Interface implementations.
>
> **__abstractmethods__** = frozenset({'__setstate__', 'inputs', 'outputs', '__getstate__',
>
> **__getstate__**()
> > Retrieve the state of the Interface
> >
> > > **Returns** the state of the object
> > >
> > > **Rtype dict**
>
> **__module__** = 'fastr.core.interface'
>
> **__setstate__**(*state*)
> > Set the state of the Interface
>
> **execute**(*target*, *payload*)
> > Execute the interface given the a target and payload. The payload should have the form {'input': {'input_id_a': (value, value), 'input_id_b': (value, value)},
> >
> > > 'output': {'output_id_a': (value, value), 'output_id_b': (value, value)}}
> >
> > > **Parameters**
> > > - **target** – the target to call
> > > - **payload** – the payload to use
> > >
> > > **Returns** the result of the execution
> > >
> > > **Return type** (tuple of) *InterfaceResult*

---

expanding
> Indicates whether or not this Interface will result in multiple samples per run. If the flow is unaffected, this
> will be zero, if it is nonzero it means that number of dimension will be added to the sample array.

inputs
> OrderedDict of Inputs connected to the Interface. The format should be {input_id: InputSpec}.

outputs
> OrderedDict of Output connected to the Interface. The format should be {output_id: OutputSpec}.

classmethod test()
> Test the plugin, interfaces do not need to be tested on import

class fastr.core.interface.**InterfaceResult**(*result_data*, *target_result*, *payload*, *sample_index=None*, *sample_id=None*, *errors=None*)

> Bases: object

The class in which Interfaces should wrap their results to be picked up by fastr

__dict__ = mappingproxy({'__module__': 'fastr.core.interface', '__doc__': '\n The cl

__init__(*result_data*, *target_result*, *payload*, *sample_index=None*, *sample_id=None*, *errors=None*)
> Initialize self. See help(type(self)) for accurate signature.

__module__ = 'fastr.core.interface'

__weakref__
> list of weak references to the object (if defined)

class fastr.core.interface.**OutputSpec**
> Bases: *fastr.core.interface.OutputSpec*

Class containing the information about an Output Specification, this is essentially a data class (but

__dict__ = mappingproxy({'__module__': 'fastr.core.interface', '__doc__': '\n Class

__module__ = 'fastr.core.interface'

static __new__(*cls*, *id_*, *cardinality*, *datatype*, *automatic=True*, *required=False*, *description=''*, *hidden=False*)
> Create new instance of OutputSpec(id, cardinality, datatype, automatic, required, description, hidden)

fastr.core.interface.**OutputSpecBase**
> alias of *fastr.core.interface.OutputSpec*

ioplugin Module

This module contains the manager class for IOPlugins and the base class for all IOPlugins

class fastr.core.ioplugin.**IOPlugin**
> Bases: fastr.abc.baseplugin.Plugin

IOPlugins are used for data import and export for the sources and sinks. The main use of the *IOPlugins* is
during execution (see *Execution*). The *IOPlugins* can be accessed via fastr.ioplugins, but generally
there should be no need for direct interaction with these objects. The use of is mainly via the URL used to
specify source and sink data.

__abstractmethods__ = frozenset({'scheme'})

__init__()
> Initialization for the IOPlugin

> **Returns** newly created IOPlugin

__module__ = 'fastr.core.ioplugin'

cleanup()
> (abstract) Clean up the IOPlugin. This is to do things like closing files or connections. Will be called when the plugin is no longer required.

expand_url(*url*)
> (abstract) Expand an URL. This allows a source to collect multiple samples from a single url. The URL will have a wildcard or point to something with info and multiple urls will be returned.
>
> > **Parameters url** (*str*) – url to expand
> >
> > **Returns** the resulting url(s), a tuple if multiple, otherwise a str
> >
> > **Return type** str or tuple of str

fetch_url(*inurl*, *outfile*)
> (abstract) Fetch a file from an external data source.
>
> > **Parameters**
> >
> > > • **inurl** – url to the item in the data store
> > >
> > > • **outpath** – path where to store the fetch data locally

fetch_value(*inurl*)
> (abstract) Fetch a value from an external data source.
>
> > **Parameters inurl** – the url of the value to retrieve
> >
> > **Returns** the fetched value

static isurl(*string*)
> Test if given string is an url.
>
> > **Parameters string** (*str*) – string to test
> >
> > **Returns** True if the string is an url, False otherwise
> >
> > **Return type** bool

path_to_url(*path*, *mountpoint=None*)
> (abstract) Construct an url from a given mount point and a relative path to the mount point.
>
> > **Parameters**
> >
> > > • **path** (*str*) – the path to determine the url for
> > >
> > > • **mountpoint** (*str or None*) – the mount point to use, will be automatically detected if None is given
> >
> > **Returns** url matching the path
> >
> > **Return type** str

static print_result(*result*)
> Print the result of the IOPlugin to stdout to be picked up by the tool
>
> > **Parameters result** – value to print as a result
> >
> > **Returns** None

pull_source_data(*inurl*, *outdir*, *sample_id*, *datatype=None*)
> Transfer the source data from inurl to be available in outdir.
>
> > **Parameters**

- **inurl** (`str`) – the input url to fetch data from
- **outdir** (`str`) – the directory to write the data to
- **datatype** (`DataType`) – the datatype of the data, used for determining the total contents of the transfer

> **Returns** None

push_sink_data (*inpath*, *outurl*, *datatype=None*)
> Write out the sink data from the inpath to the outurl.

> **Parameters**

- **inpath** (`str`) – the path of the data to be pushed
- **outurl** (`str`) – the url to write the data to
- **datatype** (`DataType`) – the datatype of the data, used for determining the total contents of the transfer

> **Returns** None

put_url (*inpath*, *outurl*)
> (`abstract`) Put the files to the external data store.

> **Parameters**

- **inpath** – path to the local data
- **outurl** – url to where to store the data in the external data store.

put_value (*value*, *outurl*)
> (`abstract`) Put the files to the external data store.

> **Parameters**

- **value** – the value to store
- **outurl** – url to where to store the data in the external data store.

scheme
> (`abstract`) This abstract property is to be overwritten by a subclass to indicate the url scheme associated with the IOPlugin.

setup (**args*, ***kwargs*)
> (`abstract`) Setup before data transfer. This can be any function that needs to be used to prepare the plugin for data transfer.

url_to_path (*url*)
> (`abstract`) Get the path to a file from a url.

> **Parameters url** (`str`) – the url to retrieve the path for

> **Returns** the corresponding path

> **Return type** str

`provenance` **Module**

class fastr.core.provenance.**Provenance** (*host=None*)
> Bases: `object`

The Provenance object keeps track of everything that happens to a data object.

__dict__ = mappingproxy({'__module__': 'fastr.core.provenance', '__doc__': '\n The P:

__init__(*host=None*)
    Initialize self. See help(type(self)) for accurate signature.

__module__ = 'fastr.core.provenance'

__weakref__
    list of weak references to the object (if defined)

activity(*identifier*, *start_time=None*, *end_time=None*, *other_attributes=None*)

agent(*identifier*, *other_attributes=None*)

collect_input_argument_provenance(*input_argument*)

collect_provenance(*job*, *advanced_flow=False*)
    Collect the provenance for this job

static data_uri(*value*, *job*)

entity(*identifier*, *other_attributes=None*)

static get_parent_provenance(*value*)
    Find the provenance of the parent job

        Parameters value (*str*) – url for the value for which to find the job

        Returns the provenance of the job that created the value

        Raises

            • **_FastrKeyError_** – if the deferred is not available (yet)

            • **_FastrValueError_** – if the value is not a valid deferred url

init_provenance(*job*)
    Create initial provenance document

serialize(*filename*, *format*)

resourcelimit Module

Module for the management of resource limits of compute resources

class fastr.core.resourcelimit.**ResourceLimit**(*cores=1*, *memory='2G'*, *time='01:00:00'*)
    Bases: `object`

__eq__(*other*)
    Check if two resource limits are equal

        Parameters other – resource limit to test against

        Return type `bool`

__getstate__()

        Return type `dict`

__hash__ = None

__init__(*cores=1*, *memory='2G'*, *time='01:00:00'*)
    An object describing resource requirements/limits for a node

        Parameters

- **cores** (`Optional`[`int`]) – number of cores

- **memory** (`Union`[`str`, `int`, `None`]) – memory specification, can be int with number of megabytes or a string with numbers ending on M, G, T, P for megabytes, gigabytes, terrabytes or petabytes. Note that the number has to be an integer, e.g. 1500M would work, whereas 1.5G would be invalid

- **time** (`Union`[`str`, `int`, `None`]) – run time specification, this can be an int with the number of seconds or a string in the HH:MM:SS, MM:SS, or SS format. Where HH, MM, and SS are integers representing the number of hours, minutes and seconds.

__module__ = **'fastr.core.resourcelimit'**

__ne__(*other*)

    Check if two resource limits are not equal

        Parameters **other** – resource limit to test against

        Return type `bool`

__setstate__(*state*)

__slots__ = **('_cores', '_memory', '_time')**

copy()

    Return a copy of current resource limit object

        Return type *ResourceLimit*

cores

    The required number of gpus

        Return type `int`

memory

    The required memory in megabytes

        Return type `int`

time

    The required time in seconds

        Return type `int`

`samples` Module

This package holds the classes for working with samples.

class fastr.core.samples.**ContainsSamples**

    Bases: *fastr.core.samples.HasSamples*

    __abstractmethods__ = **frozenset({'samples'})**

    __getitem__(*item*)

    __module__ = **'fastr.core.samples'**

    __setitem__(*key*, *value*)

    dimensions

        The dimensions has to be implemented by any subclass. It has to provide a tuple of Dimensions.

        Returns dimensions

        Return type tuple

samples

class fastr.core.samples.**HasSamples**

 Bases: *fastr.core.dimension.HasDimensions*

 Base class for all classes that supply samples. This base class allows to only define __getitem__ and size and get all other basic functions mixed in so that the object behaves similar to a Mapping.

 __abstractmethods__ = frozenset({'dimensions', '__getitem__'})

 __contains__(*item*)

 __getitem__(*item*)

 __iter__()

 __module__ = 'fastr.core.samples'

 ids()

 indexes()

 items()

 iteritems()

class fastr.core.samples.**SampleBaseId**

 Bases: tuple

 This class represents a sample id. A sample id is a multi-dimensional id that has a simple, consistent string representation.

 __add__(*other*)

  Add another SampleId, this allows to add parts to the SampleId in a convenient way.

 __dict__ = mappingproxy({'__module__': 'fastr.core.samples', '__doc__': '\n This cla

 __module__ = 'fastr.core.samples'

 static __new__(*cls*, **args*)

  Create a new SampleId

   Parameters args (*iterator/iterable of element type or element type*) – the strings to make sample id for

 __radd__(*other*)

  Add another SampleId, this allows to add parts to the SampleId in a convenient way. This is the right-hand version of the operator.

 __repr__()

  Get a string representation for the SampleBaseId

   Returns the string representation

   Return type str

 __str__()

  Get a string version for the SampleId, joins the SampleId with __ to create a single string version.

   Returns the string version

   Return type str

class fastr.core.samples.**SampleCollection**(*dimnames*, *parent*)

 Bases: collections.abc.MutableMapping, *fastr.core.dimension.HasDimensions*

The SampleCollections is a class that contains the data including a form of ordering. Each sample is reachable both by its SampleId and a SampleIndex. The object is sparse, so not all SampleId have to be defined allowing for non-rectangular data shapes.

Note: This object is meant to replace both the SampleIdList and the ValueStorage.

`__abstractmethods__ = frozenset()`

`__contains__`(*item*)

    Check if an item is in the SampleCollection. The item can be a SampleId or SampleIndex. If the item is a slicing SampleIndex, then check if it would return any data (True) or no data (False)

        Parameters item (SampleId, SampleIndex) – the item to check for

        Returns flag indicating item is in the collections

        Return type bool

`__delitem__`(*key*)

    Remove an item from the SampleCollection

        Parameters key (SampleId, SampleIndex, *tuple of both, or* SampleItem) – the key of the item to remove

`__dict__ = mappingproxy({'__module__':  'fastr.core.samples', '__doc__':  '\n The Samp`

`__getitem__`(*item*)

    Retrieve (a) SampleItem(s) from the SampleCollection using the SampleId or SampleIndex. If the item is a tuple, it should be valid tuple for constructing either a SampleId or SampleIndex.

        Parameters item (SampleId, SampleIndex, *or* tuple) – the identifier of the item to retrieve

        Returns the requested item

        Return type *SampleItem*

        Raises

            • **`FastrTypeError`** – if the item parameter is of incorrect type

            • **`KeyError`** – if the item is not found

`__init__`(*dimnames*, *parent*)

    Createa a new SampleCollection

`__iter__`()

    Iterate over the indices

`__len__`()

    Get the number of samples in the SampleCollections.

`__module__ = 'fastr.core.samples'`

`__repr__`()

    Return repr(self).

`__setitem__`(*key*, *value*)

    Set an item to the SampleCollection. The key can be a SampleId, SampleIndex or a tuple containing a SampleId and SampleIndex. The value can be a SampleItem (with the SampleId and SampleIndex matching), a tuple with values (assuming no depending jobs), or a with a list of values and a set of depending jobs.

Parameters

- **key** (`SampleId`, `SampleIndex`, *tuple of both, or* `SampleItem`) – the key of the item to store

- **value** (`SampleItem`, *tuple of values, or tuple of tuple of values and set of depending jobs*) – the value of the SampleItem to store

Raises

- **`FastrTypeError`** – if the key or value types are incorrect

- **`FastrValueError`** – if the id or values are incorrectly formed

__weakref__
> list of weak references to the object (if defined)

dimensions
> The dimensions has to be implemented by any subclass. It has to provide a tuple of Dimensions.

> > **Returns** dimensions

> > **Return type** tuple

fullid
> The full defining ID for the SampleIdList

ndims
> The number of dimensions in this SampleCollection

parent
> The parent object holding the SampleCollection

class fastr.core.samples.**SampleId**
> Bases: *fastr.core.samples.SampleBaseId*

> SampleId is an identifier for data using human readable strings

> **__module__ = 'fastr.core.samples'**

class fastr.core.samples.**SampleIndex**
> Bases: *fastr.core.samples.SampleBaseId*

> SampleId is an identifier for data using the location in the N-d data structure.

> **__module__ = 'fastr.core.samples'**

> **__repr__**()
> > Get a string representation for the SampleIndex

> > **Returns** the string representation

> > **Return type** str

> **__str__**()
> > Get a string version for the SampleId, joins the SampleId with __ to create a single string version.

> > **Returns** the string version

> > **Return type** str

expand(*size*)
> Function expanding a slice SampleIndex into a list of non-slice SampleIndex objects

> > **Parameters** **size** – the size of the collection to slice

> **isslice**
>> Flag indicating that the SampleIndex is a slice (as opposed to a simple single index).

class fastr.core.samples.**SampleItem**

> Bases: *fastr.core.samples.SampleItemBase*

> **__module__ = 'fastr.core.samples'**

> **static __new__**(*cls*, *index*, *id_*, *data*, *jobs=None*, *failed_annotations=None*)
>> Create a SampleItem. Data should be an OrderedDict of tuples.

>> **Parameters**

>>> - **index** (*tuple, slice*) – the sample index
>>> - **id** (SampleId) – the sample id
>>> - **data** (*SampleValue, Mapping*) – the data values
>>> - **jobs** (*set*) – set of jobs on which this SampleItems data depends.
>>> - **failed_annotations** (*set*) – set of tuples. The tuple is contructed like follows: (job_id, reason).

class fastr.core.samples.**SampleItemBase**

> Bases: tuple

This class represents a sample item, a combination of a SampleIndex, SampleID, value and required jobs. The SampleItem based on a named tuple and has some extra methods to combine SampleItems easily.

> **__add__**(*other*)
>> The addition operator combines two SampleItems into a single SampleItems. It merges the data and jobs and takes the index and id of the left-hand item.

>> **Parameters other** (SampleItem) – The other item to add to this one

>> **Returns** the combined SampleItem

>> **Return type** *SampleItem*

> **__dict__ = mappingproxy({'__module__': 'fastr.core.samples', '__doc__': '\n This cla**

> **__getnewargs__**()
>> Get new args gives the arguments to use to re-create this object, This is used for serialization.

> **__module__ = 'fastr.core.samples'**

> **static __new__**(*cls*, *index*, *id_*, *data*, *jobs=None*, *failed_annotations=None*)
>> Create a SampleItem. Data should be an OrderedDict of tuples.

>> **Parameters**

>>> - **index** (*tuple, slice*) – the sample index
>>> - **id** (SampleId) – the sample id
>>> - **data** (*SampleValue, Mapping*) – the data values
>>> - **jobs** (*set*) – set, tuple or list of jobs on which this SampleItems data depends.
>>> - **failed_annotations** (*set*) – set of tuples. The tuple is contructed like follows: (job_id, reason).

> **__repr__**()
>> Get a string representation for the SampleItem

>> **Returns** the string representation

> **Return type** str

cardinality
>   The cardinality of this Sample

static combine(**args*)
>   Combine a number of SampleItems into a new one.

>   > **Parameters** ***args** – the SampleItems to combine

>   > **Returns** the combined SampleItem

>   > **Return type** *SampleItem*

>   It is possible to both give multiple arguments, where each argument is a SampleItem, or a single argument
>   which is an iterable yielding SampleItems.

```python
# variables a, b, c, d are SampleItems to combine
# These are all valid ways of combining the SampleItems
comb1 = SampleItem.combine(a, b, c, d)  # Using multiple arguments
l = [a, b, c, d]
comb2 = SampleItem.combine(l)  # Using a list of arguments
comb3 = SampleItem.combine(l.__iter__())  # Using an iterator
```

data
>   The data SampleValue of the SampleItem

>   > **Returns** The value of this SampleItem

>   > **Return type** *SampleValue*

dimensionality
>   The dimensionality of this Sample

failed_annotations

id
>   The sample id of the SampleItem

>   > **Returns** The id of this SampleItem

>   > **Return type** *SampleId*

index
>   The index of the SampleItem

>   > **Returns** The index of this SampleItem

>   > **Return type** *SampleIndex*

jobs
>   The set of the jobs on which this SampleItem depends

>   > **Returns** The jobs that generated the data for this SampleItem

>   > **Return type** set

class fastr.core.samples.**SamplePayload**
>   Bases: *fastr.core.samples.SampleItemBase*

__add__(*other*)
>   The addition operator combines two SampleItems into a single SampleItems. It merges the data and jobs
>   and takes the index and id of the left-hand item.

>   > **Parameters** **other** (SampleItem) – The other item to add to this one

> > > **Returns** the combined SamplePayload
> >
> > > **Return type** *SamplePayload*
>
> **__module__ = 'fastr.core.samples'**
>
> **static __new__**(*cls*, *index*, *id_*, *data*, *jobs=None*, *failed_annotations=None*)
> > Create a SampleItem. Data should be an OrderedDict of tuples.
> >
> > > **Parameters**
> > >
> > > > • **index** (*tuple, slice*) – the sample index
> > > >
> > > > • **id** (SampleId) – the sample id
> > > >
> > > > • **data** (*SampleValue, Mapping*) – the data values
> > > >
> > > > • **jobs** (*set*) – set of jobs on which this SampleItems data depends.
> > > >
> > > > • **failed_annotations** (*set*) – set of tuples. The tuple is contructed like follows: (job_id, reason).

class fastr.core.samples.**SampleValue**(**args*, ***kwargs*)
> Bases: collections.abc.MutableMapping
>
> A collection containing the content of a sample
>
> **__abstractmethods__ = frozenset()**
>
> **__add__**(*other*)
>
> **__delitem__**(*key*)
>
> **__dict__ = mappingproxy({'__module__': 'fastr.core.samples', '__doc__': '\n A collect**
>
> **__getitem__**(*item*)
>
> **__getstate__**()
>
> **__init__**(**args*, ***kwargs*)
> > Initialize self. See help(type(self)) for accurate signature.
>
> **__iter__**()
>
> **__len__**()
>
> **__module__ = 'fastr.core.samples'**
>
> **__radd__**(*other*)
>
> **__repr__**()
> > Return repr(self).
>
> **__setitem__**(*key*, *value*)
>
> **__setstate__**(*state*)
>
> **__weakref__**
> > list of weak references to the object (if defined)
>
> **cast**(*datatype*)
>
> **is_mapping**
>
> **is_sequence**
>
> **iterelements**()
>
> **mapping_part**()

> **sequence_part**()

target Module

The module containing the classes describing the targets.

class fastr.core.target.**ProcessUsageCollection**

    Bases: `collections.abc.Sequence`

    __abstractmethods__ = frozenset()

    __dict__ = mappingproxy({'__module__': 'fastr.core.target', 'usage_type': <class 'fa

    __getitem__(*item*)

    __init__()

        Initialize self. See help(type(self)) for accurate signature.

    __len__()

    __module__ = 'fastr.core.target'

    __weakref__

        list of weak references to the object (if defined)

    aggregate(*number_of_points*)

    append(*value*)

    usage_type

        alias of *SystemUsageInfo*

class fastr.core.target.**SubprocessBasedTarget**

    Bases: *fastr.core.target.Target*

    Abstract based class for targets which call the target via a subprocess. Supplies a call_subprocess which executes the command and profiles the resulting subprocess.

    __abstractmethods__ = frozenset({'run_command'})

    __module__ = 'fastr.core.target'

    call_subprocess(*command*)

        Call a subprocess with logging/timing/profiling

            Parameters command (*list*) – the command to execute

            Returns execution info

            Return type dict

    monitor_process(*process*, *resources*)

        Monitor a process and profile the cpu, memory and io use. Register the resource use every _MONITOR_INTERVAL seconds.

        Parameters

            • **process** (*subproces.Popen*) – process to monitor

            • **resources** (*ProcessUsageCollection*) – list to append measurements to

class fastr.core.target.**SystemUsageInfo**(*timestamp*, *cpu_percent*, *vmem*, *rmem*, *read_bytes*,
                                      write_bytes)

    Bases: tuple

__getnewargs__()
> Return self as a plain tuple. Used by copy and pickle.

__module__ = 'fastr.core.target'

static __new__(*_cls*, *timestamp*, *cpu_percent*, *vmem*, *rmem*, *read_bytes*, *write_bytes*)
> Create new instance of SystemUsageInfo(timestamp, cpu_percent, vmem, rmem, read_bytes, write_bytes)

__repr__()
> Return a nicely formatted representation string

__slots__ = ()

cpu_percent
> Alias for field number 1

read_bytes
> Alias for field number 4

rmem
> Alias for field number 3

timestamp
> Alias for field number 0

vmem
> Alias for field number 2

write_bytes
> Alias for field number 5

class fastr.core.target.**Target**
> Bases: fastr.abc.baseplugin.Plugin

The abstract base class for all targets. Execution with a target should follow the following pattern:

```
>>> with Target() as target:
...     target.run_commmand(['sleep', '10'])
```

The Target context operator will set the correct paths/initialization. Within the context command can be ran and when leaving the context the target reverts the state before.

__abstractmethods__ = frozenset({'run_command'})

__enter__()
> Set the environment in such a way that the target will be on the path.

__exit__(*exc_type*, *exc_value*, *traceback*)
> Cleanup the environment where needed

__module__ = 'fastr.core.target'

run_command(*command*)
> Run a command with the target

> > **Return type** *TargetResult*

classmethod test()
> Test the plugin, interfaces do not need to be tested on import

class fastr.core.target.**TargetResult**(*return_code*, *stdout*, *stderr*, *command*, *resource_usage*,
> > > > *time_elapsed*)
> Bases: object

__dict__ = mappingproxy({'__module__': 'fastr.core.target', '__init__': <function Ta

__init__ (*return_code*, *stdout*, *stderr*, *command*, *resource_usage*, *time_elapsed*)
    Class to formalize the resulting data of a Target

      Parameters

- **return_code** (`int`) – the return code of the process
- **stdout** (`Union[str, bytes]`) – the stdout generated by the process
- **stderr** (`Union[str, bytes]`) – the stderr generated by the process
- **command** (`List[Union[str, bytes]]`) – the command executed
- **resource_usage** (`List[`*SystemUsageInfo*`]`) – the resource use during execution
- **time_elapsed** (`int`) – time used (in seconds)

__module__ = `'fastr.core.target'`

__weakref__
    list of weak references to the object (if defined)

as_dict()
    A dictionary of the data in the object (meant for serialization)

      Return type `Dict[str, Union[int, str, List[~T]]]`

`tool` Module

A module to maintain a tool.

Exported classes:

- Tool – A class encapsulating a tool.
- ParameterDescription – The base class containing the shared description of a parameter (both input and ouput).
- InputParameterDescription – A class containing the description of an input parameter.
- Output ParameterDescription – A class containing the description of an output parameter.

class `fastr.core.tool.`**Tool**(*doc=None*)
    Bases: `fastr.abc.serializable.Serializable`

    The class encapsulating a tool.

    DEFAULT_TARGET_CLASS = {'MacroNode': 'MacroTarget'}

    TOOL_REFERENCE_FILE_NAME = '__fastr_tool_ref__.json'

    TOOL_RESULT_FILE_NAME = '__fastr_tool_result.pickle.gz'

    __dataschemafile__ = 'Tool.schema.json'

    __eq__(*other*)
      Compare two Tool instances with each other.

      Parameters other (`Tool`) – the other instances to compare to

      Returns True if equal, False otherwise

    __getstate__()
      Retrieve the state of the Tool

      Returns the state of the object

      Rtype dict

__hash__ = None

__init__(*doc=None*)
> Create a new Tool :param doc: path of toolfile or a dict containing the tool data :type doc: str or dict

__module__ = 'fastr.core.tool'

__repr__()
> Get a string representation for the Tool. This will show the inputs and output defined in a table-like structure.
>
> > **Returns** the string representation
> >
> > **Return type** str

__setstate__(*state*)
> Set the state of the Tool by the given state.
>
> > **Parameters state** (*dict*) – The state to populate the object with

__str__()
> Get a string version for the Tool
>
> > **Returns** the string version
> >
> > **Return type** str

authors = None
> List of authors of the tool. These people wrapped the executable but are not responsible for executable itself.

cite = None
> This holds the citation you should use when publishing something based on this Tool

command = None
> Command is a dictionary contain information about the command which is called by this Tool: command['interpreter'] holds the (possible) interpreter to use command['targets'] holds a per os/arch dictionary of files that should be executed command['url'] is the webpage of the command to be called command['version'] is the version of the command used command['description'] can help a description of the command command['authors'] lists the original authors of the command

command_version

static compare_output_data(*current_output_data*, *reference_output_data*, *validation_result*, *output*)

create_reference(*input_data*, *output_directory*, *mount_name='__ref_tmp__'*, *copy_input=True*)

description = None
> Description of the tool and it's functionality

execute(*payload=None*, ***kwargs*)
> Execute a Tool given the payload for a single run
>
> > **Parameters payload** – the data to execute the Tool with
> >
> > **Returns** The result of the execution
> >
> > **Return type** InterFaceResult

fullid
> The full id of this tool

hash

---

help = None
> Man page for the Tool. Here usage and examples can be described in detail

id

inputs

name = None
> Name of the tool, this should be a descriptive, human readable name.

namespace = None
> The namespace this tools lives in, this will be set by the ToolManager on load

node_class = None
> Class for of the Node to use

ns_id
> The namespace and id of the Tool

outputs

path
> The path of the directory in which the tool definition file was located.

references = None
> A list of documents and in depth reading about the methods used in this tool

requirements = None
> Requirements for this Tool

> > **Warning:** Not yet implemented

tags = None
> List of tags for this tool

target
> The OS and arch matched target definition.

test (*reference=None*)
> Run the tests for this tool

test_spec
> alias of `TestSpecification`

classmethod test_tool (*reference_data_dir*, *tool=None*, *input_data=None*)
> Execute the tool with the input data specified and test the results against the refence data. This effectively tests the tool execution.
>
> > **Parameters**
> >
> > - **reference_data_dir** (*str*) – The path or vfs url of reference data to compare with
> >
> > - **source_data** (*dict*) – The source data to use

url = None
> URL to website where this tool can be downloaded from

version = None
> Version of the tool, not of the underlying software

version Module

Module containing the class that represent versions

class fastr.core.version.**Version**

    Bases: [tuple](#)

    Class representing a software version definition. Allows for sorting and extraction of parts.

    __dict__ = mappingproxy({'__module__': 'fastr.core.version', '__doc__': '\n Class re

    __module__ = 'fastr.core.version'

    static __new__(*cls*, **version*)

        Class containing a version

        Can be constructed by:

```
Version( 'major.$minor.$extra[0].$extra[1]$seperator$status$build$suffix' )
Version( major, minor, extra, status, build, suffix, seperator )
Version( (major, minor, extra, status, build, suffix, seperator) )
Version( [major, minor, extra, status, build, suffix, seperator] )
```

        Parameters

- **major** (*int*) – interger giving major version
- **minor** (*int*) – is an integer (required)
- **extra** (*list of int*) – is a list of integers
- **status** (*str*) – can be "a", "alpha", "b", "beta", "rc", or "r"
- **build** (*int*) – is an integer
- **suffix** (*str*) – can contain any combination of alpha-numeric character and "._-"
- **seperator** (*str*) – is any of ".", "-", or "_", which is located between $extra and $build

---

        Note: The method based on strings is the recommended method. For strings the major and minor version are required, where for tuple and list constructors all seven elements are optional.

        Examples:

```
>>> a = Version('0.1')
>>> print(tuple(a))
(0, 1, None, None, None, '', None)
>>> b = Version('2.5.3-rc2')
>>> print(tuple(b))
(2, 5, [3], 'rc', 2, '', '-')
>>> c = Version('1.2.3.4.5.6.7-beta8_with_suffix')
>>> print(tuple(c))
(1, 2, [3, 4, 5, 6, 7], 'beta', 8, '_with_suffix', '-')
```

    __repr__()

        Return a in-editor representation of the version

    __str__()

        Return a string representation of the version

---

build
> the build number, this is following the status (e.g. for 3.2-beta4, this would be 4)

date_version_matcher = re.compile('(\\d+)-(\\d+)-(\\d+)([_\\\-\\.])?(.*)')

extra
> extra version extension as a list

extra_string
> extra version extension as a string

major
> major version

minor
> minor version

status
> the status of the version (a, alpha, b, beta, rc or r)

suffix
> the remainder of the version which was not formatted in a known way

version_matcher = re.compile('(\\d+)\\.(\\d+)((?:\\.\\d+)+)?([_\\\-\\.])?(a(?=\\d)|b(?=

`vfs` Module

This module contains the virtual file system code. This is internally used object as used as base class for the IOPlugin.

class fastr.core.vfs.**VirtualFileSystem**
> Bases: `object`

The virtual file system class. This is an IOPlugin, but also heavily used internally in fastr for working with directories. The VirtualFileSystem uses the `vfs://` url scheme.

A typical virtual filesystem url is formatted as `vfs://mountpoint/relative/dir/from/mount.ext`

Where the `mountpoint` is defined in the *Config file*. A list of the currently known mountpoints can be found in the `fastr.config` object

```
>>> fastr.config.mounts
{'example_data': '/home/username/fastr-feature-documentation/fastr/fastr/examples/
→data',
 'home': '/home/username/',
 'tmp': '/home/username/FastrTemp'}
```

This shows that a url with the mount `home` such as `vfs://home/tempdir/testfile.txt` would be translated into `/home/username/tempdir/testfile.txt`.

There are a few default mount points defined by Fastr (that can be changed via the config file).

mountpoint	default location
home	the users home directory (`expanduser('~/')`)
tmp	the fastr temprorary dir, defaults to `tempfile.gettempdir()`
example_data	the fastr example data directory, defaults `$FASTRDIR/example/data`

__dict__ = mappingproxy({'__module__': 'fastr.core.vfs', '__doc__': "\n The virtual

---

__init__()
> Instantiate the VFS plugin

>> **Returns** the VirtualFileSysten plugin

__module__ = 'fastr.core.vfs'

__weakref__
> list of weak references to the object (if defined)

abstract = False

static copy_file_dir(*inpath*, *outpath*)
> Helper function, copies a file or directory not caring what the inpath actually is

>> **Parameters**

>>> • **inpath** – path of the things to be copied

>>> • **outpath** – path of the destination

>> **Returns** the result of shutil.copy2 or shutil.copytree (depending on inpath pointing to a file or directory)

expand_url(*url*)
> Try to expand the url. For vfs with will return the original url.

>> **Parameters** **url** – url to expand

>> **Returns** the expanded url (same as url)

fetch_url(*inurl*, *outpath*)
> Fetch the files from the vfs.

>> **Parameters**

>>> • **inurl** – url to the item in the data store, starts with `vfs://`

>>> • **outpath** – path where to store the fetch data locally

fetch_value(*inurl*)
> Fetch a value from an external vfs file.

>> **Parameters** **inurl** – url of the value to read

>> **Returns** the fetched value

path_to_url(*path*, *mountpoint=None*, *scheme=None*)
> Construct an url from a given mount point and a relative path to the mount point.

>> **Parameters** **path** (`str`) – the path to find the url for

>> **Mountpoint str** mountpoint the url should be under

>> **Returns** url of the

put_url(*inpath*, *outurl*)
> Put the files to the external data store.

>> **Parameters**

>>> • **inpath** – path of the local data

>>> • **outurl** – url to where to store the data, starts with `vfs://`

put_value(*value*, *outurl*)
> Put the value in the external data store.

Parameters
- **value** – value to store
- **outurl** – url to where to store the data, starts with `vfs://`

scheme

setup()
> The plugin setup, does nothing but needs to be implemented

url_to_path(*url*, *scheme=None*)
> Get the path to a file from a vfs url

> **Parameters** **url** (*str*) – url to get the path for

> **Returns** the matching path

> **Return type** str

> **Raises**
> - **_FastrMountUnknownError_** – if the mount in url is unknown
> - **_FastrUnknownURLSchemeError_** – if the url scheme is not correct

> Example (the mountpoint tmp points to /tmp):

```
>>> fastr.vfs.url_to_path('vfs://tmp/file.ext')
'/tmp/file.ext'
```

Subpackages

test Package

test Package

test_datatypemanager Module

test_dimension Module

test_samples Module

test_tool Module

test_version Module

test_vfs Module

data Package

data Package

Package containig data related modules

url **Module**

Module providing tools to parse and create valid urls and paths.

usage example:

When in fastr.config under the mounts section the data mount is set to /media/data, you will get the following. .. code-block:: python

```
>>> from fastr.data.url import get_path_from_url
>>> get_path_from_url('vfs://data/temp/blaat1.png')
'/media/data/temp/blaat1.png'
```

fastr.data.url.**basename**(*url*)
> Get basename of url

>> **Parameters url** (*str*) – the url

>> **Returns** the basename of the path in the url

fastr.data.url.**create_vfs_url**(*mountpoint*, *path*)
> Construct an url from a given mount point and a relative path to the mount point.

>> **Parameters**

>>> • **mountpoint** (*str*) – the name of the mountpoint

>>> • **path** (*str*) – relative path from the mountpoint

>> **Returns** the created vfs url

fastr.data.url.**dirname**(*url*)
> Get the dirname of the url

>> **Parameters url** (*str*) – the url

>> **Returns** the dirname of the path in the url

fastr.data.url.**dirurl**(*url*)
> Get the a new url only having the dirname as the path

>> **Parameters url** (*str*) – the url

>> **Returns** the modified url with only dirname as path

fastr.data.url.**full_split**(*urlpath*)
> Split the path in the url in a list of parts

>> **Parameters urlpath** – the url path

>> **Returns** a list of parts

fastr.data.url.**get_path_from_url**(*url*)
> Get the path to a file from a url. Currently supports the file:// and vfs:// scheme's

> Examples:

```
>>> url.get_path_from_url('vfs://neurodata/user/project/file.ext')
'Y:\neuro3\user\project\file.ext'


>>> 'file:///d:/data/project/file.ext'
'd:\data\project\file.ext'
```

> **Warning:** file:// will not function cross platform and is mainly for testing

fastr.data.url.**get_url_scheme**(*url*)

    Get the schem of the url

        Parameters url (*str*) – url to extract scheme from

        Returns the url scheme

        Return type str

fastr.data.url.**isurl**(*string*)

    Check if string is a valid url

        Parameters string (*str*) – potential url

        Returns flag indicating if string is a valid url

fastr.data.url.**join**(*url*, **p*)

    Join the path in the url with p

        Parameters

            • **url** (*str*) – the base url to join with

            • **p** – additional parts of the path

        Returns the url with the parts added to the path

fastr.data.url.**normurl**(*url*)

    Normalized the path of the url

        Parameters url (*str*) – the url

        Returns the normalized url

fastr.data.url.**register_url_scheme**(*scheme*)

    Register a custom scheme to behave http like. This is needed to parse all things properly.

fastr.data.url.**split**(*url*)

    Split a url in a url with the dirname and the basename part of the path of the url

        Parameters url (*str*) – the url

        Returns a tuple with (dirname_url, basename)

datatypes Package

datatypes Package

The datatypes module holds all DataTypes generated by fastr and all the base classes for these datatypes.

class fastr.datatypes.**AnyFile**(*value=None*)

    Bases: *fastr.datatypes.TypeGroup*

    Special Datatype in fastr that is a TypeGroup with all known DataTypes as its members.

    __abstractmethods__ = frozenset()

    __module__ = 'fastr.datatypes'

    description = 'TypeGroup AnyFile\nAnyFile (AnyFile) is a group of consisting of all URI

class fastr.datatypes.**AnyType**(*value=None*)

    Bases: *fastr.datatypes.TypeGroup*

    Special Datatype in fastr that is a TypeGroup with all known DataTypes as its members.

    __abstractmethods__ = frozenset()

    __module__ = 'fastr.datatypes'

    description = 'TypeGroup AnyType\nAnyType (AnyType) is a group of consisting of all Da

class fastr.datatypes.**BaseDataType**(*value=None*, *format_=None*)

    Bases: fastr.abc.baseplugin.BasePlugin

    The base class for all datatypes in the fastr type system.

    __abstractmethods__ = frozenset({'__init__'})

    __eq__(*other*)

        Test the equality of two DataType objects

            Parameters other (DataType) – the object to compare against

            Returns flag indicating equality

            Return type bool

    __getstate__()

    __hash__ = None

    __init__(*value=None*, *format_=None*)

        The BaseDataType constructor.

            Parameters

                • **value** – value to assign to the new BaseDataType object

                • **format** – the format used for the ValueType

            Returns new BaseDataType object

            Raises *FastrNotImplementedError* – if *id*, *name*, *version* or *description* is None

    __module__ = 'fastr.datatypes'

    __ne__(*other*)

        Test if two objects are not equal. This is by default done by negating the __eq__ operator

            Parameters other (DataType) – the object to compare against

            Returns flag indicating equality

            Return type bool

    __reduce_ex__(**args*, ***kwargs*)

        helper for pickle

    __repr__()

        Returns string representation of the BaseDataType

            Returns string represenation

            Return type str

    __setstate__(*state*)

    __str__()

        Returns the string version of the BaseDataType

> > **Returns** string version
>
> > **Return type** str

checksum()
> Generate a checksum for the value of this DataType
>
> > **Returns** the checksum of the value
>
> > **Return type** str

description = ''
> Description of the DataType

dot_extension = None

extension = None
> Extension related to the Type

filename = '/home/docs/checkouts/readthedocs.org/user_builds/fastr/envs/3.0.0/lib/pyth

fullid = 'fastr://types/BaseDataType'

id = 'BaseDataType'

classmethod isinstance(*value*)
> Indicate whether value is an instance for this DataType.
>
> > **Returns** the flag indicating the value is of this DataType
>
> > **Return type** bool

name = 'BaseDataType'

parent = DataTypeManager AnalyzeImageFile : <URLType: AnalyzeImageFile> AnyFile : <'

parsed_value
> The parsed value of object instantiation of this DataType.

raw_value
> The raw value of object instantiation of this DataType. For datatypes that override value (like Deferred) this is the way to access the _value field.

classmethod test()
> Define the test for the BasePluginManager. Make sure we are not one of the base classes

valid
> A boolean flag that indicates weather or not the value assigned to this DataType is valid. This property is generally overwritten by implementation of specific DataTypes.

value
> The value of object instantiation of this DataType.

version = <Version: 1.0>
> Version of the DataType definition

class fastr.datatypes.**DataType**(*value=None*, *format_=None*)
> Bases: *fastr.datatypes.BaseDataType*
>
> This class is the base class for all DataTypes that can hold a value.
>
> **__abstractmethods__ = frozenset({'__init__'})**
>
> **__init__**(*value=None*, *format_=None*)
> > The DataType constructor.
> >
> > > **Parameters**

- **value** – value to assign to the new DataType object

- **format** – the format used for the ValueType

> **Returns** new DataType object

__module__ = 'fastr.datatypes'

action(*name*)

This function can be overwritten by subclasses to implement certain action that should be performed. For example, the *Directory* DataType has an action *ensure*. This method makes sure the Directory exists. A Tool can indicate an action that should be called for an Output which will be called before execution.

> **Parameters name** (`str`) – name of the action to execute

> **Returns** None

class fastr.datatypes.**DataTypeManager**

Bases: fastr.abc.basepluginmanager.BasePluginManager

The DataTypeManager hold a mapping of all DataTypes in the fast system and can create new DataTypes from files/data structures.

__abstractmethods__ = frozenset()

__init__()

The DataTypeManager constructor will create a new DataTypeManager and populate it with all DataTypes it can find in the paths set in config.types_path.

> **Returns** the created DataTypeManager

__keytransform__(*key*)

Key transformation for this mapping. The key transformation allows indexing by both the DataType name as well as the DataType it self.

> **Parameters key** (`fastr.datatypes.BaseDataType or str`) – The name of the requested datatype or the datatype itself

> **Returns** The requested datatype

__module__ = 'fastr.datatypes'

create_enumtype(*type_id*, *options*, *name=None*)

Create a python class based on an XML file. This function return a completely functional python class based on the contents of a DataType XML file.

Such a class will be of type EnumType.

> **Parameters**

> - **type_id** (`str`) – the id of the new class

> - **options** (`iterable`) – an iterable of options, each option should be str

> **Returns** the newly created subclass of EnumType

> **Raises** *FastrTypeError* – if the options is not an iterable of str

fullid

The fullid of the datatype manager

get_type(*name*)

Read a type given a typename. This will scan all directories in types_path and attempt to load the newest version of the DataType.

> **Parameters name** (`str`) – Name of the datatype that should be imported in the system

Returns the datatype with the requested name, or None if datatype is not found

---

Note: If type is already in TypeManager it will not load anything and return the already loaded version.

---

guess_type(*value*, *exists=True*, *options=None*, *preferred=None*)
 Guess the DataType based on a value str.

> **Parameters**
>
> - **value** (`str`) – the value to guess the type for
>
> - **options** (`TypeGroup, DataType or tuple of DataTypes`) – The options that are allowed to be guessed from
>
> - **extists** (`bool`) – Indicate the value exists (if file) and can be checked for validity, if false skip validity check
>
> - **preferred** (`iterable`) – An iterable of preferred types in case multiple types match.
>
> **Returns** The resulting DataType or None if no match was found
>
> **Raises** *FastrTypeError* – if the options argument is of the wrong type

The function will first create a list of all candidate DataTypes. Subsequently, it will check for each candidate if the value would valid. If there are multiple matches, the config value for preferred types is consulted to break the ties. If non of the DataTypes are in the preferred types list, a somewhat random DataType will be picked as the most optimal result.

has_type(*name*)
 Check if the datatype with requested name exists

> **Parameters name** (`str`) – the name of the requested datatype
>
> **Returns** flag indicating if the datatype exists
>
> **Return type** bool

static isdatatype(*item*)
 Check if item is a valid datatype for the fastr system.

> **Parameters item** – item to check
>
> **Returns** flag indicating if the item is a fastr datatype
>
> **Return type** bool

match_types(**args*, ***kwargs*)
 Find the match between a list of DataTypes/TypeGroups, see resolve-datatype for details

> **Parameters**
>
> - **args** – A list of DataType/TypeGroup objects to match
>
> - **kwargs** – A 'preferred' keyword argument can be used to indicate a list of DataTypes to prefer in case of ties (first has precedence over later in list)
>
> **Returns** The best DataType match, or None if no match is possible.
>
> **Raises** *FastrTypeError* – if not all args are subclasses of BaseDataType

match_types_any(**args*)
 Find the match between a list of DataTypes/TypeGroups, see resolve-datatype for details

> **Parameters args** – A list of DataType/TypeGroup objects to match

---

> **Returns** A set with all DataTypes that match.
>
> **Return type** set
>
> **Raises** *FastrTypeError* – if not all args are subclasses of BaseDataType

plugin_class
> The PluginClass of the items of the BasePluginManager

poll_datatype(*filename*)
> Poll an xml file to see if there is a definition of a datatype in it.
>
> > **Parameters filename** (*str*) – path of the file to poll
> >
> > **Returns** tuple with (id, version, basetype) if a datatype is found or (None, None, None) if no datatype is found

populate()
> Populate Manager. After scanning for DataTypes, create the AnyType and set the preferred types

preferred_types

class fastr.datatypes.**Deferred**(*value=None*, *format_=None*)
> Bases: *fastr.datatypes.DataType*

__abstractmethods__ = frozenset()

__getstate__()

__init__(*value=None*, *format_=None*)
> The Deferred constructor.
>
> > **Parameters**
> >
> > - **value** – value to assign to the new DataType object
> >
> > - **format** – This is ignore but here for compatibility
> >
> > **Returns** new Deferred object

__module__ = 'fastr.datatypes'

__repr__()
> Returns string representation of the BaseDataType
>
> > **Returns** string represenation
> >
> > **Return type** str

__setstate__(*state*)

checksum()
> Generate a checksum for the value of this DataType
>
> > **Returns** the checksum of the value
> >
> > **Return type** str

job

classmethod lookup(*value*)
> Look up the deferred target and return that object
>
> > **Param** value
> >
> > **Returns** The value the deferred points to
> >
> > **Return type** *DataType*

Raises

- *[**FastrKeyError**](#)* – if the deferred is not available (yet)

- *[**FastrValueError**](#)* – if the value is not a valid deferrred url

parsed_value
    The value of object instantiation of this DataType.

provenance

target
    Target object for this deferred.

Raises

- *[**FastrKeyError**](#)* – if the deferred is not available (yet)

- *[**FastrValueError**](#)* – if the value is not a valid deferrred url

value
    The value of object instantiation of this DataType.

class fastr.datatypes.**EnumType**(*value=None*, *format_=None*)
    Bases: *[fastr.datatypes.DataType](#)*

The EnumType is the base for DataTypes that can have a value which is an option from a predefined set of possibilities (similar to an enum type in many programming languages).

__abstractmethods__ = frozenset()

__init__(*value=None*, *format_=None*)
    The EnumType constructor.

Parameters

- **value** – value to assign to the new EnumType object

- **format** – the format used for the ValueType

Returns new EnumType object

Raises *[FastrDataTypeNotInstantiableError](#)* – if not subclassed

__module__ = 'fastr.datatypes'

__reduce_ex__(**args*, ***kwargs*)
    helper for pickle

description = 'EnumType (EnumType) is a enumerate type with options:\n\n\nEnumType can

options = frozenset()

version = <Version: 1.0>
    Enums always have version 1.0

class fastr.datatypes.**TypeGroup**(*value=None*)
    Bases: *[fastr.datatypes.BaseDataType](#)*

The TypeGroup is a special DataType that does not hold a value of its own but is used to group a number of DataTypes. For example ITK has a list of supported file formats that all tools build on ITK support. A group can be used to conveniently specify this in multiple Tools that use the same set DataTypes.

__abstractmethods__ = frozenset({'_members'})

__init__(*value=None*)
    Dummy constructor. TypeGroups are not instantiable and cannot hold a value of its own.

> **Raises** *FastrDataTypeNotInstantiableError* – if called

__module__ = 'fastr.datatypes'

static __new__(*cls*, *value=None*, *format_=None*)
> Instantiate a TypeGroup. This will for match the value to the best matching type and instantiate that. Not that the returned object will not be of type TypeGroup but one of the TypeGroup members.

classmethod isinstance(*value*)
> Indicate whether value is an instance for this DataType.

> > **Returns** the flag indicating the value is of this DataType

> > **Return type** bool

members
> A descriptor that can act like a property for a class.

preference
> A descriptor that can act like a property for a class.

class fastr.datatypes.**URLType**(*value=None*, *format_=None*)
> Bases: *fastr.datatypes.DataType*

The URLType is the base for DataTypes that point to a resource somewhere else (typically a filesystem). The true value is actually the resource referenced by the value in this object.

__abstractmethods__ = frozenset()

__eq__(*other*)
> Test the equality of two DataType objects

> > **Parameters other** (URLType) – the object to compare against

> > **Returns** flag indicating equality

> > **Return type** bool

__hash__ = None

__init__(*value=None*, *format_=None*)
> The URLType constructor

> > **Parameters**

> > > • **value** – value to assign to the new URLType

> > > • **format** – the format used for the ValueType

> > **Returns** new URLType object

__module__ = 'fastr.datatypes'

checksum()
> Return the checksum of this URL type

> > **Returns** checksum string

> > **Return type** str

classmethod content(*inval*, *outval=None*)
> Give the contents of a URLType, this is generally useful for filetypes that consists of multiple files (e.g. AnalyzeImageFile, DICOM). The value will indicate the main file, and the contents function can determine all files that form a single data value.

> > **Parameters**

- **inval** – a value to figure out contents for this type

- **outval** – the place where the copy should point to

Returns a list of all files part of the value (e.g. header and data file)

Return type list

parsed_value
The parsed value of object instantiation of this DataType.

valid
A boolean flag that indicates weather or not the value assigned to this DataType is valid. This property is generally overwritten by implementation of specific DataTypes.

class fastr.datatypes.**ValueType**(*value=None*, *format_=None*)
Bases: *fastr.datatypes.DataType*

The ValueType is the base for DataTypes that hold simple values (not an EnumType and not a file/URL). The values is generally represented by a string.

__abstractmethods__ = frozenset()

__init__(*value=None*, *format_=None*)
The ValueType constructor

Parameters

- **value** – value to assign to the new ValueType

- **format** – the format used for the ValueType

Returns new ValueType object

__module__ = 'fastr.datatypes'

fastr.datatypes.**fastr_isinstance**(*obj*, *datatype*)
Check if an object is of a specific datatype.

Parameters

- **obj** – Object to inspect

- **datatype** (*tuple,* BaseDataType) – The datatype(s) to check

Returns flag indicating object is of datatype

Return type bool

execution Package

execution Package

This package contains all modules related directly to the execution

basenoderun Module

class fastr.execution.basenoderun.**BaseNodeRun**
Bases: fastr.abc.updateable.Updateable, fastr.abc.serializable.Serializable

NODE_RUN_MAP = {'AdvancedFlowNode': <class 'fastr.execution.flownoderun.AdvancedFlowN

NODE_RUN_TYPES = {'AdvancedFlowNodeRun': <class 'fastr.execution.flownoderun.Advanced

__abstractmethods__ = frozenset({'_update'})

classmethod __init_subclass__ (***kwargs*)
    Register nodes in class for easly location

__module__ = 'fastr.execution.basenoderun'

environmentmodules Module

This module contains a class to interact with EnvironmentModules

class fastr.execution.environmentmodules.**EnvironmentModules** (*protected=None*)
    Bases: `object`

    This class can control the module environments in python. It can list, load and unload environmentmodules. These modules are then used if subprocess is

        called from python.

    __dict__ = mappingproxy({'__module__': 'fastr.execution.environmentmodules', '__doc__

    __init__ (*protected=None*)
        Create the environmentmodules control object

            Parameters **protected** (*list*) – list of modules that should never be unloaded

            Returns newly created EnvironmentModules

    __module__ = 'fastr.execution.environmentmodules'

    __repr__ ()
        Return repr(self).

    __weakref__
        list of weak references to the object (if defined)

    avail (*namestart=None*)
        Print available modules in same way as commandline version

            Parameters **namestart** – filter on modules that start with namestart

    avail_modules
        List of avaible modules

    clear ()
        Unload all modules (except the protected modules as they cannot be unloaded). This should result in a clean environment.

    isloaded (*module*)
        Check if a specific module is loaded

            Parameters **module** – module to check

            Returns flag indicating the module is loaded

    load (*module*)
        Load specified module

            Parameters **module** – module to load

    loaded_modules
        List of currently loaded modules

reload(*module*)
> Reload specified module

>> **Parameters** **module** – module to reload

swap(*module1*, *module2*)
> Swap one module for another one

>> **Parameters**

>>> • **module1** – module to unload

>>> • **module2** – module to load

sync()
> Sync the object with the underlying environment. Re-checks the available and loaded modules

static tostring_modvalue(*value*)
> Turn a representation of a module into a string representation

>> **Parameters** **value** – module representation (either str or tuple)

>> **Returns** string representation

static totuple_modvalue(*value*)
> Turn a representation of a module into a tuple representation

>> **Parameters** **value** – module representation (either str or tuple)

>> **Returns** tuple representation (name, version, default)

unload(*module*)
> Unload specified module

>> **Parameters** **module** – module to unload

class fastr.execution.environmentmodules.**ModuleSystem**
> Bases: enum.Enum

> An enumeration.

> **__module__ = 'fastr.execution.environmentmodules'**

> **envmod = 'enviromentmodules'**

> **lmod = 'Lmod'**

executionscript Module

The executionscript is the script that wraps around a tool executable. It takes a job, builds the command, executes the command (while profiling it) and collects the results.

fastr.execution.executionscript.**execute_job**(*job*)
> Execute a Job and save the result to disk

>> **Parameters** **job** – the job to execute

fastr.execution.executionscript.**main**(*joblist=None*)
> This is the main code. Wrapped inside a function to avoid the variables being seen as globals and to shut up pylint. Also if the joblist argument is given it can run any given job, otherwise it takes the first command line argument.

flownoderun Module

class fastr.execution.flownoderun.**FlowNodeRun**(*node*, *parent*)

> Bases: *fastr.execution.noderun.NodeRun*

> A Flow NodeRun is a special subclass of Nodes in which the amount of samples can vary per Output. This allows non-default data flows.

> **__abstractmethods__ = frozenset()**

> **__module__ = 'fastr.execution.flownoderun'**

> **blocking**
>> A FlowNodeRun is (for the moment) always considered blocking.
>>
>>> **Returns** True

> **dimnames**
>> Names of the dimensions in the NodeRun output. These will be reflected in the SampleIdList of this NodeRun.

> **outputsize**
>> Size of the outputs in this NodeRun

> **set_result**(*job*, *failed_annotation*)
>> Incorporate result of a job into the FlowNodeRun.
>>
>>> **Parameters job** (*Type*) – job of which the result to store

class fastr.execution.flownoderun.**AdvancedFlowNodeRun**(*node*, *parent*)

> Bases: *fastr.execution.flownoderun.FlowNodeRun*

> **__abstractmethods__ = frozenset()**

> **__module__ = 'fastr.execution.flownoderun'**

> **execute**()
>> Execute the node and create the jobs that need to run
>>
>>> **Returns** list of jobs to run
>>>
>>> **Return type** list of *Jobs*

> **set_result**(*job*, *failed_annotation*)
>> Incorporate result of a job into the FlowNodeRun.
>>
>>> **Parameters job** (*Type*) – job of which the result to store

inputoutputrun Module

Classes for arranging the input and output for nodes.

Exported classes:

Input – An input for a node (holding datatype). Output – The output of a node (holding datatype and value). ConstantOutput – The output of a node (holding datatype and value).

> **Warning:** Don't mess with the Link, Input and Output internals from other places. There will be a huge chances of breaking the network functionality!

class fastr.execution.inputoutputrun.**AdvancedFlowOutputRun**(*node_run*, *template*)
    Bases: *fastr.execution.inputoutputrun.OutputRun*

    __abstractmethods__ = frozenset()

    __module__ = 'fastr.execution.inputoutputrun'

class fastr.execution.inputoutputrun.**BaseInputRun**(*node_run*, *template*)
    Bases: *fastr.core.samples.HasSamples*, *fastr.planning.inputoutput.BaseInput*

    Base class for all inputs runs.

    __abstractmethods__ = frozenset({'__getitem__', 'dimensions', '_update', 'itersubinput

    __init__(*node_run*, *template*)
        Instantiate a BaseInput

            Parameters

                • **node** – the parent node the input/output belongs to.

                • **description** – the ParameterDescription describing the input/output.

            Returns the created BaseInput

            Raises

                • *FastrTypeError* – if description is not of class ParameterDescription

                • *FastrDataTypeNotAvailableError* – if the DataType requested cannot be found in the fastr.types

    __module__ = 'fastr.execution.inputoutputrun'

    itersubinputs()
        Iterator over the SubInputs

            Returns iterator

        example:

```
>>> for subinput in input_a.itersubinputs():
        print subinput
```

class fastr.execution.inputoutputrun.**InputRun**(*node_run*, *template*)
    Bases: *fastr.execution.inputoutputrun.BaseInputRun*

    Class representing an input of a node. Such an input will be connected to the output of another node or the output of an constant node to provide the input value.

    __abstractmethods__ = frozenset()

    __getitem__(*key*)
        Retrieve an item from this Input.

            Parameters key (str, SampleId or tuple) – the key of the requested item, can be a key str, sample index tuple or a SampleId

            Returns the return value depends on the requested key. If the key was an int the corresponding *SubInput* will be returned. If the key was a SampleId or sample index tuple, the corresponding SampleItem will be returned.

            Return type SampleItem or *SubInput*

            Raises

                • *FastrTypeError* – if key is not of a valid type

> • *`FastrKeyError`* – if the key is not found

`__getstate__` ()
    Retrieve the state of the Input

> > **Returns** the state of the object
>
> > **Rtype dict**

`__init__` (*node_run*, *template*)
    Instantiate an input.

> > **Parameters `template`** – the Input that the InputRun is based on

`__module__` = `'fastr.execution.inputoutputrun'`

`__setstate__` (*state*)
    Set the state of the Input by the given state.

> > **Parameters `state`** (*`dict`*) – The state to populate the object with
>
> > **Returns** None

`__str__` ()
    Get a string version for the Input

> > **Returns** the string version
>
> > **Return type** str

`cardinality` (*key=None*, *job_data=None*)
    Cardinality for an Input is the sum the cardinalities of the SubInputs, unless defined otherwise.

> > **Parameters `key`** (tuple of int or `SampleId`) – key for a specific sample, can be sample index or id
>
> > **Returns** the cardinality
>
> > **Return type** int, sympy.Symbol, or None

`datatype`
    The datatype of this Input

`dimensions`
    The size of the sample collections that can accessed via this Input.

`fullid`
    The full defining ID for the Input

`get_sourced_nodes` ()
    Get a list of all *Nodes* connected as sources to this Input

> > **Returns** list of all connected *Nodes*
>
> > **Return type** list

`get_sourced_outputs` ()
    Get a list of all *Outputs* connected as sources to this Input

> > **Returns** tuple of all connected *Outputs*
>
> > **Return type** tuple

`index` (*value*)
    Find index of a SubInput

> > **Parameters `value`** (*SubInput*) – the *SubInput* to find the index of

> **Returns** key
>
> **Return type** int, str

input_group
> The id of the InputGroup this Input belongs to.

insert(*index*)
> Insert a new SubInput at index in the sources list
>
> > **Parameters key** (*int*) – positive integer for position in _source list to insert to
> >
> > **Returns** newly inserted *SubInput*
> >
> > **Return type** *SubInput*

itersubinputs()
> Iterate over the *SubInputs* in this Input.
>
> > **Returns** iterator yielding *SubInput*
>
> example:

```
>>> for subinput in input_a.itersubinputs():
        print subinput
```

remove(*value*)
> Remove a SubInput from the SubInputs list.
>
> > **Parameters value** (*SubInput*) – the *SubInput* to removed from this Input

source
> The mapping of *SubInputs* that are connected and have more than 0 elements.

class fastr.execution.inputoutputrun.**MacroOutputRun**(*node_run*, *template*)
> Bases: *fastr.execution.inputoutputrun.OutputRun*

__abstractmethods__ = frozenset()

__module__ = 'fastr.execution.inputoutputrun'

dimensions
> The dimensions has to be implemented by any subclass. It has to provide a tuple of Dimensions.
>
> > **Returns** dimensions
> >
> > **Return type** tuple

class fastr.execution.inputoutputrun.**OutputRun**(*node_run*, *template*)
> Bases: *fastr.planning.inputoutput.BaseOutput*, *fastr.core.samples.ContainsSamples*

Class representing an output of a node. It holds the output values of the tool ran. Output fields can be connected to inputs of other nodes.

__abstractmethods__ = frozenset()

__getitem__(*key*)
> Retrieve an item from this Output. The returned value depends on what type of key used:
>
> - Retrieving data using index tuple: [index_tuple]
> - Retrieving data sample_id str: [SampleId]
> - Retrieving a list of data using SampleId list: [sample_id1, ..., sample_idN]
> - Retrieving a *SubOutput* using an int or slice: [n] or [n:m]

> **Parameters key** (int, slice, SampleId or tuple) – the key of the requested item, can be a number, slice, sample index tuple or a SampleId
>
> **Returns** the return value depends on the requested key. If the key was an int or slice the corresponding *SubOutput* will be returned (and created if needed). If the key was a SampleId or sample index tuple, the corresponding SampleItem will be returned. If the key was a list of SampleId a tuple of SampleItem will be returned.
>
> **Return type** *SubInput* or SampleItem or list of SampleItem
>
> **Raises**
>
> - *FastrTypeError* – if key is not of a valid type
>
> - *FastrKeyError* – if the parent Node has not been executed

__getstate__()

Retrieve the state of the Output

> **Returns** the state of the object
>
> **Rtype dict**

__init__(*node_run*, *template*)

Instantiate an Output

> **Parameters**
>
> - **node** – the parent node the output belongs to.
>
> - **description** – the ParameterDescription describing the output.
>
> **Returns** created Output
>
> **Raises**
>
> - *FastrTypeError* – if description is not of class ParameterDescription
>
> - *FastrDataTypeNotAvailableError* – if the DataType requested cannot be found in the fastr.types

__module__ = **'fastr.execution.inputoutputrun'**

__setitem__(*key*, *value*)

Store an item in the Output

> **Parameters**
>
> - **key** (tuple of int or SampleId) – key of the value to store
>
> - **value** – the value to store
>
> **Returns** None
>
> **Raises** *FastrTypeError* – if key is not of correct type

__setstate__(*state*)

Set the state of the Output by the given state.

> **Parameters state** (*dict*) – The state to populate the object with
>
> **Returns** None

__str__()

Get a string version for the Output

> **Returns** the string version

Return type str

automatic
Flag indicating that the Output is generated automatically without being specified on the command line

cardinality(*key=None*, *job_data=None*)
Cardinality of this Output, may depend on the inputs of the parent Node.

Parameters **key** (tuple of int or `SampleId`) – key for a specific sample, can be sample index or id

Returns the cardinality

Return type int, sympy.Symbol, or None

Raises

- **`FastrCardinalityError`** – if cardinality references an invalid *Input*

- **`FastrTypeError`** – if the referenced cardinality values type cannot be case to int

- **`FastrValueError`** – if the referenced cardinality value cannot be case to int

datatype
The datatype of this Output

fullid
The full defining ID for the Output

iterconvergingindices(*collapse_dims*)
Iterate over all data, but collapse certain dimension to create lists of data.

Parameters **collapse_dims** (*iterable of int*) – dimension to collapse

Returns iterator SampleIndex (possibly containing slices)

listeners
The list of *Links* connected to this Output.

preferred_types
The list of preferred `DataTypes` for this Output.

resulting_datatype
The `DataType` that will the results of this Output will have.

samples
The SampleCollection of the samples in this Output. None if the NodeRun has not yet been executed. Otherwise a SampleCollection.

valid
Check if the output is valid, i.e. has a valid cardinality

class fastr.execution.inputoutputrun.**SourceOutputRun**(*node_run*, *template*)
Bases: *fastr.execution.inputoutputrun.OutputRun*

Output for a SourceNodeRun, this type of Output determines the cardinality in a different way than a normal NodeRun.

__abstractmethods__ = frozenset()

__getitem__(*item*)
Retrieve an item from this Output. The returned value depends on what type of key used:

- Retrieving data using index tuple: [index_tuple]

- Retrieving data sample_id str: [SampleId]

- Retrieving a list of data using SampleId list: [sample_id1, . . . , sample_idN]

- Retrieving a [*SubOutput*](#) using an int or slice: [n] or [n:m]

> **Parameters key** (int, slice, SampleId or tuple) – the key of the requested item, can be a
> number, slice, sample index tuple or a SampleId
>
> **Returns** the return value depends on the requested key. If the key was an int or slice the corre-
> sponding [*SubOutput*](#) will be returned (and created if needed). If the key was a SampleId
> or sample index tuple, the corresponding SampleItem will be returned. If the key was a
> list of SampleId a tuple of SampleItem will be returned.
>
> **Return type** [*SubInput*](#) or SampleItem or list of SampleItem
>
> **Raises**
>
> - [*FastrTypeError*](#) – if key is not of a valid type
>
> - [*FastrKeyError*](#) – if the parent NodeRun has not been executed

__init__(*node_run*, *template*)
> Instantiate a FlowOutput
>
> **Parameters**
>
> - **node** – the parent node the output belongs to.
>
> - **description** – the ParameterDescription describing the output.
>
> **Returns** created FlowOutput
>
> **Raises**
>
> - [*FastrTypeError*](#) – if description is not of class ParameterDescription
>
> - [*FastrDataTypeNotAvailableError*](#) – if the DataType requested cannot be found
>   in the fastr.types

__module__ = 'fastr.execution.inputoutputrun'

__setitem__(*key*, *value*)
> Store an item in the Output
>
> **Parameters**
>
> - **key** (tuple of int or SampleId) – key of the value to store
>
> - **value** – the value to store
>
> **Returns** None
>
> **Raises** [*FastrTypeError*](#) – if key is not of correct type

cardinality(*key=None*, *job_data=None*)
> Cardinality of this SourceOutput, may depend on the inputs of the parent NodeRun.
>
> **Parameters key** (tuple of int or SampleId) – key for a specific sample, can be sample index
> or id
>
> **Returns** the cardinality
>
> **Return type** [int](#), sympy.Symbol, or [None](#)

dimensions
> The dimensions of this SourceOutputRun

linearized
> A linearized version of the sample data, this is lazily cached linearized version of the underlying SampleCollection.

ndims
> The number of dimensions in this SourceOutput

size
> The sample size of the SourceOutput

class fastr.execution.inputoutputrun.**SubInputRun**(*input_*)
> Bases: *fastr.execution.inputoutputrun.BaseInputRun*

> This class is used by *Input* to allow for multiple links to an *Input*. The SubInput class can hold only a single Link to a (Sub)Output, but behaves very similar to an *Input* otherwise.

> **__abstractmethods__ = frozenset()**

> **__getitem__**(*key*)
> > Retrieve an item from this SubInput.

> > > **Parameters key** (int, SampleId or SampleIndex) – the key of the requested item, can be a number, sample index tuple or a SampleId

> > > **Returns** the return value depends on the requested key. If the key was an int the corresponding *SubInput* will be returned. If the key was a SampleId or sample index tuple, the corresponding SampleItem will be returned.

> > > **Return type** SampleItem or *SubInput*

> > > **Raises** *FastrTypeError* – if key is not of a valid type

> > ---
> > **Note:** As a SubInput has only one SubInput, only requesting int key 0 or -1 is allowed, and it will return self
> > ---

> **__getstate__**()
> > Retrieve the state of the SubInput

> > > **Returns** the state of the object

> > > **Rtype dict**

> **__init__**(*input_*)
> > Instantiate an SubInput.

> > > **Parameters input** (*Input*) – the parent of this SubInput.

> > > **Returns** the created SubInput

> **__module__ = 'fastr.execution.inputoutputrun'**

> **__setstate__**(*state*)
> > Set the state of the SubInput by the given state.

> > > **Parameters state** (*dict*) – The state to populate the object with

> > > **Returns** None

> **__str__**()
> > Get a string version for the SubInput

> > > **Returns** the string version

> > > **Return type** str

cardinality(*key=None*, *job_data=None*)
    Get the cardinality for this SubInput. The cardinality for a SubInputs is defined by the incoming link.

> **Parameters key** (SampleIndex or SampleId) – key for a specific sample, can be sample index or id
>
> **Returns** the cardinality
>
> **Return type** int, sympy.Symbol, or None

description
    The description object of this input/output

dimensions
    The sample size of the SubInput

fullid
    The full defining ID for the SubInput

get_sourced_nodes()
    Get a list of all *Nodes* connected as sources to this SubInput

> **Returns** list of all connected *Nodes*
>
> **Return type** list

get_sourced_outputs()
    Get a list of all *Outputs* connected as sources to this SubInput

> **Returns** list of all connected *Outputs*
>
> **Return type** list

input_group
    The id of the InputGroup this SubInputs parent belongs to.

iteritems()
    Iterate over the SampleItems that are in the SubInput.

> **Returns** iterator yielding SampleItem objects

itersubinputs()
    Iterate over SubInputs (for a SubInput it will yield self and stop iterating after that)

> **Returns** iterator yielding *SubInput*

    example:

```
>>> for subinput in input_a.itersubinputs():
        print subinput
```

node
    The Node to which this SubInputs parent belongs

source
    A list with the source *Link*. The list is to be compatible with *Input*

source_output
    The *Output* linked to this SubInput

class fastr.execution.inputoutputrun.**SubOutputRun**(*output*, *index*)
    Bases: *fastr.execution.inputoutputrun.OutputRun*

    The SubOutput is an Output that represents a slice of another Output.

    __abstractmethods__ = **frozenset**()

__getitem__(*key*)

Retrieve an item from this SubOutput. The returned value depends on what type of key used:

- Retrieving data using index tuple: [index_tuple]
- Retrieving data sample_id str: [SampleId]
- Retrieving a list of data using SampleId list: [sample_id1, ..., sample_idN]
- Retrieving a [*SubOutput*](#) using an int or slice: [n] or [n:m]

> **Parameters key** (int, slice, SampleId or tuple) – the key of the requested item, can be a number, slice, sample index tuple or a SampleId

> **Returns** the return value depends on the requested key. If the key was an int or slice the corresponding [*SubOutput*](#) will be returned (and created if needed). If the key was a SampleId or sample index tuple, the corresponding SampleItem will be returned. If the key was a list of SampleId a tuple of SampleItem will be returned.

> **Return type** [*SubInput*](#) or SampleItem or list of SampleItem

> **Raises** [*FastrTypeError*](#) – if key is not of a valid type

__getstate__()

Retrieve the state of the SubOutput

> **Returns** the state of the object

> **Rtype dict**

__init__(*output*, *index*)

Instantiate a SubOutput

> **Parameters**
>
> - **output** – the parent output the suboutput slices.
> - **index** ([*int or slice*](#)) – the way to slice the parent output

> **Returns** created SubOutput

> **Raises**
>
> - [*FastrTypeError*](#) – if the output argument is not an instance of [*Output*](#)
> - [*FastrTypeError*](#) – if the index argument is not an int or slice

__len__()

Return the length of the Output.

Note: In a SubOutput this is always 1.

__module__ = 'fastr.execution.inputoutputrun'

__setitem__(*key*, *value*)

A function blocking the assignment operator. Values cannot be assigned to a SubOutput.

> **Raises** [*FastrNotImplementedError*](#) – if called

__setstate__(*state*)

Set the state of the SubOutput by the given state.

> **Parameters state** ([*dict*](#)) – The state to populate the object with

> **Returns** None

__str__()
> Get a string version for the SubOutput
>
>> **Returns** the string version
>>
>> **Return type** str

cardinality(*key=None*, *job_data=None*)
> Cardinality of this SubOutput depends on the parent Output and `self.index`
>
>> **Parameters key** (tuple of int or `SampleId`) – key for a specific sample, can be sample index or id
>>
>> **Returns** the cardinality
>>
>> **Return type** int, sympy.Symbol, or None
>>
>> **Raises**
>>
>> - **_FastrCardinalityError_** – if cardinality references an invalid _Input_
>> - **_FastrTypeError_** – if the referenced cardinality values type cannot be case to int
>> - **_FastrValueError_** – if the referenced cardinality value cannot be case to int

datatype
> The datatype of this SubOutput

fullid
> The full defining ID for the SubOutput

indexrep
> Simple representation of the index.

listeners
> The list of _Links_ connected to this Output.

node
> The NodeRun to which this SubOutput belongs

preferred_types
> The list of preferred `DataTypes` for this SubOutput.

resulting_datatype
> The `DataType` that will the results of this SubOutput will have.

samples
> The `SampleCollection` for this SubOutput

job Module

This module contains the Job class and some related classes.

class fastr.execution.job.**InlineJob**(**args*, ***kwargs*)
> Bases: _fastr.execution.job.Job_
>
> Job that does not actually need to run but is used for consistency in data processing and logging.
>
> **__init__**(**args*, ***kwargs*)
> > Create a job
> >
> >> **Parameters**

- **node** ([Node](#)) – the node the job is based on
- **sample_id** – the id of the sample
- **sample_index** – the index of the sample
- **input_arguments** – the argument list
- **output_arguments** – the argument list
- **hold_jobs** – the jobs on which this jobs depend
- **preferred_types** – The list of preferred types to use

Returns

__module__ = 'fastr.execution.job'

collect_provenance()
  Collect the provenance for this job

get_result()
  Get the result of the job if it is available. Load the output file if found and check if the job matches the current object. If so, load and return the result.

  Returns Job after execution or None if not available

  Return type Job | None

class fastr.execution.job.**Job**(*node*, *sample_id*, *sample_index*, *input_arguments*, *output_arguments*, *hold_jobs=None*, *preferred_types=None*)
  Bases: fastr.abc.serializable.Serializable

  Class describing a job.

  Arguments: tool_name - the name of the tool (str) tool_version - the version of the tool (Version) argument - the arguments used when calling the tool (list) tmpdir - temporary directory to use to store output data hold_jobs - list of jobs that need to finished before this job can run (list)

  COMMAND_DUMP = '__fastr_command__.pickle.gz'

  INFO_DUMP = '__fastr_extra_job_info__.json'

  PROV_DUMP = '__fastr_prov__.json'

  RESULT_DUMP = '__fastr_result__.pickle.gz'

  STDERR_DUMP = '__fastr_stderr__.txt'

  STDOUT_DUMP = '__fastr_stdout__.txt'

  __getstate__()
    Get the state of the job

    Returns job state

    Return type [dict](#)

  __init__(*node*, *sample_id*, *sample_index*, *input_arguments*, *output_arguments*, *hold_jobs=None*, *preferred_types=None*)
    Create a job

    Parameters

    - **node** ([Node](#)) – the node the job is based on
    - **sample_id** ([SampleId](#)) – the id of the sample
    - **sample_index** ([SampleIndex](#)) – the index of the sample

- **input_arguments** (Dict[str, *SampleIndex*]) – the argument list
- **output_arguments** (Dict[str, Dict[~KT, ~VT]]) – the argument list
- **hold_jobs** (Optional[List[str]]) – the jobs on which this jobs depend
- **preferred_types** (Optional[List[~T]]) – The list of preferred types to use

> **Returns**

__module__ = 'fastr.execution.job'

__repr__ ()
    String representation of the Job

__setstate__ (*state*)
    Set the state of the job

> **Parameters state** (*dict*) –

static cast_to_type (*value*, *datatypes*)
    Try to cast value to one of the given datatypes. Will try all the datatypes in order.

> **Parameters datatypes** (*tuple*) – Possible datatypes to cast to
>
> **Return type** *DataType*
>
> **Returns** casted value

clean ()

collect_provenance ()
    Collect the provenance for this job

commandfile
    The path of the command pickle

commandurl
    The url of the command pickle

create_payload ()
    Create the payload for this object based on all the input/output arguments

> **Returns** the payload
>
> **Return type** dict

ensure_tmp_dir ()

execute ()
    Execute this job

> **Returns** The result of the execution
>
> **Return type** InterFaceResult

extrainfofile
    The path where the extra job info document is saved

extrainfourl
    The url where the extra job info document is saved

classmethod fill_output_argument (*output_spec*, *cardinality*, *desired_type*, *requested*, *tm-purl*)
    This is an abstract class method. The method should take the argument_dict generated from calling self.get_argument_dict() and turn it into a list of commandline arguments that represent this Input/Output.

> **Parameters**

- **cardinality** (*int*) – the cardinality for this output (can be non for automatic outputs)
- **desired_type** (*DataType*) – the desired datatype for this output
- **requested** (*bool*) – flag to indicate that the output is requested by Fastr

> **Returns** the values for this output
>
> **Return type** list

fullid
> The full id of the job

get_deferred(*output_id*, *cardinality_nr*, *sample_id=None*)
> Get a deferred pointing to a specific output value in the Job
>
> > **Parameters**
> >
> > - **output_id** (*str*) – the output to select from
> > - **cardinality_nr** (*int*) – the index of the cardinality
> > - **sample_id** (*str*) – the sample id to select (optional)
> >
> > **Returns** The deferred

get_output_datatype(*output_id*)
> Get the datatype for a specific output
>
> > **Parameters** **output_id** (*str*) – the id of the output to get the datatype for
> >
> > **Returns** the requested datatype
> >
> > **Return type** tuple

get_result()
> Get the result of the job if it is available. Load the output file if found and check if the job matches the current object. If so, load and return the result.
>
> > **Returns** Job after execution or None if not available
> >
> > **Return type** Job | None

classmethod get_value(*value*)
> Get a value
>
> > **Parameters**
> >
> > - **value** – the url of the value
> > - **datatype** – datatype of the value
> >
> > **Returns** the retrieved value

hash_inputs()
> Create hashes for all input values and store them in the info store

hash_results()
> Create hashes of all output values and store them in the info store

id
> The id of this job

logfile
> The path of the result pickle

logurl
> The url of the result pickle

provfile
> The path where the prov document is saved

provurl
> The url where the prov document is saved

resources
> The compute resources required for this job

status
> The status of the job

stderrfile
> The path where the stderr text is saved

stderrurl
> The url where the stderr text is saved

stdoutfile
> The path where the stdout text is saved

stdouturl
> The url where the stdout text is saved

tmpurl
> The URL of the tmpdir to use

tool

classmethod translate_argument(*value*)
> Translate an argument from a URL to an actual path.

> **Parameters**
>> • **value** – value to translate
>>
>> • **datatype** – the datatype of the value

> **Returns** the translated value

static translate_output_results(*value*, *datatypes*, *mountpoint=None*)
> Translate the results for on Output

> **Parameters**
>> • **value** – the results value for the output
>>
>> • **datatypes** (*tuple*) – tuple of possible datatypes for the output
>>
>> • **preferred_type** – the preferred datatype of the output

> **Returns** the update value for the result

translate_results(*result*)
> Translate the results of an interface (using paths etc) to the proper form using URI's instead.

> **Parameters** **result** (*dict*) – the result data of an interface

> **Returns** the translated result

> **Return type** dict

validate_results(*payload*)
> Validate the results of the Job

> **Returns** flag indicating the results are complete and valid

---

> **write**()

class fastr.execution.job.**JobCleanupLevel**

> Bases: enum.Enum
>
> The cleanup level for Jobs that are finished.
>
> **__module__ = 'fastr.execution.job'**
>
> **all = 'all'**
>
> **no_cleanup = 'no_cleanup'**
>
> **non_failed = 'non_failed'**

class fastr.execution.job.**JobState**(_, *stage*, *error*)

> Bases: enum.Enum
>
> The possible states a Job can be in. An overview of the states and the adviced transitions are depicted in the following figure:
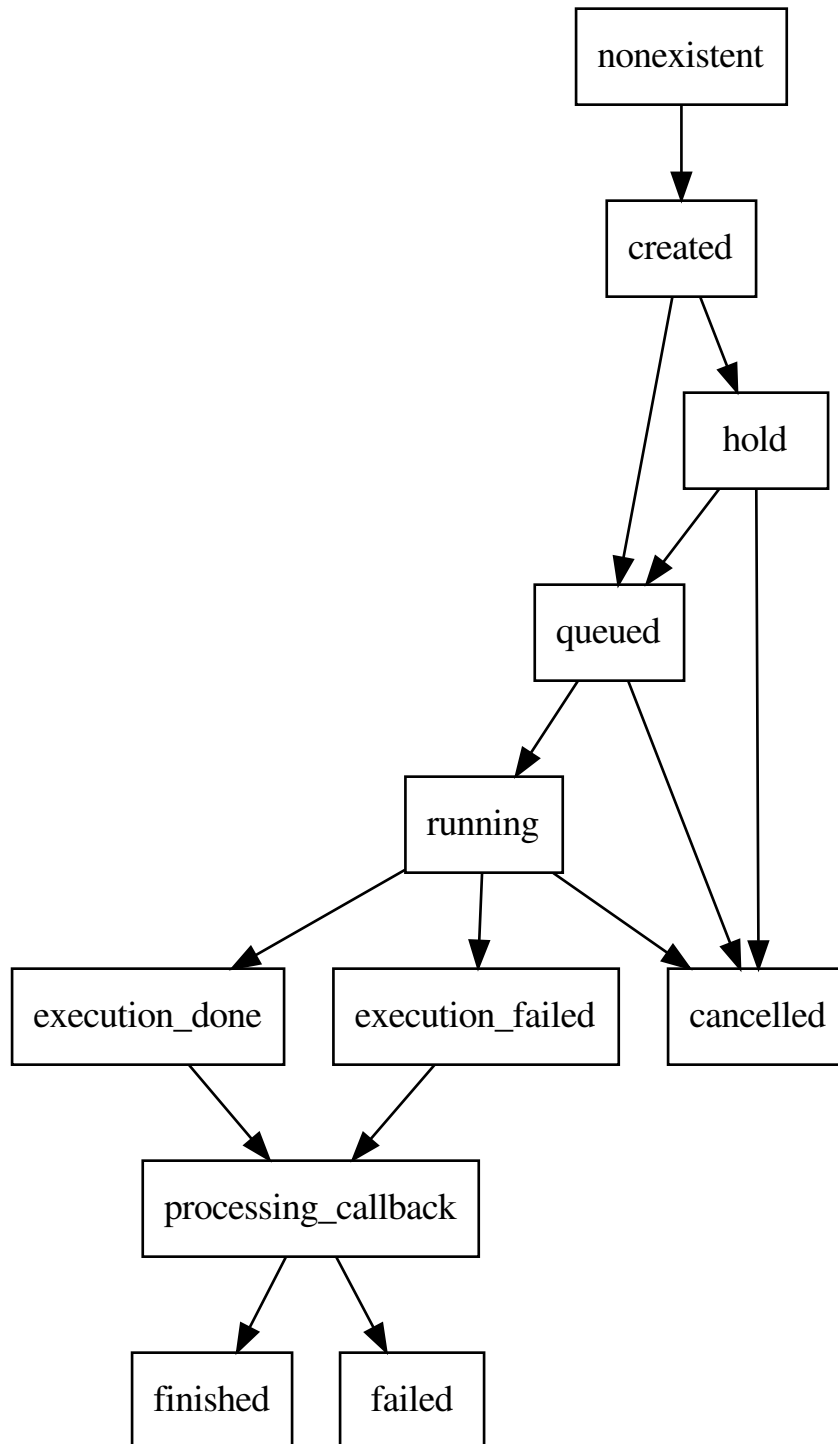
__**init**__ (_, *stage*, *error*)

Initialize self. See help(type(self)) for accurate signature.

__module__ = 'fastr.execution.job'

cancelled = ('cancelled', 'done', True)

created = ('created', 'idle', False)

done

execution_done = ('execution_done', 'in_progress', False)

execution_failed = ('execution_failed', 'in_progress', True)

failed = ('failed', 'done', True)

finished = ('finished', 'done', False)

hold = ('hold', 'idle', False)

idle

in_progress

nonexistent = ('nonexistent', 'idle', False)

processing_callback = ('processing_callback', 'in_progress', False)

queued = ('queued', 'idle', False)

running = ('running', 'in_progress', False)

class fastr.execution.job.**SinkJob**(*node*, *sample_id*, *sample_index*, *input_arguments*, *output_arguments*, *hold_jobs=None*, *substitutions=None*, *preferred_types=None*)

Bases: *fastr.execution.job.Job*

Special SinkJob for the Sink

__init__(*node*, *sample_id*, *sample_index*, *input_arguments*, *output_arguments*, *hold_jobs=None*, *substitutions=None*, *preferred_types=None*)
Create a job

> **Parameters**
>
> - **node** (Node) – the node the job is based on
> - **sample_id** – the id of the sample
> - **sample_index** – the index of the sample
> - **input_arguments** – the argument list
> - **output_arguments** – the argument list
> - **hold_jobs** – the jobs on which this jobs depend
> - **preferred_types** – The list of preferred types to use
>
> **Returns**

__module__ = 'fastr.execution.job'

__repr__()
String representation for the SinkJob

create_payload()
Create the payload for this object based on all the input/output arguments

> **Returns** the payload

> **Return type** dict

get_result()
> Get the result of the job if it is available. Load the output file if found and check if the job matches the current object. If so, load and return the result.
>
> > **Returns** Job after execution

hash_inputs()
> Create hashes for all input values and store them in the info store

id
> The id of this job

substitute(*value*, *datatype=None*)
> Substitute the special fields that can be used in a SinkJob.
>
> > **Parameters**
> >
> > - **value** (`str`) – the value to substitute fields in
> >
> > - **datatype** (`BaseDataType`) – the datatype for the value
> >
> > **Returns** string with substitutions performed
> >
> > **Return type** str

tmpurl
> The URL of the tmpdir to use

validate_results(*payload*)
> Validate the results of the SinkJob
>
> > **Returns** flag indicating the results are complete and valid

class fastr.execution.job.**SourceJob**(*node*, *sample_id*, *sample_index*, *input_arguments*, *output_arguments*, *hold_jobs=None*, *preferred_types=None*)

Bases: *fastr.execution.job.Job*

Special SourceJob for the Source

__module__ = **'fastr.execution.job'**

__repr__()
> String representation for the SourceJob

collect_provenance()
> Collect the provenance for this job

get_output_datatype(*output_id*)
> Get the datatype for a specific output
>
> > **Parameters** **output_id** (`str`) – the id of the output to get the datatype for
> >
> > **Returns** the requested datatype
> >
> > **Return type** *BaseDataType*

hash_inputs()
> Create hashes for all input values and store them in the info store

validate_results(*payload*)
> Validate the results of the Job
>
> > **Returns** flag indicating the results are complete and valid

linkrun **Module**

The link module contain the Link class. This class represents the links in a network. These links lead from an output (BaseOutput) to an input (BaseInput) and indicate the desired data flow. Links are smart objects, in the sense that when you set their start or end point, they register themselves with the Input and Output. They do all the book keeping, so as long as you only set the source and target of the Link, the link should be valid.

> **Warning:** Don't mess with the Link, Input and Output internals from other places. There will be a huge chances of breaking the network functionality!

class fastr.execution.linkrun.**LinkRun**(*link*, *parent=None*)

> Bases: fastr.abc.updateable.Updateable, fastr.abc.serializable.Serializable

> Class for linking outputs ([*BaseOutput*](#)) to inputs ([*BaseInput*](#))

> Examples:

```
>>> import fastr
>>> network = fastr.Network()
>>> link1 = network.create_link( n1.ouputs['out1'], n2.inputs['in2'] )

link2 = Link()
link2.source = n1.ouputs['out1']
link2.target = n2.inputs['in2']
```

> **__abstractmethods__ = frozenset()**

> **__dataschemafile__ = 'Link.schema.json'**

> **__eq__**(*other*)
>> Test for equality between two Links
>>
>>> **Parameters other** ([*LinkRun*](#)) – object to test against
>>>
>>> **Returns** True for equality, False otherwise
>>>
>>> **Return type** [bool](#)

> **__getitem__**(*index*)
>> Get a an item for this Link. The item will be retrieved from the connected output, but a diverging or converging flow can change the number of samples/cardinality.
>>
>>> **Parameters index** ([*SampleIndex*](#)) – index of the item to retrieve
>>>
>>> **Returns** the requested item
>>>
>>> **Return type** SampleItem
>>>
>>> **Raises** [*FastrIndexError*](#) – if the index length does not match the number dimensions in the source data (after collapsing/expanding)

> **__getstate__**()
>> Retrieve the state of the Link
>>
>>> **Returns** the state of the object
>>>
>>> **Rtype dict**

> **__hash__ = None**

> **__init__**(*link*, *parent=None*)
>> Create a new Link in a Network.

Parameters

- **link** (`Link`) – the base link

- **parent** (`Network` or None) – the parent network, if None is given the fastr.current_network is assumed to be the parent

Returns newly created LinkRun

Raises

- **`FastrValueError`** – if parent is not given and *fastr.current_network* is not set

- **`FastrValueError`** – if the source output is not in the same network as the Link

- **`FastrValueError`** – if the target input is not in the same network as the Link

__module__ = **'fastr.execution.linkrun'**

__repr__()
Get a string representation for the Link

> **Returns** the string representation
>
> **Return type** str

__setstate__(*state*)
Set the state of the Link by the given state.

> **Parameters** **state** (*dict*) – The state to populate the object with
>
> **Returns** None
>
> **Raises** **`FastrValueError`** – if the parent network and *fastr.current_network* are not set

cardinality(*index=None*)
Cardinality for a Link is given by source Output and the collapse/expand settings

> **Parameters** **key** (`SampleIndex`) – key for a specific sample (can be only a sample index!)
>
> **Returns** the cardinality
>
> **Return type** int, sympy.Symbol
>
> **Raises** **`FastrIndexError`** – if the index length does not match the number of dimension in the data

collapse
The converging dimensions of this link. Collapsing changes some dimensions of sample lists into cardinality, reshaping the data.

Collapse can be set to a tuple or an int/str, in which case it will be automatically wrapped in a tuple. The int will be seen as indices of the dimensions to collapse. The str will be seen as the name of the dimensions over which to collapse.

> **Raises** **`FastrTypeError`** – if assigning a collapse value of a wrong type

collapse_indexes
The converging dimensions of this link as integers. Dimension names are replaces with the corresponding int.

Collapsing changes some dimensions of sample lists into cardinality, reshaping the data

classmethod createobj(*state*, *network=None*)
Create object function for Link

> **Parameters**

- **cls** – The class to create
- **state** – The state to use to create the Link
- **network** – the parent Network

> **Returns** newly created Link

destroy()
> The destroy function of a link removes all default references to a link. This means the references in the network, input and output connected to this link. If there is no references in other places in the code, it will destroy the link (reference count dropping to zero).
>
> This function is called when a source for an input is set to another value and the links becomes disconnected. This makes sure there is no dangling links.

dimensions
> The dimensions of the data delivered by the link. This can be different from the source dimensions because the link can make data collapse or expand.

expand
> Flag indicating that the link will expand the cardininality into a new sample dimension to be created.

fullid
> The full defining ID for the Input

parent
> The Network to which this Link belongs.

size
> The size of the data delivered by the link. This can be different from the source size because the link can make data collapse or expand.

source
> The source *BaseOutput* of the Link. Setting the source will automatically register the Link with the source BaseOutput. Updating source will also make sure the Link is unregistered with the previous source.
>
> > **Raises** *FastrTypeError* – if assigning a non *BaseOutput*

status

target
> The target *BaseInput* of the Link. Setting the target will automatically register the Link with the target BaseInput. Updating target will also make sure the Link is unregistered with the previous target.
>
> > **Raises** *FastrTypeError* – if assigning a non *BaseInput*

macronoderun Module

class fastr.execution.macronoderun.**MacroNodeRun**(*node*, *parent*)
> Bases: *fastr.execution.noderun.NodeRun*

> MacroNodeRun encapsulates an entire network in a single node.

> **__abstractmethods__ = frozenset()**

> **__getstate__**()
> > Retrieve the state of the MacroNodeRun
> >
> > > **Returns** the state of the object
> > >
> > > **Rtype dict**

__init__(*node*, *parent*)

>> Parameters **network** ([Network](#)) – network to create macronode for

__module__ = **'fastr.execution.macronoderun'**

__setstate__(*state*)
> Set the state of the NodeRun by the given state.

>> Parameters **state** ([*dict*](#)) – The state to populate the object with

>> Returns None

execute()
> Execute the node and create the jobs that need to run

>> Returns list of jobs to run

>> Return type list of [*Jobs*](#)

get_output_info(*output*)

network_run

networkanalyzer Module

Module that defines the NetworkAnalyzer and holds the reference implementation.

class fastr.execution.networkanalyzer.**DefaultNetworkAnalyzer**
> Bases: *fastr.execution.networkanalyzer.NetworkAnalyzer*

> Default implementation of the NetworkAnalyzer.

> **__module__** = **'fastr.execution.networkanalyzer'**

> **analyze_network**(*network*, *chunk*)
>> Analyze a chunk of a Network. Simply process the Nodes in the chunk sequentially.

>>> Parameters

>>> • **network** – Network corresponding with the chunk

>>> • **chunk** – The chunk of the network to analyze

class fastr.execution.networkanalyzer.**NetworkAnalyzer**
> Bases: [object](#)

> Base class for NetworkAnalyzers

> **__dict__** = **mappingproxy({'__module__': 'fastr.execution.networkanalyzer', '__doc__':**

> **__module__** = **'fastr.execution.networkanalyzer'**

> **__weakref__**
>> list of weak references to the object (if defined)

> **analyze_network**(*network*, *chunk*)
>> Analyze a chunk of a Network.

>>> Parameters

>>> • **network** – Network corresponding with the chunk

>>> • **chunk** – The chunk of the network to analyze

`networkchunker` Module

This module contains the NetworkChunker class and its default implementation the DefaultNetworkChunker

class `fastr.execution.networkchunker.`**`DefaultNetworkChunker`**
>   Bases: *`fastr.execution.networkchunker.NetworkChunker`*

>   The default implementation of the NetworkChunker. It tries to create as large as possible chunks so the execution blocks as little as possible.

>   **`__init__`**`()`
>   >   Initialize self. See help(type(self)) for accurate signature.

>   **`__module__`** `= 'fastr.execution.networkchunker'`

>   **`chunck_network`**(*network*)
>   >   Create a list of Network chunks that can be pre-analyzed completely. Each chunk needs to be executed before the next can be analyzed and executed.

>   >   The returned chunks are (at the moment) in the format of a tuple (start, nodes) which are both tuples. The tuple contain the nodes where to start execution (should ready if previous chunks are done) and all nodes of the chunk respectively.

>   >   >   **Parameters** **`network`** – Network to split into chunks

>   >   >   **Returns** tuple containing chunks

class `fastr.execution.networkchunker.`**`NetworkChunker`**
>   Bases: `object`

>   The base class for NetworkChunkers. A Network chunker is a class that takes a Network and produces a list of chunks that can each be analyzed and executed in one go.

>   **`__dict__`** `= mappingproxy({'__module__': 'fastr.execution.networkchunker', '__doc__':`

>   **`__module__`** `= 'fastr.execution.networkchunker'`

>   **`__weakref__`**
>   >   list of weak references to the object (if defined)

>   **`chunck_network`**(*network*)
>   >   Create a list of Network chunks that can be pre-analyzed completely. Each chunk needs to be executed before the next can be analyzed and executed.

>   >   >   **Parameters** **`network`** – Network to split into chunks

>   >   >   **Returns** list containing chunks

`networkrun` Module

Network module containing Network facilitators and analysers.

class `fastr.execution.networkrun.`**`NetworkRun`**(*network*)
>   Bases: `fastr.abc.serializable.Serializable`

>   The Network class represents a workflow. This includes all Nodes (including ConstantNodes, SourceNodes and Sinks) and Links.

>   **`NETWORK_DUMP_FILE_NAME`** `= '__fastr_network__.json'`

>   **`SINK_DUMP_FILE_NAME`** `= '__sink_data__.json'`

>   **`SOURCE_DUMP_FILE_NAME`** `= '__source_data__.pickle.gz'`

`__bool__` ()
> A network run is True if it finish running successfully and False otherwise

`__eq__` (*other*)
> Compare two Networks and see if they are equal.

> > **Parameters other** (`Network`) –

> > **Returns** flag indicating that the Networks are the same

> > **Return type** bool

`__getitem__` (*item*)
> Get an item by its fullid. The fullid can point to a link, node, input, output or even subinput/suboutput.

> > **Parameters item** (`str, unicode`) – fullid of the item to retrieve

> > **Returns** the requested item

`__getstate__` ()
> Retrieve the state of the Network

> > **Returns** the state of the object

> > **Rtype dict**

`__hash__` **= None**

`__init__` (*network*)
> Create a new, empty Network

> > **Parameters name** (`str`) – name of the Network

> > **Returns** newly created Network

> > **Raises** `OSError` – if the tmp mount in the config is not a writable directory

`__module__` **= 'fastr.execution.networkrun'**

`__ne__` (*other*)
> Tests for non-equality, this is the negated version __eq__

`__repr__` ()
> Return repr(self).

`__setstate__` (*state*)
> Set the state of the Network by the given state. This completely overwrites the old state!

> > **Parameters state** (`dict`) – The state to populate the object with

> > **Returns** None

abort (*signal_code=None*, *current_frame=None*)

check_id (*id_*)
> Check if an id for an object is valid and unused in the Network. The method will always returns True if it does not raise an exception.

> > **Parameters id** (`str`) – the id to check

> > **Returns** True

> > **Raises**

> > > • **`FastrValueError`** – if the id is not correctly formatted

> > > • **`FastrValueError`** – if the id is already in use

constantlist

execute(*sourcedata*, *sinkdata*, *execution_plugin=None*, *tmpdir=None*, *cluster_queue=None*, *timestamp=None*)

Execute the Network with the given data. This will analyze the Network, create jobs and send them to the execution backend of the system.

> **Parameters**
>
> - **sourcedata** (*dict*) – dictionary containing all data for the sources
> - **sinkdata** (*dict*) – dictionary containing directives for the sinks
> - **execution_plugin** (*str*) – the execution plugin to use (None will use the config value)
>
> **Raises**
>
> - ***FastrKeyError*** – if a source has not corresponding key in sourcedata
> - ***FastrKeyError*** – if a sink has not corresponding key in sinkdata

execution_finished()

fullid

> The fullid of the Network

generate_jobs()

global_id

> The global id of the Network, this is different for networks used in macronodes, as they still have parents.

id

> The id of the Network. This is a read only property.

job_finished(*job*)

Call-back handler for when a job is finished. Will collect the results and handle blocking jobs. This function is automatically called when the execution plugin finished a job.

> **Parameters** **job** (*Job*) – the job that finished

long_id

network

nodegroups

> Give an overview of the nodegroups in the network

register_signals()

set_data(*sourcedata*, *sinkdata*)

sinklist

sourcelist

unregister_signals()

noderun Module

A module to maintain a run of a network node.

class fastr.execution.noderun.**NodeRun**(*node*, *parent*)

> Bases: *fastr.execution.basenoderun.BaseNodeRun*

The class encapsulating a node in the network. The node is responsible for setting and checking inputs and outputs based on the description provided by a tool instance.

__abstractmethods__ = frozenset()

__dataschemafile__ = 'NodeRun.schema.json'

__eq__(*other*)
> Compare two Node instances with each other. This function ignores the parent and update status, but tests rest of the dict for equality. equality

> > **Parameters other** (`NodeRun`) – the other instances to compare to

> > **Returns** True if equal, False otherwise

__getstate__()
> Retrieve the state of the NodeRun

> > **Returns** the state of the object

> > **Rtype dict**

__hash__ = None

__init__(*node*, *parent*)
> Instantiate a node.

> > **Parameters**

> > > • **node** (`Tool`) – The node to base the noderun on

> > > • **parent** (`Network`) – the parent network of the node

> > **Returns** the newly created NodeRun

__module__ = 'fastr.execution.noderun'

__repr__()
> Get a string representation for the NodeRun

> > **Returns** the string representation

> > **Return type** str

__setstate__(*state*)
> Set the state of the NodeRun by the given state.

> > **Parameters state** (`dict`) – The state to populate the object with

> > **Returns** None

__str__()
> Get a string version for the NodeRun

> > **Returns** the string version

> > **Return type** str

blocking
> Indicate that the results of this NodeRun cannot be determined without first executing the NodeRun, causing a blockage in the creation of jobs. A blocking Nodes causes the Chunk borders.

create_job(*sample_id*, *sample_index*, *job_data*, *job_dependencies*, ***kwargs*)
> Create a job based on the sample id, job data and job dependencies.

> > **Parameters**

> > > • **sample_id** (`SampleId`) – the id of the corresponding sample

- **sample_index** (SampleIndex) – the index of the corresponding sample

- **job_data** ([*dict*](#)) – dictionary containing all input data for the job

- **job_dependencies** – other jobs that need to finish before this job can run

> **Returns** the created job

> **Return type** [*Job*](#)

classmethod createobj(*state*, *network=None*)
  Create object function for generic objects

  Parameters

  - **cls** – The class to create

  - **state** – The state to use to create the Link

  - **network** – the parent Network

  Returns newly created Link

dimnames
  Names of the dimensions in the NodeRun output. These will be reflected in the SampleIdList of this NodeRun.

execute()
  Execute the node and create the jobs that need to run

  > **Returns** list of jobs to run

  > **Return type** list of [*Jobs*](#)

find_source_index(*target_index*, *target*, *source*)

fullid
  The full defining ID for the NodeRun inside the network

get_sourced_nodes()
  A list of all Nodes connected as sources to this NodeRun

  > **Returns** list of all nodes that are connected to an input of this node

global_id
  The global defining ID for the Node from the main network (goes out of macro nodes to root network)

id
  The id of the NodeRun

input_groups

  A list of input groups for this NodeRun. An input group is InputGroup object filled according to the
    NodeRun

listeners
  All the listeners requesting output of this node, this means the listeners of all Outputs and SubOutputs

merge_dimensions

name
  Name of the Tool the NodeRun was based on. In case a Toolless NodeRun was used the class name is given.

outputsize
  Size of the outputs in this NodeRun

parent
>   The parent network of this node.

resources
>   Number of cores required for the execution of this NodeRun

set_result (*job*, *failed_annotation*)
>   Incorporate result of a job into the NodeRun.

>   **Parameters**
>
>   - **job** (`Type`) – job of which the result to store
>
>   - **failed_annotation** – A set of annotations, None if no errors else containing a tuple describing the errors

status

tool

update_input_groups ()
>   Update all input groups in this node

sinknoderun Module

class fastr.execution.sinknoderun.**SinkNodeRun** (*node*, *parent*)
>   Bases: *fastr.execution.noderun.NodeRun*

>   Class which handles where the output goes. This can be any kind of file, e.g. image files, textfiles, config files, etc.

>   **__abstractmethods__ = frozenset()**

>   **__dataschemafile__ = 'SinkNodeRun.schema.json'**

>   **__getstate__** ()
>   >   Retrieve the state of the NodeRun

>   >   **Returns** the state of the object

>   >   **Rtype dict**

>   **__init__** (*node*, *parent*)
>   >   Instantiation of the SinkNodeRun.

>   >   **Parameters**
>   >
>   >   - **node** (`Node`) – The Node that this Run is based on.
>   >
>   >   - **parent** (`NetworkRun`) – The NetworkRun that this NodeRun belongs to
>   >
>   >   **Returns** newly created sink node run

>   **__module__ = 'fastr.execution.sinknoderun'**

>   **__setstate__** (*state*)
>   >   Set the state of the NodeRun by the given state.

>   >   **Parameters state** (*dict*) – The state to populate the object with

>   >   **Returns** None

>   **create_job** (*sample_id*, *sample_index*, *job_data*, *job_dependencies*, ***kwargs*)
>   >   Create a job for a sink based on the sample id, job data and job dependencies.

> Parameters

> - **sample_id** (SampleId) – the id of the corresponding sample
> - **job_data** ([*dict*](#)) – dictionary containing all input data for the job
> - **job_dependencies** – other jobs that need to finish before this job can run

> Returns the created job

> Return type [*Job*](#)

datatype
> The datatype of the data this sink can store.

execute()
> Execute the sink node and create the jobs that need to run

> Returns list of jobs to run

> Return type list of [*Jobs*](#)

input
> The default input of the sink NodeRun

set_data(*data*)
> Set the targets of this sink node.

> Parameters **data** ([*dict or list of urls*](#)) – the targets rules for where to write the data

> The target rules can include a few fields that can be filled out:

field	description
sample_id	the sample id of the sample written in string form
cardinality	the cardinality of the sample written
ext	the extension of the datatype of the written data, including the .
extension	the extension of the datatype of the written data, excluding the .
network	the id of the network the sink is part of
node	the id of the node of the sink
timestamp	the iso formatted datetime the network execution started
uuid	the uuid of the network run (generated using uuid.uuid1)

> An example of a valid target could be:

```
>>> target = 'vfs://output_mnt/some/path/image_{sample_id}_{cardinality}{ext}'
```

> Note: The {ext} and {extension} are very similar but are both offered. In many cases having a `name.{extension}` will feel like the correct way to do it. However, if you have DataTypes with and without extension that can both exported by the same sink, this would cause either `name.ext` or `name.` to be generated. In this particular case `name{ext}` can help as it will create either `name.ext` or `name.`

> Note: If a datatype has multiple extensions (e.g. .tiff and .tif) the first extension defined in the extension tuple of the datatype will be used.

set_result(*job*, *failed_annotation*)
> Incorporate result of a sink job into the Network.

Parameters

- **job** (*Type*) – job of which the result to store

- **failed_annotation** (*set*) – A set of annotations, None if no errors else containing a tuple describing the errors

sourcenoderun Module

class fastr.execution.sourcenoderun.**SourceNodeRun**(*node*, *parent*)

Bases: *fastr.execution.flownoderun.FlowNodeRun*

Class providing a connection to data resources. This can be any kind of file, stream, database, etc from which data can be received.

__abstractmethods__ = frozenset()

__dataschemafile__ = 'SourceNodeRun.schema.json'

__eq__(*other*)

Compare two Node instances with each other. This function ignores the parent and update status, but tests rest of the dict for equality. equality

Parameters **other** (*NodeRun*) – the other instances to compare to

Returns True if equal, False otherwise

__getstate__()

Retrieve the state of the SourceNodeRun

Returns the state of the object

Rtype dict

__hash__ = None

__init__(*node*, *parent*)

Instantiation of the SourceNodeRun.

Parameters

- **node** (*Node*) – The Node that this Run is based on.

- **parent** (*NetworkRun*) – The NetworkRun that this NodeRun belongs to

Returns newly created sink node run

__module__ = 'fastr.execution.sourcenoderun'

__setstate__(*state*)

Set the state of the SourceNodeRun by the given state.

Parameters **state** (*dict*) – The state to populate the object with

Returns None

create_job(*sample_id*, *sample_index*, *job_data*, *job_dependencies*, ***kwargs*)

Create a job based on the sample id, job data and job dependencies.

Parameters

- **sample_id** (*SampleId*) – the id of the corresponding sample

- **sample_index** (*SampleIndex*) – the index of the corresponding sample

- **job_data** (*dict*) – dictionary containing all input data for the job

- **job_dependencies** – other jobs that need to finish before this job can run

> **Returns** the created job
>
> **Return type** [*Job*](#)

datatype
> The datatype of the data this source supplies.

dimnames
> Names of the dimensions in the SourceNodeRun output. These will be reflected in the SampleIdLists.

execute()
> Execute the source node and create the jobs that need to run
>
> > **Returns** list of jobs to run
> >
> > **Return type** list of [*Jobs*](#)

output
> Shorthand for `self.outputs['output']`

outputsize
> The size of output of this SourceNodeRun

set_data(*data*, *ids=None*)
> Set the data of this source node.
>
> > **Parameters**
> >
> > - **data** ([*dict, OrderedDict or list of urls*](#)) – the data to use
> >
> > - **ids** – if data is a list, a list of accompanying ids

sourcegroup

valid
> This does nothing. It only overloads the valid method of NodeRun(). The original is intended to check if the inputs are connected to some output. Since this class does not implement inputs, it is skipped.

class fastr.execution.sourcenoderun.**ConstantNodeRun**(*node*, *parent*)
> Bases: [*fastr.execution.sourcenoderun.SourceNodeRun*](#)

Class encapsulating one output for which a value can be set. For example used to set a scalar value to the input of a node.

__abstractmethods__ = frozenset()

__dataschemafile__ = 'ConstantNodeRun.schema.json'

__getstate__()
> Retrieve the state of the ConstantNodeRun
>
> > **Returns** the state of the object
> >
> > **Rtype dict**

__init__(*node*, *parent*)
> Instantiation of the ConstantNodeRun.
>
> > **Parameters**
> >
> > - **datatype** – The datatype of the output.
> >
> > - **data** – the prefilled data to use.
> >
> > - **id** – The url pattern.

This class should never be instantiated directly (unless you know what you are doing). Instead create a constant using the network class like shown in the usage example below.

usage example:

```
>>> import fastr
>>> network = fastr.Network()
>>> source = network.create_source(datatype=types['ITKImageFile'], id_=
↪'sourceN')
```

or alternatively create a constant node by assigning data to an item in an InputDict:

```
>>> node_a.inputs['in'] = ['some', 'data']
```

which automatically creates and links a ConstantNodeRun to the specified Input

__module__ = 'fastr.execution.sourcenoderun'

__setstate__(*state*)
    Set the state of the ConstantNodeRun by the given state.

        Parameters **state** (*dict*) – The state to populate the object with

        Returns None

data
    The data stored in this constant node

execute()
    Execute the constant node and create the jobs that need to run

        Returns list of jobs to run

        Return type list of *Jobs*

set_data(*data=None*, *ids=None*)
    Set the data of this constant node in the correct way. This is mainly for compatibility with the parent class SourceNodeRun

        Parameters

            • **data** (*dict or list of urls*) – the data to use

            • **ids** – if data is a list, a list of accompanying ids

helpers Package

helpers Package

fastr.helpers.**config = # [bool] Flag to enable/disable debugging debug = False # [str] Dire**
    Configuration of the fastr system

checksum Module

This module contains a number of functions for checksumming files and objects

fastr.helpers.checksum.**checksum**(*filepath*, *algorithm='md5'*, *hasher=None*, *chunksize=32768*)
    Generate the checksum of a file

        Parameters

- **filepath** (`str, list`) – path of the file(s) to checksum
- **algorithm** (`str`) – the algorithm to use
- **hasher** (`_hashlib.HASH`) – a hasher to continue updating (rather then creating a new one)

    Returns the checksum

    Return type str

fastr.helpers.checksum.**checksum_directory**(*directory*, *algorithm='md5'*, *hasher=None*)
    Generate the checksum of an entire directory

    Parameters

- **directory** (`str`) – path of the file(s) to checksum
- **algorithm** (`str`) – the algorithm to use
- **hasher** (`_hashlib.HASH`) – a hasher to continue updating (rather then creating a new one)

    Returns the checksum

    Return type str

fastr.helpers.checksum.**hashsum**(*objects*, *hasher=None*)
    Generate the md5 checksum of (a) python object(s)

    Parameters

- **objects** – the objects to hash
- **hasher** – the hasher to use as a base

    Returns the hash generated

    Return type str

fastr.helpers.checksum.**md5_checksum**(*filepath*)
    Generate the md5 checksum of a file

    Parameters filepath (`str, list`) – path of the file(s) to checksum

    Returns the checksum

    Return type str

fastr.helpers.checksum.**sha1_checksum**(*filepath*)
    Generate the sha1 checksum of a file

    Parameters filepath (`str, list`) – path of the file(s) to checksum

    Returns the checksum

    Return type str

classproperty Module

Module containing the code to create class properties.

class fastr.helpers.classproperty.**ClassPropertyDescriptor**(*fget*)
    Bases: object

    A descriptor that can act like a property for a class.

> **__dict__** = mappingproxy({'__module__': 'fastr.helpers.classproperty', '__doc__': '\n

> **__get__**(*obj*, *cls=None*)

> **__init__**(*fget*)
> > Initialize self. See help(type(self)) for accurate signature.

> **__module__** = 'fastr.helpers.classproperty'

> **__weakref__**
> > list of weak references to the object (if defined)

fastr.helpers.classproperty.**classproperty**(*func*)
> Decorator to create a "class property"

> > **Parameters** **func** – the function to wrap

> > **Returns** a class property

> > **Return type** *ClassPropertyDescriptor*

clear_pycs **Module**

A small tool to wipe all .pyc files from fastr

fastr.helpers.clear_pycs.**dir_list**(*directory*)
> Find all .pyc files

> > **Parameters** **directory** (*str*) – directory to search

> > **Returns** all .pyc files

> > **Return type** list

fastr.helpers.clear_pycs.**main**()
> Main entry poitn

configmanager **Module**

This module defines the Fastr Config class for managing the configuration of Fastr. The config object is stored directly in the fastr top-level module.

class fastr.helpers.configmanager.**Config**(**configfiles*)
> Bases: object

> Class contain the fastr configuration

> **DEFAULT_FIELDS** = {'debug': (<class 'bool'>, False, 'Flag to enable/disable debugging'

> **__dict__** = mappingproxy({'__module__': 'fastr.helpers.configmanager', '__doc__': '\n

> **__init__**(**configfiles*)
> > Initialize self. See help(type(self)) for accurate signature.

> **__module__** = 'fastr.helpers.configmanager'

> **__repr__**()
> > Return repr(self).

> **__weakref__**
> > list of weak references to the object (if defined)

> **debug**

examplesdir

execution_plugin

executionscript

filesynchelper_url

get_field(*item*)

job_cleanup_level

logdir

logging_config

loglevel

logtype

mounts

networks_path

pim_batch_size

pim_debug

pim_host

pim_update_interval

plugins_path

preferred_types

process_pool_worker_number

protected_modules

read_config(*filename*)
    Read a configuration and update the configuration object accordingly

        Parameters **filename** – the configuration file to read

read_config_files = None
    Trace of the config files read by this object

read_config_string(*value*)

register_fields(*fields_spec*)
    Register extra fields to the configuration manager.

reporting_plugins

resourcesdir

schemadir

set_field(*item*, *value*)

slurm_job_check_interval

slurm_partition

source_job_limit

systemdir

tools_path

types_path

userdir

warn_develop

web_hostname

web_port

web_secret_key

web_url()

> Construct a fqdn from the web['hostname'] and web['port'] settings. :return: FQDN :rtype: str

class fastr.helpers.configmanager.**EmptyDefault**(*data=None*)

> Bases: [object](#)

Empty defaultdict.

__add__(*right*)

__delitem__(*key*)

__dict__ = mappingproxy({'__module__': 'fastr.helpers.configmanager', '__doc__': ' 1

__getitem__(*item*)

__iadd__(*right*)

__init__(*data=None*)

> Initialize self. See help(type(self)) for accurate signature.

__module__ = 'fastr.helpers.configmanager'

__radd__(*other*)

__setitem__(*key*, *value*)

__weakref__

> list of weak references to the object (if defined)

append(*value*)

asdict()

aslist()

extend(*other*)

merge_default(*field_spec*)

> Merge the default into this EmptyDefault given the field spec :param field_spec: Field specification :return: Merged value

prepend(*value*)

update(*other*)

events Module

class fastr.helpers.events.**EventType**

> Bases: [enum.Enum](#)

An enumeration.

__module__ = 'fastr.helpers.events'

```
job_updated = 'job_updated'

log_record_emitted = 'log_record_emitted'

run_finished = 'run_finished'

run_started = 'run_started'
```

class fastr.helpers.events.**FastrLogEventHandler**(*level=0*)

    Bases: `logging.Handler`

    Logging handler that sends the log records into the event system

    __module__ = **'fastr.helpers.events'**

    emit(*record*)

        Do whatever it takes to actually log the specified logging record.

        This version is intended to be implemented by subclasses and so raises a NotImplementedError.

fastr.helpers.events.**emit_event**(*event_type*, *data*)

    Emit an event to all listeners :type event_type: *EventType* :param event_type: The type of event to emit :param data: The data object to send along

fastr.helpers.events.**register_listener**(*event_type*, *function*)

    Register a listeners to a specific event type

    Parameters

        • **event_type** (*EventType*) – The EventType to listen on

        • **function** (`Callable`[[`object`], None]) – The callable that will be called on each event

fastr.helpers.events.**remove_listener**(*event_type*, *function*)

    Remove a listeren from a type of event

    Parameters

        • **event_type** (*EventType*) – The event type to remove the listeners from

        • **function** (`Callable`[[`object`], None]) – The function to remove

filesynchelper Module

Some helper functions that aid with NFS file sync issues.

class fastr.helpers.filesynchelper.**FileSyncHelper**

    Bases: `object`

    __dict__ = mappingproxy({'__module__': 'fastr.helpers.filesynchelper', '_namespace':

    __init__()

        Initialize self. See help(type(self)) for accurate signature.

    __module__ = **'fastr.helpers.filesynchelper'**

    __weakref__

        list of weak references to the object (if defined)

    has_file_promise(*url*)

    job_finished(*jobfile*)

    load(*url*)

> **make_file_promise**(*url*)
>
> **store**(*url*, *data*)
>
> **wait_for_file**(*path*, *timeout=300*)
>
> **wait_for_job**(*jobfile*)
>
> **wait_for_pickle**(*url*, *timeout=300*)
>
> **wait_for_vfs_url**(*vfs_url*, *timeout=300*)

fastr.helpers.filesynchelper.**filesynchelper_enabled**()

`iohelpers` Module

fastr.helpers.iohelpers.**link_or_copy**(*source*, *destination*)

fastr.helpers.iohelpers.**load_gpickle**(*path*, *retry_scheme=None*)

fastr.helpers.iohelpers.**load_json**(*path*)

fastr.helpers.iohelpers.**save_gpickle**(*path*, *data*)

fastr.helpers.iohelpers.**save_json**(*path*, *data*, *indent=2*)

`jsonschemaparser` Module

The JSON schema parser validates a json data structure and if possible casts data to the correct type and fills out default values. The result in a valid document that can be used to construct objects.

class fastr.helpers.jsonschemaparser.**FastrRefResolver**(*base_uri*, *referrer*, *store=()*, *cache_remote=True*, *handlers=()*)

> Bases: `jsonschema.validators.RefResolver`
>
> Adapted version of the RefResolver for handling inter-file references more to our liking
>
> **__init__**(*base_uri*, *referrer*, *store=()*, *cache_remote=True*, *handlers=()*)
> > Create a new FastrRefResolver
> >
> > > **Parameters**
> > >
> > > - **base_uri** (*str*) – URI of the referring document
> > > - **referrer** – the actual referring document
> > > - **store** (*dict*) – a mapping from URIs to documents to cache
> > > - **cache_remote** (*bool*) – whether remote refs should be cached after first resolution
> > > - **handlers** (*dict*) – a mapping from URI schemes to functions that should be used to retrieve them
>
> **__module__** = 'fastr.helpers.jsonschemaparser'
>
> **classmethod from_schema**(*schema*, **args*, ***kwargs*)
> > Instantiate a RefResolver based on a schema
>
> **static readfastrschema**(*name*)
> > Open a json file based on a fastr:// url that points to a file in the fastr.schemadir
> >
> > > **Parameters name** (*str*) – the url of the file to open

---

> **Returns** the resulting json schema data

static readfile(*filename*)
> Open a json file based on a simple filename

> > **Parameters filename** (`str`) – the path of the file to read

> > **Returns** the resulting json schema data

`fastr.helpers.jsonschemaparser.`**any_of_draft4**(*validator*, *any_of*, *instance*, *schema*)
> The oneOf directory needs to be done stepwise, because a validation even if it fails will try to change types / set defaults etc. Therefore we first create a copy of the data per subschema and test if they match. Then for all the schemas that are valid, we perform the validation on the actual data so that only the valid subschemas will effect the data.

> > **Parameters**

> > > • **validator** – the json schema validator

> > > • **any_of** (`dict`) – the current oneOf

> > > • **instance** – the current object instance

> > > • **schema** (`dict`) – the current json schema

`fastr.helpers.jsonschemaparser.`**extend**(*validator_cls*)
> Extend the given `jsonschema.IValidator` with the Seep layer.

`fastr.helpers.jsonschemaparser.`**getblueprinter**(*uri*, *blueprint=None*)
> Instantiate the given data using the blueprinter.

> > **Parameters blueprint** – a blueprint (JSON Schema with Seep properties)

`fastr.helpers.jsonschemaparser.`**items_prevalidate**(*validator*, *items*, *instance*, *schema*)
> The pre-validation function for items

> > **Parameters**

> > > • **validator** – the json schema validator

> > > • **items** (`dict`) – the current items

> > > • **instance** – the current object instance

> > > • **schema** (`dict`) – the current json schema

`fastr.helpers.jsonschemaparser.`**not_draft4**(*validator*, *not_schema*, *instance*, *schema*)
> The not needs to use a temporary copy of the instance, not to change the instance with the invalid schema

> > **Parameters**

> > > • **validator** – the json schema validator

> > > • **not_schema** (`dict`) – the current oneOf

> > > • **instance** – the current object instance

> > > • **schema** (`dict`) – the current json schema

`fastr.helpers.jsonschemaparser.`**one_of_draft4**(*validator*, *one_of*, *instance*, *schema*)
> The one_of directory needs to be done stepwise, because a validation even if it fails will try to change types / set defaults etc. Therefore we first create a copy of the data per subschema and test if they match. Once we found a proper match, we only validate that branch on the real data so that only the valid piece of schema will effect the data.

> > **Parameters**

- **validator** – the json schema validator

- **one_of** (*dict*) – the current one_of

- **instance** – the current object instance

- **schema** (*dict*) – the current json schema

fastr.helpers.jsonschemaparser.**pattern_properties_prevalid**(*validator,* *pat-*
tern_properties,
instance, schema)

> The pre-validation function for patternProperties

> > **Parameters**

> > > - **validator** – the json schema validator

> > > - **pattern_properties** (*dict*) – the current patternProperties

> > > - **instance** (*dict*) – the current object instance

> > > - **schema** (*dict*) – the current json schema

fastr.helpers.jsonschemaparser.**properties_postvalidate**(*validator,* *properties,* *in-*
stance, schema)

> # All arguments must be used because this function is called like this # pylint: disable=unused-argument The
> post-validation function for properties

> > **Parameters**

> > > - **validator** – the json schema validator

> > > - **properties** (*dict*) – the current properties

> > > - **instance** – the current object instance

> > > - **schema** (*dict*) – the current json schema

fastr.helpers.jsonschemaparser.**properties_prevalidate**(*validator,* *properties,* *in-*
stance, schema)

> The pre-validation function for properties

> > **Parameters**

> > > - **validator** – the json schema validator

> > > - **properties** (*dict*) – the current properties

> > > - **instance** – the current object instance

> > > - **schema** (*dict*) – the current json schema

lazy_module Module

This module contains the Manager class for Plugins in the fastr system

class fastr.helpers.lazy_module.**LazyModule**(*name, parent, plugin_manager*)

> Bases: module

> A module that allows content to be loaded lazily from plugins. It generally is (almost) empty and gets (partially)
> populated when an attribute cannot be found. This allows lazy loading and plugins depending on other plugins.

> **__getattr__**(*item*)

> > The getattr is called when getattribute does not return a value and is used as a fallback. In this case we try
> > to find the value normally and will trigger the plugin manager if it cannot be found.

> Parameters **item** (*str*) – attribute to retrieve
>
> Returns the requested attribute

__init__(*name*, *parent*, *plugin_manager*)
> Initialize self. See help(type(self)) for accurate signature.

__module__ = **'fastr.helpers.lazy_module'**

__repr__()
> Return repr(self).

`lockfile` Module

A module implenting a lock that ensures a directory is only being used by a single fastr run.

class fastr.helpers.lockfile.**DirectoryLock**(*directory*)
> Bases: `object`
>
> A lock for a directory, it creates a directory to set the locked state and if successful writes the pid in a file inside that directory to claim the lock
>
> **__del__**()
>
> **__dict__** = **mappingproxy({'__module__': 'fastr.helpers.lockfile', '__doc__': '\n A lo**
>
> **__enter__**()
>
> **__exit__**(*type*, *value*, *traceback*)
>
> **__init__**(*directory*)
> > Initialize self. See help(type(self)) for accurate signature.
>
> **__module__** = **'fastr.helpers.lockfile'**
>
> **__weakref__**
> > list of weak references to the object (if defined)
>
> **acquire**()
>
> **lock_dir**
>
> **lock_dir_name** = **'.fastr.lock'**
>
> **pid_file**
>
> **pid_file_name** = **'pid'**
>
> **release**()

`procutils` Module

fastr.helpers.procutils.**which**(*name*)

> **Find executable by name on the PATH, returns the executable that will be** found in case it is used for a
> > Popen call

`rest_generation` Module

fastr.helpers.rest_generation.**create_rest_table**(*data*, *headers*)

> Create a ReST table from data. The data should be a list of columns and the headers should be a list of column names.
>
> > **Parameters**
> >
> > > - **data** (*list*) – List of lists/tuples representing the columns
> > >
> > > - **headers** (*list*) – List of strings for the column names
> >
> > **Returns** a string representing the table in ReST
> >
> > **Return type** str

`schematotable` Module

A module to generate reStructuredText tables from json schema files

class fastr.helpers.schematotable.**SchemaPrinter**(*schema*, *skipfirst=False*)

> Bases: `object`
>
> Object that create a table in reStructuedText from a json schema
>
> **__dict__** = mappingproxy({'__module__': 'fastr.helpers.schematotable', '__doc__': '\n
>
> **__init__**(*schema*, *skipfirst=False*)
>
> > Create the printer object
> >
> > > **Parameters**
> > >
> > > > - **schema** (*dict*) – the json schema to print
> > > >
> > > > - **skipfirst** (*bool*) – flag to indicate that the first line should not be printed
>
> **__module__** = 'fastr.helpers.schematotable'
>
> **__str__**()
>
> > String representation of json schema (that is the printed table)
>
> **__weakref__**
>
> > list of weak references to the object (if defined)
>
> **descend**(*properties*)
>
> > Descend into a subschema
> >
> > > **Parameters properties** (*dict*) – the properties in the subschema
>
> **parse**(*schema=None*)
>
> > Parse a schema
> >
> > > **Parameters schema** (*dict*) – the schema to parse
>
> **printlines**()
>
> > Given a parsed schema (parsing happens when the object is constructed), print all the lines
> >
> > > **Returns** the printed table
> > >
> > > **Return type** str

`sysinfo` Module

This module contains function to help gather system information use for the provenance of the Job execution.

`fastr.helpers.sysinfo.`**`get_cpu_usage`**`()`
>   Get the current CPU usage

> > **Returns** CPU usage info

> > **Return type** [dict]

`fastr.helpers.sysinfo.`**`get_drmaa_info`**`()`
>   Get information about the SGE cluster (if applicable)

> > **Returns** cluster info

> > **Return type** [dict]

`fastr.helpers.sysinfo.`**`get_hostinfo`**`()`
>   Get all information about the current host machine

> > **Returns** host info

> > **Return type** [dict]

`fastr.helpers.sysinfo.`**`get_memory_usage`**`()`
>   Get the current memory usage

> > **Returns** memory usage info

> > **Return type** [dict]

`fastr.helpers.sysinfo.`**`get_mounts`**`()`
>   Get the current mounts known on the system

> > **Returns** mount info

> > **Return type** [dict]

`fastr.helpers.sysinfo.`**`get_os`**`()`
>   Get information about the OS

> > **Returns** OS information

> > **Return type** [dict]

`fastr.helpers.sysinfo.`**`get_processes`**`()`
>   Get a list of all currently running processes

> > **Returns** process information

> > **Return type** [list]

`fastr.helpers.sysinfo.`**`get_python`**`()`
>   Get information about the currently used Python implementation

> > **Returns** python info

> > **Return type** [dict]

`fastr.helpers.sysinfo.`**`get_sysinfo`**`()`
>   Get system information (cpu, memory, mounts and users)

> > **Returns** system information

> > **Return type** [dict]

`fastr.helpers.sysinfo.`**`get_users`**`()`
>   Get current users on the system

>>   **Returns** user info

>>   **Return type** dict

`fastr.helpers.sysinfo.`**`namedtuple_to_dict`**`(`*ntuple*`)`
>   Helper function to convert a named tuple into a dict

>>   **Parameters ntuple** (`namedtuple`) – the namedtuple to convert

>>   **Returns** named tuple as a dict

>>   **Return type** dict

`xmltodict` Module

This module contains tool for converting python dictionaries into XML object and vice-versa.

`fastr.helpers.xmltodict.`**`dump`**`(`*data*, *filehandle*`)`
>   Write a dict to an XML file

>>   **Parameters**

>>   • **data** – data to write

>>   • **filehandle** – file handle to write to

`fastr.helpers.xmltodict.`**`dumps`**`(`*data*`)`
>   Write a dict to an XML string

>>   **Parameters data** – data to write

>>   **Returns** the XML data

>>   **Return type** str

`fastr.helpers.xmltodict.`**`load`**`(`*filehandle*`)`
>   Load an xml file and parse it to a dict

>>   **Parameters filehandle** – file handle to load

>>   **Returns** the parsed data

`fastr.helpers.xmltodict.`**`loads`**`(`*data*`)`
>   Load an xml string and parse it to a dict

>>   **Parameters data** (`str`) – the xml data to load

>>   **Returns** the parsed data

planning Package

`planning` Package

`inputgroup` Module

class `fastr.planning.inputgroup.`**`InputGroup`**`(`**args*, ***kwargs*`)`
>   Bases: `collections.OrderedDict`, `fastr.core.dimension.HasDimensions`

>   A class representing a group of inputs. Input groups allow the

__abstractmethods__ = frozenset()

__getitem__(*key*)
 x.__getitem__(y) <==> x[y]

__init__(**args*, ***kwargs*)
 Create a new InputGroup representation

> **Parameters**
>> • **parent** (NodeRun) – the parent node
>>
>> • **id** ([str](#)) – the id of the input group
>
> **Raises** [*FastrTypeError*](#) – if parent is not a NodeRun

---

Note: This is a wrapped version of fastr.planning.inputgroup.__init__ which triggers an update of the object after being called

---

__module__ = 'fastr.planning.inputgroup'

__setitem__(**args*, ***kwargs*)
 Assign an input to this input group.

> **Parameters**
>> • **key** ([str](#)) – id of the input
>>
>> • **value** ([Input](#)) – the input to assign
>
> **Raises** [*FastrTypeError*](#) – if value of valid type

Note: This is a wrapped version of fastr.planning.inputgroup.__setitem__ which triggers an update of the object after being called

---

__updatefunc__()
 Update the InputGroup. Triggers when a change is made to the content of the InputGroup. Automatically recalculates the size, primary Input etc.

__updatetriggers__ = ['__init__', '__setitem__', '__delitem__', 'clear', 'pop', 'popit

dimensions
 The dimensions of this InputGroup

empty
 Bool indicating that this InputGroup is empty (has no data connected)

find_source_index(*target_size*, *target_dimnames*, *source_size*, *source_dimnames*, *target_index*)

fullid

iterinputvalues
 Iterate over the item in this InputGroup

> **Returns** iterator yielding SampleItems

parent
 The parent node of this InputGroup

primary
 The primary Input in this InputGroup. The primary Input is the Input that defines the size of this Input-Group. In case of ties it will be the first in the tool definition.

> **classmethod solve_broadcast**(*target_size*, *target_dimnames*, *source_size*, *source_dimnames*, *target_index*, *nodegroups=None*)

inputgroupcombiner Module

class fastr.planning.inputgroupcombiner.**BaseInputGroupCombiner**(*parent*)

> Bases: *fastr.core.dimension.HasDimensions*

> An object that takes the different input groups and combines them in the correct way.

> **__abstractmethods__ = frozenset({'iter_input_groups', 'merge', 'unmerge'})**

> **__init__**(*parent*)
>> Initialize self. See help(type(self)) for accurate signature.

> **__iter__**()

> **__module__ = 'fastr.planning.inputgroupcombiner'**

> **dimensions**
>> The dimensions has to be implemented by any subclass. It has to provide a tuple of Dimensions.

>>> **Returns** dimensions

>>> **Return type** tuple

> **fullid**
>> The full id of the InputGroupCombiner

> **input_groups**

> **iter_input_groups**()
>> Iterate over all the merged samples :return:

> **merge**(*list_of_items*)
>> Given a list of items for each input group, it returns the combined list of items.

>>> **Parameters list_of_items** (*list*) – items to combine

>>> **Returns** combined list

> **merge_failed_annotations**(*list_of_failed_annotations*)

> **merge_payloads**(*sample_payloads*)

> **merge_sample_data**(*list_of_sample_data*)

> **merge_sample_id**(*list_of_sample_ids*)

> **merge_sample_index**(*list_of_sample_indexes*)

> **merge_sample_jobs**(*list_of_sample_jobs*)

> **unmerge**(*item*)
>> Given a item it will recreate the seperate items, basically this is the inverse operation of merge. However, this create an OrderedDict so that specific input groups can be easily retrieved. To get a round trip, the values of the OrderedDict should be taken:

```
>>> odict_of_items = combiner.unmerge(item)
>>> item = combiner.merge(odict_of_items.values())
```

>>> **Parameters item** (*list*) – the item to unmerge

>>> **Returns** items

> **Return type** OrderedDict

update()

class fastr.planning.inputgroupcombiner.**DefaultInputGroupCombiner**(*parent*)

> Bases: *fastr.planning.inputgroupcombiner.BaseInputGroupCombiner*

The default input group combiner combines the input group in a cross product version, taking each combinations of samples between the input groups. So if there are two input groups with one with size N and the other with size M x P the result would be N x M x P samples, with all possible combinations of the samples in each input group.

__abstractmethods__ = frozenset()

__module__ = 'fastr.planning.inputgroupcombiner'

iter_input_groups()

> Iterate over all the merged samples :return:

merge(*list_of_items*)

> Given a list of items for each input group, it returns the combined list of items.
>
> > **Parameters** **list_of_items** (*list*) – items to combine
> >
> > **Returns** combined list

unmerge(*item*)

> Given a item it will recreate the seperate items, basically this is the inverse operation of merge. However, this create an OrderedDict so that specific input groups can be easily retrieved. To get a round trip, the values of the OrderedDict should be taken:

```
>>> odict_of_items = combiner.unmerge(item)
>>> item = combiner.merge(odict_of_items.values())
```

> > **Parameters** **item** (*list*) – the item to unmerge
> >
> > **Returns** items
> >
> > **Return type** OrderedDict

class fastr.planning.inputgroupcombiner.**MergingInputGroupCombiner**(*input_groups,*
merge_dimension)

> Bases: *fastr.planning.inputgroupcombiner.BaseInputGroupCombiner*

The merging input group combiner takes a similar approach as the default combiner but merges dimensions that are the same. If input group A has N(3) x M(2) samples and B has M(2) x P(4) it wil not result in N(3) x M(2) x M(2) x P(4), but merge the dimensions M leading to N(3) x M(2) x P(4) in resulting size.

__abstractmethods__ = frozenset()

__init__(*input_groups, merge_dimension*)

> Initialize self. See help(type(self)) for accurate signature.

__module__ = 'fastr.planning.inputgroupcombiner'

iter_input_groups()

> Iterate over all the merged samples :return:

merge(*list_of_items*)

> Given a list of items for each input group, it returns the combined list of items.
>
> > **Parameters** **list_of_items** (*list*) – items to combine

> **Returns** combined list

unmerge(*item*)

> Given a item it will recreate the seperate items, basically this is the inverse operation of merge. However, this create an OrderedDict so that specific input groups can be easily retrieved. To get a round trip, the values of the OrderedDict should be taken:

```
>>> odict_of_items = combiner.unmerge(item)
>>> item = combiner.merge(odict_of_items.values())
```

> **Parameters** **item** (*list*) – the item to unmerge
>
> **Returns** items
>
> **Return type** OrderedDict

update()

`inputoutput` Module

Classes for arranging the input and output for nodes.

Exported classes:

Input – An input for a node (holding datatype). Output – The output of a node (holding datatype and value). ConstantOutput – The output of a node (holding datatype and value).

> **Warning:** Don't mess with the Link, Input and Output internals from other places. There will be a huge chances of breaking the network functionality!

class `fastr.planning.inputoutput.`**`AdvancedFlowOutput`**(*node*, *description*)

> Bases: *fastr.planning.inputoutput.Output*

Output for nodes that have an advanced flow. This means that the output sample id and index is not the same as the input sample id and index. The AdvancedFlowOutput has one extra dimensions that is created by the Node.

`__abstractmethods__` `= frozenset()`

`__module__` `= 'fastr.planning.inputoutput'`

dimensions

> The list of the dimensions in this Output. This will be a tuple of Dimension.

class `fastr.planning.inputoutput.`**`BaseInput`**(*node*, *description*)

> Bases: *fastr.planning.inputoutput.BaseInputOutput*

Base class for all inputs.

`__abstractmethods__` `= frozenset({'dimensions', '_update', 'itersubinputs', 'fullid'})`

`__init__`(*node*, *description*)

> Instantiate a BaseInput
>
> **Parameters**
>
> - **node** – the parent node the input/output belongs to.
>
> - **description** – the `ParameterDescription` describing the input/output.
>
> **Returns** the created BaseInput

Raises

- *[**FastrTypeError**](...)* – if description is not of class `ParameterDescription`

- *[**FastrDataTypeNotAvailableError**](...)* – if the DataType requested cannot be found in the `types`

__lshift__(*other*)

__module__ = **'fastr.planning.inputoutput'**

__rrshift__(*other*)

check_cardinality(*key=None*, *planning=False*)
    Check if the actual cardinality matches the cardinality specified in the ParameterDescription. Optionally you can use a key to test for a specific sample.

> **Parameters key** – sample_index (tuple of int) or `SampleId` for desired sample

> **Returns** flag indicating that the cardinality is correct

> **Return type** [bool](...)

> **Raises** *[**FastrCardinalityError**](...)* – if the Input/Output has an incorrect cardinality description.

create_link_from(*value*)

description_type
    alias of *[fastr.core.interface.InputSpec](...)*

itersubinputs()
    Iterator over the SubInputs

> **Returns** iterator

    example:

```
>>> for subinput in input_a.itersubinputs():
        print subinput
```

class fastr.planning.inputoutput.**BaseInputOutput**(*node*, *description*)
    Bases: *[fastr.core.dimension.HasDimensions](...)*, fastr.abc.updateable.Updateable, fastr.abc.serializable.Serializable

    Base class for Input and Output classes. It mainly implements the properties to access the data from the underlying ParameterDescription.

__abstractmethods__ = **frozenset({'dimensions', '_update', 'fullid'})**

__getstate__()
    Retrieve the state of the BaseInputOutput

> **Returns** the state of the object

> **Rtype dict**

__init__(*node*, *description*)
    Instantiate a BaseInputOutput

> **Parameters**

> - **node** – the parent node the input/output belongs to.

> - **description** – the `ParameterDescription` describing the input/output.

> **Returns** created BaseInputOutput

Raises

- *FastrTypeError* – if description is not of class `ParameterDescription`

- *FastrDataTypeNotAvailableError* – if the DataType requested cannot be found in the `types`

__iter__()
    This function is blocked to avoid support for iteration using a lecacy __getitem__ method.

    Returns None

    Raises *FastrNotImplementedError* – always

__module__ = 'fastr.planning.inputoutput'

__ne__(*other*)
    Check two Node instances for inequality. This is the inverse of __eq__

    Parameters other (`BaseInputOutput`) – the other instances to compare to

    Returns True if unequal, False otherwise

__repr__()
    Get a string representation for the Input/Output

    Returns the string representation

    Return type str

__setstate__(*state*)
    Set the state of the BaseInputOutput by the given state.

    Parameters state (*dict*) – The state to populate the object with

    Returns None

cardinality(*key=None*, *job_data=None*)
    Determine the cardinality of this Input/Output. Optionally a key can be given to determine for a sample.

    Parameters key – key for a specific sample

    Returns the cardinality

    Return type int, sympy.Symbol, or None

check_cardinality(*key=None*)
    Check if the actual cardinality matches the cardinality specified in the ParameterDescription. Optionally you can use a key to test for a specific sample.

    Parameters key – sample_index (tuple of int) or `SampleId` for desired sample

    Returns flag indicating that the cardinality is correct

    Return type bool

    Raises *FastrCardinalityError* – if the Input/Output has an incorrect cardinality description.

datatype
    The datatype of this Input/Output

description
    The description object of this input/output

description_type = None

fullid
    The fullid of the Input/Output, the fullid should be unnique and makes the object retrievable by the network.

id
    Id of the Input/Output

node
    The NodeRun to which this Input/Output belongs

required
    Flag indicating that the Input/Output is required

class fastr.planning.inputoutput.**BaseOutput**(*node*, *description*)
    Bases: *fastr.planning.inputoutput.BaseInputOutput*

    Base class for all outputs.

    __abstractmethods__ = frozenset({'dimensions', '_update', 'fullid'})

    __init__(*node*, *description*)
        Instantiate a BaseOutput

        Parameters

            • **node** – the parent node the output belongs to.

            • **description** – the ParameterDescription describing the output.

        Returns created BaseOutput

        Raises

            • *FastrTypeError* – if description is not of class ParameterDescription

            • *FastrDataTypeNotAvailableError* – if the DataType requested cannot be found in the types

    __module__ = 'fastr.planning.inputoutput'

    automatic
        Flag indicating that the Output is generated automatically without being specified on the command line

    blocking
        Flag indicating that this Output will cause blocking in the execution

    description_type
        alias of *fastr.core.interface.OutputSpec*

class fastr.planning.inputoutput.**Input**(*node*, *description*)
    Bases: *fastr.planning.inputoutput.BaseInput*

    Class representing an input of a node. Such an input will be connected to the output of another node or the output of an constant node to provide the input value.

    __abstractmethods__ = frozenset()

    __eq__(*other*)
        Compare two Input instances with each other. This function ignores the parent node and update status, but tests rest of the dict for equality.

        Parameters other (*Input*) – the other instances to compare to

        Returns True if equal, False otherwise

        Return type bool

__getitem__ (*key*)

Retrieve an item from this Input.

> **Parameters key** (`Union[int, str]`) – the key of the requested item
>
> **Return type** *SubInput*
>
> **Returns** The *SubInput* corresponding with the key will be returned.
>
> **Raises**
>
> - **FastrTypeError** – if key is not of a valid type
>
> - **FastrKeyError** – if the key is not found

__getstate__ ()

Retrieve the state of the Input

> **Returns** the state of the object
>
> **Rtype dict**

__hash__ = None

__init__ (*node*, *description*)

Instantiate an input.

> **Parameters**
>
> - **node** (`NodeRun`) – the parent node of this input.
>
> - **description** (`ParameterDescription`) – the ParameterDescription of the input.
>
> **Returns** the created Input

__module__ = 'fastr.planning.inputoutput'

__setitem__ (*key*, *value*)

Create a link between a SubInput of this Inputs and an Output/Constant

> **Parameters**
>
> - **key** (*int*, *str*) – the key of the SubInput
>
> - **value** (`BaseOutput`, *list*, *tuple*, *dict*, *OrderedDict*) – the target to link, can be an output or a value to create a constant for
>
> **Raises** **FastrTypeError** – if key is not of a valid type

__setstate__ (*state*)

Set the state of the Input by the given state.

> **Parameters state** (*dict*) – The state to populate the object with
>
> **Returns** None

__str__ ()

Get a string version for the Input

> **Returns** the string version
>
> **Return type** str

append (*value*)

When you want to append a link to an Input, you can use the append property. This will automatically create a new SubInput to link to.

example:

```
>>> link = node2['input'].append(node1['output'])
```

will create a new SubInput in node2['input'] and link to that.

cardinality(*key=None*, *job_data=None*)
    Cardinality for an Input is the sum the cardinalities of the SubInputs, unless defined otherwise.

> **Parameters key** (tuple of int or `SampleId`) – key for a specific sample, can be sample index or id

> **Returns** the cardinality

> **Return type** int, sympy.Symbol, or None

clear()

datatype
    The datatype of this Input

dimensions
    The list names of the dimensions in this Input. This will be a list of str.

fullid
    The full defining ID for the Input

> **Return type** str

get_sourced_nodes()
    Get a list of all *Nodes* connected as sources to this Input

> **Returns** list of all connected *Nodes*

> **Return type** list

get_sourced_outputs()
    Get a list of all *Outputs* connected as sources to this Input

> **Returns** tuple of all connected *Outputs*

> **Return type** tuple

index(*value*)
    Find index of a SubInput

> **Parameters value** (*SubInput*) – the *SubInput* to find the index of

> **Returns** key

> **Return type** int, str

input_group
    The id of the `InputGroup` this Input belongs to.

> **Return type** str

insert(*index*)
    Insert a new SubInput at index in the sources list

> **Parameters key** (*int*) – positive integer for position in _source list to insert to

> **Returns** newly inserted *SubInput*

> **Return type** *SubInput*

itersubinputs()
    Iterate over the *SubInputs* in this Input.

Returns iterator yielding *SubInput*

example:

```
>>> for subinput in input_a.itersubinputs():
        print subinput
```

remove(*value*)
    Remove a SubInput from the SubInputs list based on the connected Link.

> **Parameters value** (*SubInput*, <fastr.planning.inputoutput.SubInput>') – the *SubInput* or
> *SubLink* to removed from this Input

source
    The mapping of *SubInputs* that are connected and have more than 0 elements.

class fastr.planning.inputoutput.**MacroInput**(*node*, *description*)
    Bases: *fastr.planning.inputoutput.Input*

__abstractmethods__ = frozenset()

__module__ = 'fastr.planning.inputoutput'

input_group
    The id of the InputGroup this Input belongs to.

class fastr.planning.inputoutput.**MacroOutput**(*node*, *description*)
    Bases: *fastr.planning.inputoutput.Output*

__abstractmethods__ = frozenset()

__module__ = 'fastr.planning.inputoutput'

dimensions
    The list of the dimensions in this Output. This will be a tuple of Dimension.

class fastr.planning.inputoutput.**Output**(*node*, *description*)
    Bases: *fastr.planning.inputoutput.BaseOutput*

Class representing an output of a node. It holds the output values of the tool ran. Output fields can be connected
to inputs of other nodes.

__abstractmethods__ = frozenset()

__eq__(*other*)
    Compare two Output instances with each other. This function ignores the parent node, listeners and update
    status, but tests rest of the dict for equality.

> **Parameters other** (*Output*) – the other instances to compare to

> **Returns** True if equal, False otherwise

> **Return type** bool

__getitem__(*key*)
    Retrieve an item from this Output. The returned value depends on what type of key used:

- Retrieving data using index tuple: [index_tuple]

- Retrieving data sample_id str: [SampleId]

- Retrieving a list of data using SampleId list: [sample_id1, ..., sample_idN]

- Retrieving a *SubOutput* using an int or slice: [n] or [n:m]

> > Parameters **key** (`Union`[`int`, `slice`]) – the key of the requested suboutput, can be a number or slice
>
> > Return type *`SubOutput`*
>
> > Returns the *`SubOutput`* for the corresponding index
>
> > Raises *`FastrTypeError`* – if key is not of a valid type

__getstate__()
> Retrieve the state of the Output

> > Returns the state of the object

> > Rtype dict

__hash__ = None

__init__(*node*, *description*)
> Instantiate an Output

> > Parameters

> > > • **node** – the parent node the output belongs to.

> > > • **description** – the `ParameterDescription` describing the output.

> > Returns created Output

> > Raises

> > > • *`FastrTypeError`* – if description is not of class `ParameterDescription`

> > > • *`FastrDataTypeNotAvailableError`* – if the DataType requested cannot be found in the `types`

__module__ = 'fastr.planning.inputoutput'

__setstate__(*state*)
> Set the state of the Output by the given state.

> > Parameters **state** (*`dict`*) – The state to populate the object with

> > Returns None

__str__()
> Get a string version for the Output

> > Returns the string version

> > Return type str

cardinality()
> Cardinality of this Output, may depend on the inputs of the parent Node.

> > Returns the cardinality

> > Return type int, sympy.Symbol, or None

> > Raises

> > > • *`FastrCardinalityError`* – if cardinality references an invalid *`Input`*

> > > • *`FastrTypeError`* – if the referenced cardinality values type cannot be case to int

> > > • *`FastrValueError`* – if the referenced cardinality value cannot be case to int

datatype
   The datatype of this Output

dimensions
   The list of the dimensions in this Output. This will be a tuple of Dimension.

fullid
   The full defining ID for the Output

listeners
   The list of *Links* connected to this Output.

preferred_types
   The list of preferred `DataTypes` for this Output.

resulting_datatype
   The `DataType` that will the results of this Output will have.

valid
   Check if the output is valid, i.e. has a valid cardinality

class fastr.planning.inputoutput.**SourceOutput**(*node*, *description*)
   Bases: *fastr.planning.inputoutput.Output*

   Output for a SourceNodeRun, this type of Output determines the cardinality in a different way than a normal
   NodeRun.

   __abstractmethods__ = frozenset()

   __init__(*node*, *description*)
      Instantiate a FlowOutput

         Parameters

            • **node** – the parent node the output belongs to.

            • **description** – the `ParameterDescription` describing the output.

         Returns created FlowOutput

         Raises

            • *FastrTypeError* – if description is not of class `ParameterDescription`

            • *FastrDataTypeNotAvailableError* – if the DataType requested cannot be found
              in the `types`

   __module__ = 'fastr.planning.inputoutput'

   cardinality()
      Cardinality of this SourceOutput, may depend on the inputs of the parent NodeRun.

         Parameters key (tuple of int or `SampleId`) – key for a specific sample, can be sample index
            or id

         Returns the cardinality

         Return type int, sympy.Symbol, or None

   linearized
      A linearized version of the sample data, this is lazily cached linearized version of the underlying Sam-
      pleCollection.

class fastr.planning.inputoutput.**SubInput**(*input_*)
   Bases: *fastr.planning.inputoutput.BaseInput*

This class is used by *Input* to allow for multiple links to an *Input*. The SubInput class can hold only a single Link to a (Sub)Output, but behaves very similar to an *Input* otherwise.

__abstractmethods__ = frozenset()

__eq__(*other*)
:   Compare two SubInput instances with each other. This function ignores the parent, node, source and update status, but tests rest of the dict for equality.

    > **Parameters other** (*SubInput*) – the other instances to compare to

    > **Returns** True if equal, False otherwise

__getitem__(*key*)
:   Retrieve an item from this SubInput.

    > **Parameters key** (*int*) – the index of the requested item

    > **Returns** the corresponding *SubInput*

    > **Return type** *SubInput*

    > **Raises** *FastrTypeError* – if key is not of a valid type

    ---

    Note: As a SubInput has only one SubInput, only requesting int key 0 or -1 is allowed, and it will return self

    ---

__getstate__()
:   Retrieve the state of the SubInput

    > **Returns** the state of the object

    > **Rtype dict**

__hash__ = None

__init__(*input_*)
:   Instantiate an SubInput.

    > **Parameters input** (*Input*) – the parent of this SubInput.

    > **Returns** the created SubInput

__module__ = 'fastr.planning.inputoutput'

__setstate__(*state*)
:   Set the state of the SubInput by the given state.

    > **Parameters state** (*dict*) – The state to populate the object with

    > **Returns** None

__str__()
:   Get a string version for the SubInput

    > **Returns** the string version

    > **Return type** str

cardinality(*key=None*, *job_data=None*)
:   Get the cardinality for this SubInput. The cardinality for a SubInputs is defined by the incoming link.

    > **Parameters key** (*SampleIndex* or *SampleId*) – key for a specific sample, can be sample index or id

> **Returns** the cardinality
>
> **Return type** int, sympy.Symbol, or None

description
> The description object of this input/output

dimensions
> List of dimension for this SubInput

fullid
> The full defining ID for the SubInput

get_sourced_nodes()
> Get a list of all *Nodes* connected as sources to this SubInput
>
> > **Returns** list of all connected *Nodes*
> >
> > **Return type** list

get_sourced_outputs()
> Get a list of all *Outputs* connected as sources to this SubInput
>
> > **Returns** list of all connected *Outputs*
> >
> > **Return type** list

input_group
> The id of the InputGroup this SubInputs parent belongs to.

iteritems()
> Iterate over the SampleItems that are in the SubInput.
>
> > **Returns** iterator yielding SampleItem objects

itersubinputs()
> Iterate over SubInputs (for a SubInput it will yield self and stop iterating after that)
>
> > **Returns** iterator yielding *SubInput*
>
> example:

```
>>> for subinput in input_a.itersubinputs():
        print subinput
```

node
> The Node to which this SubInputs parent belongs

remove(*value*)
> Remove a SubInput from parent Input.
>
> > **Parameters** **value** (*SubInput*) – the *SubInput* to removed from this Input

source
> A list with the source *Link*. The list is to be compatible with *Input*

source_output
> The *Output* linked to this SubInput

class fastr.planning.inputoutput.**SubOutput**(*output*, *index*)
> Bases: *fastr.planning.inputoutput.Output*

The SubOutput is an Output that represents a slice of another Output.

__abstractmethods__ = frozenset()

__eq__(*other*)

Compare two SubOutput instances with each other. This function ignores the parent, node and update status, but tests rest of the dict for equality. equality

> **Parameters** **other** (`SubOutput`) – the other instances to compare to
>
> **Returns** True if equal, False otherwise
>
> **Return type** bool

__getstate__()

Retrieve the state of the SubOutput

> **Returns** the state of the object
>
> **Rtype dict**

__hash__ = None

__init__(*output*, *index*)

Instantiate a SubOutput

> **Parameters**
>
> - **output** – the parent output the suboutput slices.
> - **index** (*int or slice*) – the way to slice the parent output
>
> **Returns** created SubOutput
>
> **Raises**
>
> - **`FastrTypeError`** – if the output argument is not an instance of `Output`
> - **`FastrTypeError`** – if the index argument is not an int or slice

__len__()

Return the length of the Output.

Note: In a SubOutput this is always 1.

__module__ = 'fastr.planning.inputoutput'

__setstate__(*state*)

Set the state of the SubOutput by the given state.

> **Parameters** **state** (*dict*) – The state to populate the object with
>
> **Returns** None

__str__()

Get a string version for the SubOutput

> **Returns** the string version
>
> **Return type** str

cardinality()

Cardinality of this SubOutput depends on the parent Output and `self.index`

> **Parameters** **key** (tuple of int or `SampleId`) – key for a specific sample, can be sample index or id
>
> **Returns** the cardinality
>
> **Return type** int, sympy.Symbol, or None

Raises

- *FastrCardinalityError* – if cardinality references an invalid *Input*
- *FastrTypeError* – if the referenced cardinality values type cannot be case to int
- *FastrValueError* – if the referenced cardinality value cannot be case to int

datatype
> The datatype of this SubOutput

fullid
> The full defining ID for the SubOutput

indexrep
> Simple representation of the index.

listeners
> The list of *Links* connected to this Output.

node
> The NodeRun to which this SubOutput belongs

preferred_types
> The list of preferred DataTypes for this SubOutput.

resulting_datatype
> The DataType that will the results of this SubOutput will have.

samples
> The SampleCollection for this SubOutput

`link` Module

The link module contain the Link class. This class represents the links in a network. These links lead from an output (BaseOutput) to an input (BaseInput) and indicate the desired data flow. Links are smart objects, in the sense that when you set their start or end point, they register themselves with the Input and Output. They do all the book keeping, so as long as you only set the source and target of the Link, the link should be valid.

> **Warning:** Don't mess with the Link, Input and Output internals from other places. There will be a huge chances of breaking the network functionality!

class fastr.planning.link.**Link**(*source*, *target*, *parent*, *id_=None*, *collapse=None*, *expand=None*)
> Bases: *fastr.core.dimension.HasDimensions*, fastr.abc.updateable.Updateable, fastr.abc.serializable.Serializable

Class for linking outputs (*BaseOutput*) to inputs (*BaseInput*)

Examples:

```
>>> import fastr
>>> network = fastr.create_network()
>>> link1 = network.create_link( n1.ouputs['out1'], n2.inputs['in2'] )

link2 = Link()
link2.source = n1.ouputs['out1']
link2.target = n2.inputs['in2']
```

__abstractmethods__ = frozenset()

__dataschemafile__ = 'Link.schema.json'

__eq__(*other*)
>   Test for equality between two Links

>>   **Parameters other** (*Link*) – object to test against

>>   **Return type** bool

>>   **Returns** True for equality, False otherwise

__getstate__()
>   Retrieve the state of the Link

>>   **Return type** dict

>>   **Returns** the state of the object

__hash__ = None

__init__(*source*, *target*, *parent*, *id_=None*, *collapse=None*, *expand=None*)
>   Create a new Link in a Network.

>>   **Parameters**

>>>   • **source** (*BaseOutput*) – the source output

>>>   • **target** (*BaseInput*) – the target input

>>>   • **parent** (*Network* or None) – the parent network

>>>   • **id** (*str or None*) – the id of the link, if no **id_** is given, the id will be in the form of "link_{:d}"

>>>   • **collapse** (*int, str, or tuple of int/str*) – the dimensions that the link has to collapse on

>>>   • **expand** (*bool*) – Does this link need to expand the cardinality into a new sample dimension

>>   **Returns** newly created Link

>>   **Raises**

>>>   • **FastrValueError** – if parent is not given

>>>   • **FastrValueError** – if the source output is not in the same network as the Link

>>>   • **FastrValueError** – if the target input is not in the same network as the Link

__module__ = 'fastr.planning.link'

__repr__()
>   Get a string representation for the Link

>>   **Return type** str

__setstate__(*state*)
>   Set the state of the Link by the given state.

>>   **Parameters state** (*dict*) – The state to populate the object with

>>   **Returns** None

>>   **Raises** **FastrValueError** – if the parent network is not set

---

cardinality(*index=None*)
> Cardinality for a Link is given by source Output and the collapse/expand settings

>> **Parameters index** (`Optional`[*SampleIndex*]) – index for a specific sample (can be only a sample index!)

>> **Return type** `Union`[`int`, `Symbol`]

>> **Returns** the cardinality

>> **Raises** *FastrIndexError* – if the index length does not match the number of dimension in the data

collapse
> The converging dimensions of this link. Collapsing changes some dimensions of sample lists into cardinality, reshaping the data.

> Collapse can be set to a tuple or an int/str, in which case it will be automatically wrapped in a tuple. The int will be seen as indices of the dimensions to collapse. The str will be seen as the name of the dimensions over which to collapse.

>> **Raises** *FastrTypeError* – if assigning a collapse value of a wrong type

collapse_indexes
> The converging dimensions of this link as integers. Dimension names are replaces with the corresponding int.

> Collapsing changes some dimensions of sample lists into cardinality, reshaping the data

classmethod createobj(*state*, *network=None*)
> Create object function for Link

>> **Parameters**

>>> • **cls** – The class to create

>>> • **state** (`dict`) – The state to use to create the Link

>>> • **network** – the parent Network

>> **Returns** newly created Link

destroy()
> The destroy function of a link removes all default references to a link. This means the references in the network, input and output connected to this link. If there is no references in other places in the code, it will destroy the link (reference count dropping to zero).

> This function is called when a source for an input is set to another value and the links becomes disconnected. This makes sure there is no dangling links.

dimensions
> The dimensions of the data delivered by the link. This can be different from the source dimensions because the link can make data collapse or expand.

draw(*context*, *graph*)

expand
> Flag indicating that the link will expand the cardininality into a new sample dimension to be created.

fullid
> The full defining ID for the Input

parent
> The Network to which this Link belongs.

source
> The source [`BaseOutput`] of the Link. Setting the source will automatically register the Link with the
> source BaseOutput. Updating source will also make sure the Link is unregistered with the previous source.
>
> > **Raises** [`FastrTypeError`] – if assigning a non [`BaseOutput`]

status

target
> The target [`BaseInput`] of the Link. Setting the target will automatically register the Link with the target
> BaseInput. Updating target will also make sure the Link is unregistered with the previous target.
>
> > **Raises** [`FastrTypeError`] – if assigning a non [`BaseInput`]

`network` Module

Network module containing Network facilitators and analysers.

class fastr.planning.network.**Network**(*id_='unnamed_network'*, *version=None*, *file-name=None*)
> Bases: `fastr.abc.serializable.Serializable`

The NetworkRun contains the entire Run state for a Network execution. It has a working copy of the network,
but also includes all temporary data required for the execution. These objects are meant to be single use.

NETWORK_DUMP_FILE_NAME = '__fastr_network__.json'

SINK_DUMP_FILE_NAME = '__sink_data__.json'

SOURCE_DUMP_FILE_NAME = '__source_data__.pickle.gz'

__dataschemafile__ = 'Network.schema.json'

__eq__(*other*)
> Compare two Networks and see if they are equal.
>
> > **Parameters other** ([`Network`]) –
> >
> > **Returns** flag indicating that the Networks are the same
> >
> > **Return type** [bool]

__getitem__(*item*)
> Get an item by its fullid. The fullid can point to a link, node, input, output or even subinput/suboutput.
>
> > **Parameters item** ([`str, unicode`]) – fullid of the item to retrieve
> >
> > **Returns** the requested item

__getstate__()
> Retrieve the state of the Network
>
> > **Returns** the state of the object
> >
> > **Rtype dict**

__hash__ = None

__init__(*id_='unnamed_network'*, *version=None*, *filename=None*)
> Create a new, empty Network
>
> > **Parameters name** ([`str`]) – name of the Network
> >
> > **Returns** newly created Network
> >
> > **Raises** [`OSError`] – if the tmp mount in the config is not a writable directory

__module__ = **'fastr.planning.network'**

__ne__(*other*)
    Tests for non-equality, this is the negated version __eq__

__repr__()
    Return repr(self).

__setstate__(*state*)
    Set the state of the Network by the given state. This completely overwrites the old state!

>   **Parameters state** (*dict*) – The state to populate the object with

>   **Returns** None

add_link(*link*)
    Add a Link to the Network. Make sure the link is in the link list and the link parent is set to this Network

>   **Parameters link** (*Link*) – link to add

>   **Raises**

>   - *FastrTypeError* – if link is incorrectly typed

>   - *FastrNetworkMismatchError* – if the link already belongs to another Network

add_node(*node*)
    Add a Node to the Network. Make sure the node is in the node list and the node parent is set to this Network

>   **Parameters node** (*Node*) – node to add

>   **Raises** *FastrTypeError* – if node is incorrectly typed

add_stepid(*stepid*, *node*)
    Add a Node to a specific step id

>   **Parameters**

>   - **stepid** (*str*) – the stepid that the node will be added to

>   - **node** (*Node*) – the node to add to the stepid

check_id(*id_*)
    Check if an id for an object is valid and unused in the Network. The method will always returns True if it does not raise an exception.

>   **Parameters id** (*str*) – the id to check

>   **Returns** True

>   **Raises**

>   - *FastrValueError* – if the id is not correctly formatted

>   - *FastrValueError* – if the id is already in use

create_constant(*datatype*, *data*, *id_=None*, *stepid=None*, *resources=None*, *nodegroup=None*)
    Create a ConstantNode in this Network. The Node will be automatically added to the Network.

>   **Parameters**

>   - **datatype** (BaseDataType) – The DataType of the constant node

>   - **data** (*datatype or list of datatype*) – The data to hold in the constant node

>   - **id** (*str*) – The id of the constant node to be created

- **stepid** (`str`) – The stepid to add the created constant node to

- **resources** – The resources required to run this node

- **nodegroup** (`str`) – The group the node belongs to, this can be important for FlowNodes and such, as they will have matching dimension names.

Returns the newly created constant node

Return type *ConstantNode*

create_link (*source*, *target*, *id_=None*, *collapse=None*, *expand=None*)
  Create a link between two Nodes and add it to the current Network.

  Parameters

  - **source** (*BaseOutput*) – the output that is the source of the link

  - **target** (*BaseInput*) – the input that is the target of the link

  - **id** (`str`) – the id of the link

  Returns the created link

  Type *Link*

create_macro (*network*, *resources=None*, *id_=None*)

create_node (*tool*, *tool_version*, *id_=None*, *stepid=None*, *resources=None*, *nodegroup=None*)
  Create a Node in this Network. The Node will be automatically added to the Network.

  Parameters

  - **tool** (*Tool*) – The Tool to base the Node on

  - **id** (`str`) – The id of the node to be created

  - **stepid** (`str`) – The stepid to add the created node to

  - **resources** – The resources required to run this node

  - **nodegroup** (`str`) – The group the node belongs to, this can be important for FlowNodes and such, as they will have matching dimension names.

  Returns the newly created node

  Return type *Node*

create_reference (*source_data*, *output_directory*)

create_sink (*datatype*, *id_=None*, *stepid=None*, *resources=None*, *nodegroup=None*)
  Create a SinkNode in this Network. The Node will be automatically added to the Network.

  Parameters

  - **datatype** (BaseDataType) – The DataType of the sink node

  - **id** (`str`) – The id of the sink node to be created

  - **stepid** (`str`) – The stepid to add the created sink node to

  - **resources** – The resources required to run this node

  Returns the newly created sink node

  Return type *SinkNode*

create_source (*datatype*, *id_=None*, *stepid=None*, *resources=None*, *nodegroup=None*)
  Create a SourceNode in this Network. The Node will be automatically added to the Network.

Parameters

- **datatype** (`BaseDataType`) – The DataType of the source source_node
- **id** (`str`) – The id of the source source_node to be created
- **stepid** (`str`) – The stepid to add the created source source_node to
- **resources** – The resources required to run this node
- **nodegroup** (`str`) – The group the node belongs to, this can be important for FlowNodes and such, as they will have matching dimension names.

Returns the newly created source source_node

Return type `SourceNode`

draw (*name=None*, *draw_dimensions=True*, *hide_unconnected=True*, *context=None*, *graph=None*, *expand_macro=False*)

draw_network (*name='network_layout'*, *img_format='svg'*, *draw_dimension=True*, *hide_unconnected=True*, *expand_macro=False*)
Output a dot file and try to convert it to an image file.

Parameters **img_format** (`str`) – extension of the image format to convert to

Returns path of the image created or None if failed

Return type str or None

execute (*sourcedata*, *sinkdata*, *blocking=True*, ***kwargs*)

fullid
The fullid of the Network, within the network scope

global_id
The global id of the Network, this is different for networks used in macronodes, as they still have parents.

id
The id of the Network. This is a read only property.

is_valid ()

namespace = None
The namespace this network lives in, this will be set by the NetworkManager on load

nodegroups
Give an overview of the nodegroups in the network

ns_id
The namespace and id of the Tool

remove (*value*)
Remove an item from the Network.

Parameters **value** (`Node` or `Link`) – the item to remove

classmethod test (*reference_data_dir*, *network=None*, *source_data=None*, *force_remove_temp=False*, *tmp_results_dir=None*)
Execute the network with the source data specified and test the results against the refence data. This effectively tests the network execution.

Parameters

- **reference_data_dir** (`str`) – The path or vfs url of reference data to compare with
- **source_data** (`dict`) – The source data to use

- **force_remove_temp** – Make sure the tmp results directory is cleaned at end of test

- **tmp_results_dir** – Path to results directory

node **Module**

A module to maintain a network node.

Exported classes:

Node – A class encapsulating a tool. ConstantNode – A node encapsulating an Output to set scalar values. SourceNode – A class providing a handle to a file.

class fastr.planning.node.**AdvancedFlowNode**(*tool*, *id_=None*, *parent=None*, *resource_limits=None*, *nodegroup=None*)

  Bases: *fastr.planning.node.FlowNode*

  __abstractmethods__ = frozenset()

  __module__ = 'fastr.planning.node'

class fastr.planning.node.**BaseNode**

  Bases: *fastr.core.dimension.HasDimensions*, fastr.abc.updateable.Updateable, fastr.abc.serializable.Serializable

  NODE_TYPES = {'AdvancedFlowNode': <class 'fastr.planning.node.AdvancedFlowNode'>, 'Co

  __abstractmethods__ = frozenset({'dimensions', '_update'})

  classmethod __init_subclass__(***kwargs*)
    Register nodes in class for easly location

  __module__ = 'fastr.planning.node'

class fastr.planning.node.**ConstantNode**(*datatype*, *data*, *id_=None*, *parent=None*, *resource_limits=None*, *nodegroup=None*)

  Bases: *fastr.planning.node.SourceNode*

  Class encapsulating one output for which a value can be set. For example used to set a scalar value to the input of a node.

  __abstractmethods__ = frozenset()

  __dataschemafile__ = 'ConstantNode.schema.json'

  __getstate__()
    Retrieve the state of the ConstantNode

    Returns the state of the object

    Rtype dict

  __init__(*datatype*, *data*, *id_=None*, *parent=None*, *resource_limits=None*, *nodegroup=None*)
    Instantiation of the ConstantNode.

    Parameters

    - **datatype** – The datatype of the output.

    - **data** – the prefilled data to use.

    - **id** – The url pattern.

This class should never be instantiated directly (unless you know what you are doing). Instead create a constant using the network class like shown in the usage example below.

usage example:

```
>>> import fastr
>>> network = fastr.create_network()
>>> source = network.create_source(datatype=types['ITKImageFile'], id_=
↪'sourceN')
```

or alternatively create a constant node by assigning data to an item in an InputDict:

```
>>> node_a.inputs['in'] = ['some', 'data']
```

which automatically creates and links a ConstantNode to the specified Input

__module__ = 'fastr.planning.node'

__setstate__(*state*)

Set the state of the ConstantNode by the given state.

> **Parameters state** (*dict*) – The state to populate the object with

> **Returns** None

data

The data stored in this constant node

draw(*context*, *graph*, *color=None*)

print_value

set_data(*data=None*, *ids=None*)

Set the data of this constant node in the correct way. This is mainly for compatibility with the parent class SourceNode

> **Parameters**

> • **data** (*dict or list of urls*) – the data to use

> • **ids** – if data is a list, a list of accompanying ids

class fastr.planning.node.**FlowNode**(*tool*, *id_=None*, *parent=None*, *resource_limits=None*, *nodegroup=None*)

Bases: *fastr.planning.node.Node*

A Flow Node is a special subclass of Nodes in which the amount of samples can vary per Output. This allows non-default data flows.

__abstractmethods__ = frozenset()

__init__(*tool*, *id_=None*, *parent=None*, *resource_limits=None*, *nodegroup=None*)

Instantiate a flow node.

> **Parameters**

> • **tool** (*Tool*) – The tool to base the node on

> • **id** (*str*) – the id of the node

> • **parent** (*Network*) – the parent network of the node

> **Returns** the newly created FlowNode

__module__ = 'fastr.planning.node'

> **blocking**
> A FlowNode is (for the moment) always considered blocking.
>
> > **Returns** True

> **dimensions**
> Names of the dimensions in the Node output. These will be reflected in the SampleIdList of this Node.

> **outputsize**
> Size of the outputs in this Node

class fastr.planning.node.**InputDict**
> Bases: `collections.OrderedDict`

The container containing the Inputs of Node. Implements helper functions for the easy linking syntax.

> **__module__ = 'fastr.planning.node'**

> **__setitem__**(*key*, *value*)
> Set an item in the input dictionary. The behaviour depends on the type of the value. For a *BaseInput*, the input will simply be added to the list of inputs. For a *BaseOutput*, a link between the output and input will be created.
>
> > **Parameters**
> >
> > - **key** (*str*) – id of the input to assign/link
> >
> > - **value** (*BaseInput* or *BaseOutput*) – either the input to add or the output to link
> >
> > - **dict_setitem** – the setitem function to use for the underlying OrderedDict insert

class fastr.planning.node.**MacroNode**(*value*, *id_=None*, *parent=None*, *resource_limits=None*, *nodegroup=None*)
> Bases: *fastr.planning.node.Node*

MacroNode encapsulates an entire network in a single node.

> **__abstractmethods__ = frozenset()**

> **__eq__**(*other*)
> Compare two MacroNode instances with each other. This function ignores the parent and update status, but tests rest of the dict for equality. equality
>
> > **Parameters other** (*MacroNode*) – the other instances to compare to
> >
> > **Returns** True if equal, False otherwise

> **__getstate__**()
> Retrieve the state of the MacroNode
>
> > **Returns** the state of the object
> >
> > **Rtype dict**

> **__hash__ = None**

> **__init__**(*value*, *id_=None*, *parent=None*, *resource_limits=None*, *nodegroup=None*)
>
> > **Parameters network** (*Network*) – network to create macronode for

> **__module__ = 'fastr.planning.node'**

> **__setstate__**(*state*)
> Set the state of the Node by the given state.
>
> > **Parameters state** (*dict*) – The state to populate the object with
> >
> > **Returns** None

draw (*context*, *graph*, *color=None*)

draw_link_target (*context*, *port_name*, *input=True*)

get_output_info (*output*)
 This functions maps the output dimensions based on the input dimensions of the macro. This is cached for speed as this can become rather costly otherwise

> **Parameters output** – output to get info for

> **Returns** tuple of Dimensions

network

class fastr.planning.node.**Node** (*tool*, *id_=None*, *node_class=None*, *parent=None*, *resource_limits=None*, *nodegroup=None*)
 Bases: *fastr.planning.node.BaseNode*

The class encapsulating a node in the network. The node is responsible for setting and checking inputs and outputs based on the description provided by a tool instance.

__abstractmethods__ = frozenset()

__dataschemafile__ = 'Node.schema.json'

__eq__ (*other*)
 Check two Node instances for equality.

> **Parameters other** (*Node*) – the other instances to compare to

> **Returns** True if equal, False otherwise

__getstate__ ()
 Retrieve the state of the Node

> **Returns** the state of the object

> **Rtype dict**

__hash__ = None

__init__ (*tool*, *id_=None*, *node_class=None*, *parent=None*, *resource_limits=None*, *nodegroup=None*)
 Instantiate a node.

> **Parameters**
>
> - **tool** (*Tool*) – The tool to base the node on
>
> - **id** (*str*) – the id of the node
>
> - **node_class** (*str*) – The class of the NodeRun to create (e.g. SourceNodeRun, NodeRun)
>
> - **parent** (*Network*) – the parent network of the node
>
> - **cores** (*int*) – number of cores required for executing this Node
>
> - **memory** (*str*) – amount of memory required in the form d+[mMgG] where M is for megabyte and G for gigabyte
>
> - **walltime** (*str*) – amount of time required in second or in the form HOURS:MINUTES:SECOND
>
> **Returns** the newly created Node

__module__ = 'fastr.planning.node'

__ne__(*other*)

Check two Node instances for inequality. This is the inverse of __eq__

> **Parameters other** ([Node](#)) – the other instances to compare to
>
> **Returns** True if unequal, False otherwise

__repr__()

Get a string representation for the Node

> **Returns** the string representation
>
> **Return type** [str](#)

__setstate__(*state*)

Set the state of the Node by the given state.

> **Parameters state** ([dict](#)) – The state to populate the object with
>
> **Returns** None

__str__()

Get a string version for the Node

> **Returns** the string version
>
> **Return type** [str](#)

blocking

Indicate that the results of this Node cannot be determined without first executing the Node, causing a blockage in the creation of jobs. A blocking Nodes causes the Chunk borders.

classmethod createobj(*state*, *network=None*)

Create object function for generic objects

> **Parameters**
>
> - **cls** – The class to create
> - **state** – The state to use to create the Link
> - **network** – the parent Network
>
> **Returns** newly created Link

dimensions

The dimensions has to be implemented by any subclass. It has to provide a tuple of Dimensions.

> **Returns** dimensions
>
> **Return type** [tuple](#)

dimnames

Names of the dimensions in the Node output. These will be reflected in the SampleIdList of this Node.

draw(*context*, *graph*, *color=None*)

draw_id(*context*)

draw_link_target(*context*, *port_name*, *input=True*)

find_source_index(*target_index*, *target*, *source*)

fullid

The full defining ID for the Node inside the network

get_sourced_nodes()

A list of all Nodes connected as sources to this Node

> **Returns** list of all nodes that are connected to an input of this node

global_id
> The global defining ID for the Node from the main network (goes out of macro nodes to root network)

id
> The id of the Node

input_groups

> **A list of input groups for this Node. An input group is InputGroup** object filled according to the
> > Node

inputs = None
> A list of inputs of this Node

listeners
> All the listeners requesting output of this node, this means the listeners of all Outputs and SubOutputs

merge_dimensions

name
> Name of the Tool the Node was based on. In case a Toolless Node was used the class name is given.

nodegroup

outputs = None
> A list of outputs of this Node

outputsize
> The size of output of this SourceNode

parent
> The parent is the Network this Node is part of

status

tool

update_input_groups()
> Update all input groups in this node

class fastr.planning.node.**OutputDict**
> Bases: `collections.OrderedDict`

The container containing the Inputs of Node. Only checks if the inserted values are actually outputs.

__module__ = 'fastr.planning.node'

__setitem__(*key*, *value*)
> Set an output.

> > **Parameters**

> > > • **key** (*str*) – the of the item to set

> > > • **value** (*BaseOutput*) – the output to set

> > > • **dict_setitem** – the setitem function to use for the underlying OrderedDict insert

class fastr.planning.node.**SinkNode**(*datatype*, *id_=None*, *parent=None*, *resource_limits=None*,
> *nodegroup=None*)
> Bases: *fastr.planning.node.Node*

Class which handles where the output goes. This can be any kind of file, e.g. image files, textfiles, config files,
etc.

`__abstractmethods__ = frozenset()`

`__dataschemafile__ = 'SinkNode.schema.json'`

`__getstate__`()
> Retrieve the state of the Node

>> **Returns** the state of the object

>> **Rtype dict**

`__init__`(*datatype*, *id_=None*, *parent=None*, *resource_limits=None*, *nodegroup=None*)
> Instantiation of the SourceNode.

>> **Parameters**

>>> • **`datatype`** – The datatype of the output.

>>> • **`id`** – the id of the node to create

>> **Returns** newly created sink node

> usage example:

```
>>> import fastr
>>> network = fastr.create_network()
>>> sink = network.create_sink(datatype=types['ITKImageFile'], id_='SinkN')
```

`__module__ = 'fastr.planning.node'`

`__setstate__`(*state*)
> Set the state of the Node by the given state.

>> **Parameters** **`state`** (*[dict](#)*) – The state to populate the object with

>> **Returns** None

`datatype`
> The datatype of the data this sink can store.

`draw`(*context*, *graph*, *color=None*)

`input`
> The default input of the sink Node

`class` fastr.planning.node.**SourceNode**(*datatype*, *id_=None*, *parent=None*, *resource_limits=None*, *nodegroup=None*)
> Bases: *[fastr.planning.node.FlowNode](#)*

> Class providing a connection to data resources. This can be any kind of file, stream, database, etc from which data can be received.

`__abstractmethods__ = frozenset()`

`__dataschemafile__ = 'SourceNode.schema.json'`

`__getstate__`()
> Retrieve the state of the SourceNode

>> **Returns** the state of the object

>> **Rtype dict**

`__init__`(*datatype*, *id_=None*, *parent=None*, *resource_limits=None*, *nodegroup=None*)
> Instantiation of the SourceNode.

>> **Parameters**

- **datatype** – The (id of) the datatype of the output.

- **id** – The url pattern.

This class should never be instantiated directly (unless you know what you are doing). Instead create a source using the network class like shown in the usage example below.

usage example:

```
>>> import fastr
>>> network = fastr.create_network()
>>> source = network.create_source(datatype=types['ITKImageFile'], id_=
→'sourceN')
```

__module__ = 'fastr.planning.node'

__setstate__(*state*)
   Set the state of the SourceNode by the given state.

   Parameters state (*dict*) – The state to populate the object with

   Returns None

datatype
   The datatype of the data this source supplies.

dimensions
   The dimensions in the SourceNode output. These will be reflected in the SampleIdLists.

draw(*context*, *graph*, *color=None*)

nodegroup

output
   Shorthand for `self.outputs['output']`

set_data(*data*, *ids=None*)
   Set the data of this source node.

   Parameters

   - **data** (*dict, OrderedDict or list of urls*) – the data to use

   - **ids** – if data is a list, a list of accompanying ids

sourcegroup

valid
   This does nothing. It only overloads the valid method of Node(). The original is intended to check if the inputs are connected to some output. Since this class does not implement inputs, it is skipped.

Subpackages

test Package

test_network Module

test_node Module

plugins Package

`plugins` Package

The plugins module holds all plugins loaded by Fastr. It is empty on start and gets filled by the BasePluginManager

class `fastr.plugins.`**`BlockingExecution`**(*finished_callback=None*, *cancelled_callback=None*)

> Bases: *`fastr.plugins.executionplugin.ExecutionPlugin`*

> The blocking execution plugin is a special plugin which is meant for debug purposes. It will not queue jobs but immediately execute them inline, effectively blocking fastr until the Job is finished. It is the simplest execution plugin and can be used as a template for new plugins or for testing purposes.

> **`__abstractmethods__ = frozenset()`**

> **`__init__`**(*finished_callback=None*, *cancelled_callback=None*)
> > Setup the ExecutionPlugin

> > > **Parameters**

> > > > • **`finished_callback`** – the callback to call after a job finished

> > > > • **`cancelled_callback`** – the callback to call after a job cancelled

> > > **Returns** newly created ExecutionPlugin

> **`__module__ = 'fastr.plugins'`**

> **`cleanup`**()
> > Method to call to clean up the ExecutionPlugin. This can be to clear temporary data, close connections, etc.

> > > **Parameters** **`force`** – force cleanup (e.g. kill instead of join a process)

> **`filename = '/home/docs/checkouts/readthedocs.org/user_builds/fastr/envs/3.0.0/lib/pytho`**

> **`module = <module 'blockingexecution' from '/home/docs/checkouts/readthedocs.org/user_b`**

> **classmethod `test`**()
> > Test the plugin, default behaviour is just to instantiate the plugin

class `fastr.plugins.`**`CommaSeperatedValueFile`**

> Bases: *`fastr.core.ioplugin.IOPlugin`*

> The CommaSeperatedValueFile an expand-only type of IOPlugin. No URLs can actually be fetched, but it can expand a single URL into a larger amount of URLs.

> The `csv://` URL is a `vfs://` URL with a number of query variables available. The URL mount and path should point to a valid CSV file. The query variable then specify what column(s) of the file should be used.

> The following variable can be set in the query:

variable	usage
value	the column containing the value of interest, can be int for index or string for key
id	the column containing the sample id (optional)
header	indicates if the first row is considered the header, can be `true` or `false` (optional)
delimiter	the delimiter used in the csv file (optional)
quote	the quote character used in the csv file (optional)
reformat	a reformatting string so that `value = reformat.format(value)` (used before relative_path)
relative_path	indicates the entries are relative paths (for files), can be `true` or `false` (optional)

The header is by default `false` if the neither the `value` and `id` are set as a string. If either of these are a string, the header is required to define the column names and it automatically is assumed `true`

The delimiter and quota characters of the file should be detected automatically using the `Sniffer`, but can be forced by setting them in the URL.

Example of valid `csv` URLs:

```
# Use the first column in the file (no header row assumed)
csv://mount/some/dir/file.csv?value=0

# Use the images column in the file (first row is assumed header row)
csv://mount/some/dir/file.csv?value=images

# Use the segmentations column in the file (first row is assumed header row)
# and use the id column as the sample id
csv://mount/some/dir/file.csv?value=segmentations&id=id

# Use the first column as the id and the second column as the value
# and skip the first row (considered the header)
csv://mount/some/dir/file.csv?value=1&id=0&header=true

# Use the first column and force the delimiter to be a comma
csv://mount/some/dir/file.csv?value=0&delimiter=,
```

__abstractmethods__ = frozenset()

__init__()
> Initialization for the IOPlugin

> > **Returns** newly created IOPlugin

__module__ = 'fastr.plugins'

expand_url(*url*)
> (`abstract`) Expand an URL. This allows a source to collect multiple samples from a single url. The URL will have a wildcard or point to something with info and multiple urls will be returned.

> > **Parameters url** (*str*) – url to expand

> > **Returns** the resulting url(s), a tuple if multiple, otherwise a str

> > **Return type** str or tuple of str

filename = '/home/docs/checkouts/readthedocs.org/user_builds/fastr/envs/3.0.0/lib/pyth

module = <module 'commaseperatedvaluefile' from '/home/docs/checkouts/readthedocs.org/

scheme = 'csv'

class fastr.plugins.**CrossValidation**
> Bases: `flowinterface.FlowPlugin`

Advanced flow plugin that generated a cross-validation data flow. The node need an input with data and an input number of folds. Based on that the outputs test and train will be supplied with a number of data sets.

__abstractmethods__ = frozenset()

__module__ = 'fastr.plugins'

static execute(*payload*)

filename = '/home/docs/checkouts/readthedocs.org/user_builds/fastr/envs/3.0.0/lib/pyth

module = <module 'crossvalidation' from '/home/docs/checkouts/readthedocs.org/user_bui

class fastr.plugins.**DRMAAExecution**(*finished_callback=None*, *cancelled_callback=None*)
    Bases: *fastr.plugins.executionplugin.ExecutionPlugin*

A DRMAA execution plugin to execute Jobs on a Grid Engine cluster. It uses a configuration option for selecting the queue to submit to. It uses the python `drmaa` package.

Note: To use this plugin, make sure the `drmaa` package is installed and that the execution is started on an SGE submit host with DRMAA libraries installed.

Note: This plugin is at the moment tailored to SGE, but it should be fairly easy to make different subclasses for different DRMAA supporting systems.

CANCELS_DEPENDENCIES = False

GE_NATIVE_SPEC = {'CWD': '-cwd', 'DEPENDS': '-hold_jid {hold_list}', 'DEPENDS_SEP': ',

NATIVE_SPEC = {'grid_engine': {'CWD': '-cwd', 'DEPENDS': '-hold_jid {hold_list}', 'DEP

SUPPORTS_CANCEL = True

SUPPORTS_DEPENDENCY = True

SUPPORTS_HOLD_RELEASE = True

TORQUE_NATIVE_SPEC = {'CWD': '', 'DEPENDS': '-W depend=afterok:{hold_list}', 'DEPENDS_

__abstractmethods__ = frozenset()

__init__(*finished_callback=None*, *cancelled_callback=None*)
    Setup the ExecutionPlugin

        Parameters

            • **finished_callback** – the callback to call after a job finished

            • **cancelled_callback** – the callback to call after a job cancelled

        Returns newly created ExecutionPlugin

__module__ = 'fastr.plugins'

cleanup()
    Method to call to clean up the ExecutionPlugin. This can be to clear temporary data, close connections, etc.

        Parameters **force** – force cleanup (e.g. kill instead of join a process)

collect_jobs()

configuration_fields = {'drmaa_engine': (<class 'str'>, 'grid_engine', 'The engine to

create_native_spec(*queue*, *walltime*, *memory*, *ncores*, *outputLog*, *errorLog*, *hold_job*, *hold*)
    Create the native spec for the DRMAA scheduler. Needs to be implemented in the subclasses

        Parameters

            • **queue** (*str*) – the queue to submit to

            • **walltime** (*str*) – walltime specified

            • **memory** (*str*) – memory requested

            • **ncores** (*int*) – number of cores requested

- **outputLog** (*str*) – the location of the stdout log

- **errorLog** (*str*) – the location of stderr log

- **hold_job** (*list*) – list of jobs to depend on

- **hold** (*bool*) – flag if job should be submitted in hold mode

>   Returns

current_jobs

dispatch_callbacks()

filename = '/home/docs/checkouts/readthedocs.org/user_builds/fastr/envs/3.0.0/lib/pyth

job_regression_check()

module = <module 'drmaaexecution' from '/home/docs/checkouts/readthedocs.org/user_buil

send_job(*command*, *arguments*, *queue=None*, *resources=None*, *job_name=None*, *joinLog-Files=False*, *outputLog=None*, *errorLog=None*, *hold_job=None*, *hold=False*)

spec_fields

submit_jobs()

classmethod test()
>   Test the plugin, default behaviour is just to instantiate the plugin

class fastr.plugins.**DockerTarget**(*binary*, *docker_image*)
>   Bases: *fastr.core.target.Target*

A tool target that is located in a Docker images. Can be run using docker-py. A docker target only need two variables: the binary to call within the docker container, and the docker container to use.

```
{
  "arch": "*",
  "os": "*",
  "binary": "bin/test.py",
  "docker_image": "fastr/test"
}
```

```
<target os="*" arch="*" binary="bin/test.py" docker_image="fastr/test">
```

__abstractmethods__ = frozenset()

__enter__()
>   Set the environment in such a way that the target will be on the path.

__exit__(*exc_type*, *exc_value*, *traceback*)
>   Cleanup the environment where needed

__init__(*binary*, *docker_image*)
>   Define a new docker target.

>>   **Parameters docker_image** (*str*) – Docker image to use

__module__ = 'fastr.plugins'

container

filename = '/home/docs/checkouts/readthedocs.org/user_builds/fastr/envs/3.0.0/lib/pyth

module = <module 'dockertarget' from '/home/docs/checkouts/readthedocs.org/user_builds

monitor_docker(*container*, *resources*)
  Monitor a docker container and profile the cpu, memory and io use. Register the resource use every _MONITOR_INTERVAL seconds.

  Parameters

  • **container** (*ContainerCollection*) – process to monitor

  • **resources** (*ProcessUsageCollection*) – list to append measurements to

run_command(*command*)
  Run a command with the target

  Return type *TargetResult*

class fastr.plugins.**ElasticsearchReporter**
  Bases: *fastr.plugins.reportingplugin.ReportingPlugin*

  __abstractmethods__ = frozenset()

  __init__()
    The BasePlugin constructor.

    Returns the created plugin

    Return type BasePlugin

    Raises *FastrPluginNotLoaded* – if the plugin did not load correctly

  __module__ = 'fastr.plugins'

  activate()
    Activate the reporting plugin

  configuration_fields = {'elasticsearch_debug': (<class 'bool'>, False, 'Setup elastic

  elasticsearch_update_status(*job*)

  filename = '/home/docs/checkouts/readthedocs.org/user_builds/fastr/envs/3.0.0/lib/pyth

  job_updated(*job*)

  module = <module 'elasticsearchreporter' from '/home/docs/checkouts/readthedocs.org/use

  classmethod test()
    Test the plugin, default behaviour is just to instantiate the plugin

class fastr.plugins.**FastrInterface**(*id_*, *document*)
  Bases: *fastr.core.interface.Interface*

  The default Interface for fastr. For the command-line Tools as used by fastr. It build a commandline call based on the input/output specification.

  The fields that can be set in the interface:

Attribute		Description
id		The id of this Tool (used internally in fastr)
inputs[]		List of Inputs that can are accepted by the Tool
	id	ID of the Input
	name	Longer name of the Input (more human readable)

Continued on next page

Table 3.1 – continued from previous page

Attribute		Description
	datatype	The ID of the DataType of the Input[1]
	enum[]	List of possible values for an EnumType (created on the fly by fastr)[1]
	prefix	Commandline prefix of the Input (e.g. –in, -i)
	cardinality	Cardinality of the Input
	repeat_prefix	Flag indicating if for every value of the Input the prefix is repeated
	required	Flag indicating if the input is required
	nospace	Flag indicating if there is no space between prefix and value (e.g. –in=val)
	format	For DataTypes that have multiple representations, indicate which one to use
	default	Default value for the Input
	description	Long description for an input
outputs[]		List of Outputs that are generated by the Tool (and accessible to fastr)
	id	ID of the Output
	name	Longer name of the Output (more human readable)
	datatype	The ID of the DataType of the Output[1]
	enum[]	List of possible values for an EnumType (created on the fly by fastr)[1]
	prefix	Commandline prefix of the Output (e.g. –out, -o)
	cardinality	Cardinality of the Output
	repeat_prefix	Flag indicating if for every value of the Output the prefix is repeated
	required	Flag indicating if the input is required
	nospace	Flag indicating if there is no space between prefix and value (e.g. –out=val)
	format	For DataTypes that have multiple representations, indicate which one to use
	description	Long description for an input
	action	Special action (defined per DataType) that needs to be performed before creating output value (e.g. 'ensure' will make sure an output directory exists)

Continued on next page

Table 3.1 – continued from previous page

Attribute		Description
	`automatic`	Indicate that output doesn't require commandline argument, but is created automatically by a Tool[2]
	`method`	The collector plugin to use for the gathering automatic output, see the *Collector plugins*
	`location`	Definition where to an automatically, usage depends on the `method`[2]

> **`__abstractmethods__ = frozenset()`**
>
> **`__dataschemafile__ = 'FastrInterface.schema.json'`**
>
> **`__eq__`**(*other*)
>> Return self==value.
>
> **`__getstate__`**()
>> Get the state of the FastrInterface object.
>>
>>> **Returns** state of interface
>>>
>>> **Return type** dict
>
> **`__hash__ = None`**
>
> **`__init__`**(*id_*, *document*)
>> The BasePlugin constructor.
>>
>>> **Returns** the created plugin
>>>
>>> **Return type** BasePlugin
>>>
>>> **Raises** *FastrPluginNotLoaded* – if the plugin did not load correctly
>
> **`__module__ = 'fastr.plugins'`**
>
> **`__setstate__`**(*state*)
>> Set the state of the Interface
>
> **`check_input_id`**(*id_*)
>> Check if an id for an object is valid and unused in the Tool. The method will always returns True if it does not raise an exception.
>>
>>> **Parameters** **id**(*str*) – the id to check
>>>
>>> **Returns** True
>>>
>>> **Raises**
>>>
>>> • *FastrValueError* – if the id is not correctly formatted
>>> • *FastrValueError* – if the id is already in use
>
> **`check_output_id`**(*id_*)
>> Check if an id for an object is valid and unused in the Tool. The method will always returns True if it does not raise an exception.

---

[1] `datatype` and `enum` are conflicting entries, if both specified `datatype` has presedence
[2] More details on defining automatica output are given in [TODO]

> Parameters **id** (*str*) – the id to check
>
> Returns True
>
> Raises
>
> > • *FastrValueError* – if the id is not correctly formatted
> >
> > • *FastrValueError* – if the id is already in use

static collect_errors(*result*)
> Special error collection for fastr interfaces

collect_results(*result*)
> Collect all results of the interface

collector_plugin_type
> alias of fastrinterface.CollectorPlugin

collectors = CollectorPluginManager [37m[42m[1mLoaded[0m json : <CollectorPlugin: Js

execute(*target*, *payload*)
> Execute the interface using a specific target and payload (containing a set of values for the arguments)
>
> Parameters
>
> > • **target** (*SampleId*) – the target to use
> >
> > • **payload** (*dict*) – the values for the arguments
>
> Returns result of the execution
>
> Return type *InterfaceResult*

expanding
> Indicates whether or not this Interface will result in multiple samples per run. If the flow is unaffected, this will be zero, if it is nonzero it means that number of dimension will be added to the sample array.

filename = '/home/docs/checkouts/readthedocs.org/user_builds/fastr/envs/3.0.0/lib/pyth

get_arguments(*values*)
> Get the argument list for this interface
>
> Returns return list of arguments

get_command(*target*, *payload*)

get_specials(*payload*, *output*, *cardinality_nr*)
> Get special attributes. Returns tuples for specials, inputs and outputs that are used for formatting substitutions.
>
> Parameters
>
> > • **output** – Output for which to get the specials
> >
> > • **cardinality_nr** (*int*) – the cardinality number

inputs
> OrderedDict of Inputs connected to the Interface. The format should be {input_id: InputSpec}.

module = <module 'fastrinterface' from '/home/docs/checkouts/readthedocs.org/user_buil

outputs
> OrderedDict of Output connected to the Interface. The format should be {output_id: OutputSpec}.

class fastr.plugins.**FileSystem**

> Bases: *fastr.core.ioplugin.IOPlugin*

> The FileSystem plugin is create to handle file:// type or URLs. This is generally not a good practice, as this is not portable over between machines. However, for test purposes it might be useful.

> The URL scheme is rather simple: file://host/path (see wikipedia for details)

> We do not make use of the host part and at the moment only support localhost (just leave the host empty) leading to file:/// URLs.

> > **Warning:** This plugin ignores the hostname in the URL and does only accept driver letters on Windows in the form c:/

> **__abstractmethods__ = frozenset()**

> **__init__**()
> > Initialization for the IOPlugin
> >
> > > **Returns** newly created IOPlugin

> **__module__ = 'fastr.plugins'**

> **fetch_url**(*inurl*, *outpath*)
> > Fetch the files from the file.
> >
> > > **Parameters**
> > >
> > > • **inurl** – url to the item in the data store, starts with file://
> > >
> > > • **outpath** – path where to store the fetch data locally

> **fetch_value**(*inurl*)
> > Fetch a value from an external file file.
> >
> > > **Parameters** **inurl** – url of the value to read
> > >
> > > **Returns** the fetched value

> **filename = '/home/docs/checkouts/readthedocs.org/user_builds/fastr/envs/3.0.0/lib/pytho**

> **module = <module 'filesystem' from '/home/docs/checkouts/readthedocs.org/user_builds/fa**

> **path_to_url**(*path*, *mountpoint=None*)
> > Construct an url from a given mount point and a relative path to the mount point.

> **put_url**(*inpath*, *outurl*)
> > Put the files to the external data store.
> >
> > > **Parameters**
> > >
> > > • **inpath** – path of the local data
> > >
> > > • **outurl** – url to where to store the data, starts with file://

> **put_value**(*value*, *outurl*)
> > Put the value in the external data store.
> >
> > > **Parameters**
> > >
> > > • **value** – value to store
> > >
> > > • **outurl** – url to where to store the data, starts with file://

> **scheme = 'file'**

---

url_to_path(*url*)

Get the path to a file from a url. Currently supports the file:// scheme

Examples:

```
>>> 'file:///d:/data/project/file.ext'
'd:\data\project\file.ext'
```

> **Warning:** file:// will not function cross platform and is mainly for testing

class fastr.plugins.**FlowInterface**(*id_*, *document*)

Bases: `fastr.core.interface.Interface`

The Interface use for AdvancedFlowNodes to create the advanced data flows that are not implemented in the fastr. This allows nodes to implement new data flows using the plugin system.

The definition of `FlowInterfaces` are very similar to the default `FastrInterfaces`.

> **Note:** A flow interface should be using a specific FlowPlugin

__abstractmethods__ = frozenset()

__dataschemafile__ = 'FastrInterface.schema.json'

__eq__(*other*)

Return self==value.

__getstate__()

Get the state of the FastrInterface object.

> **Returns** state of interface
>
> **Return type** dict

__hash__ = None

__init__(*id_*, *document*)

The BasePlugin constructor.

> **Returns** the created plugin
>
> **Return type** BasePlugin
>
> **Raises** *FastrPluginNotLoaded* – if the plugin did not load correctly

__module__ = 'fastr.plugins'

__setstate__(*state*)

Set the state of the Interface

execute(*target*, *payload*)

Execute the interface given the a target and payload. The payload should have the form {'input': {'input_id_a': (value, value), 'input_id_b': (value, value)},

'output': {'output_id_a': (value, value), 'output_id_b': (value, value)}}

> **Parameters**
>
> - **target** – the target to call
>
> - **payload** – the payload to use

> **Returns** the result of the execution
>
> **Return type** (tuple of) *InterfaceResult*

expanding
Indicates whether or not this Interface will result in multiple samples per run. If the flow is unaffected, this will be zero, if it is nonzero it means that number of dimension will be added to the sample array.

filename = '/home/docs/checkouts/readthedocs.org/user_builds/fastr/envs/3.0.0/lib/pytho

flow_plugin_type
alias of `flowinterface.FlowPlugin`

flow_plugins = FlowPluginManager [37m[42m[1mLoaded[0m CrossValidation :   <FlowPlugin:

inputs
OrderedDict of Inputs connected to the Interface. The format should be {input_id: InputSpec}.

module = <module 'flowinterface' from '/home/docs/checkouts/readthedocs.org/user_build

outputs
OrderedDict of Output connected to the Interface. The format should be {output_id: OutputSpec}.

class fastr.plugins.**HTTPPlugin**
Bases: *fastr.core.ioplugin.IOPlugin*

> **Warning:** This Plugin is still under development and has not been tested at all. example url: https: //server.io/path/to/resource

__abstractmethods__ = frozenset()

__init__()
Initialization for the IOPlugin

> **Returns** newly created IOPlugin

__module__ = 'fastr.plugins'

fetch_url(*inurl*, *outpath*)
Download file from server.

> **Parameters**
>
> - **inurl** – url to the file.
>
> - **outpath** – path to store file

filename = '/home/docs/checkouts/readthedocs.org/user_builds/fastr/envs/3.0.0/lib/pytho

module = <module 'httpplugin' from '/home/docs/checkouts/readthedocs.org/user_builds/fa

scheme = ('https', 'http')

class fastr.plugins.**LinearExecution**(*finished_callback=None*, *cancelled_callback=None*)
Bases: *fastr.plugins.executionplugin.ExecutionPlugin*

An execution engine that has a background thread that executes the jobs in order. The queue is a simple FIFO queue and there is one worker thread that operates in the background. This plugin is meant as a fallback when other plugins do not function properly. It does not multi-processing so it is safe to use in environments that do no support that.

__abstractmethods__ = frozenset()

__init__ (*finished_callback=None*, *cancelled_callback=None*)

Setup the ExecutionPlugin

> **Parameters**
>
> > • **finished_callback** – the callback to call after a job finished
> >
> > • **cancelled_callback** – the callback to call after a job cancelled
>
> **Returns** newly created ExecutionPlugin

__module__ = **'fastr.plugins'**

cleanup()

Method to call to clean up the ExecutionPlugin. This can be to clear temporary data, close connections, etc.

> **Parameters force** – force cleanup (e.g. kill instead of join a process)

exec_worker()

filename = '/home/docs/checkouts/readthedocs.org/user_builds/fastr/envs/3.0.0/lib/pytho

module = <module 'linearexecution' from '/home/docs/checkouts/readthedocs.org/user_bui

classmethod test()

Test the plugin, default behaviour is just to instantiate the plugin

class fastr.plugins.**LocalBinaryTarget** (*binary*, *paths=None*, *environment_variables=None*, *initscripts=None*, *modules=None*, *interpreter=None*, ***kwargs*)

Bases: *fastr.core.target.SubprocessBasedTarget*

A tool target that is a local binary on the system. Can be found using environmentmodules or a path on the executing machine. A local binary target has a number of fields that can be supplied:

- binary (required): the name of the binary/script to call, can also be called bin for backwards compatibility.

- modules: list of modules to load, this can be environmentmodules or lmod modules. If modules are given, the paths, environment_variables and initscripts are ignored.

- paths: a list of paths to add following the structure {"value": "/path/to/dir", "type": "bin"}. The types can be bin if the it should be added to $PATH or lib if it should be added to te library path (e.g. $LD_LIBRARY_PATH for linux).

- environment_variables: a dictionary of environment variables to set.

- initscript: a list of script to run before running the main tool

- interpreter: the interpreter to use to call the binary e.g. python

The LocalBinaryTarget will first check if there are modules given and the module subsystem is loaded. If that is the case it will simply unload all current modules and load the given modules. If not it will try to set up the environment itself by using the following steps:

1. Prepend the bin paths to $PATH

2. Prepend the lib paths to the correct environment variable

3. Setting the other environment variables given ($PATH and the system library path are ignored and cannot be set that way)

4. Call the initscripts one by one

The definition of the target in JSON is very straightforward:

```json
{
  "binary": "bin/test.py",
  "interpreter": "python",
  "paths": [
    {
      "type": "bin",
      "value": "vfs://apps/test/bin"
    },
    {
      "type": "lib",
      "value": "./lib"
    }
  ],
  "environment_variables": {
    "othervar": 42,
    "short_var": 1,
    "testvar": "value1"
  },
  "initscripts": [
    "bin/init.sh"
  ],
  "modules": ["elastix/4.8"]
}
```

In XML the definition would be in the form of:

```xml
<target os="linux" arch="*" modules="elastix/4.8" bin="bin/test.py" interpreter=
↪"python">
  <paths>
    <path type="bin" value="vfs://apps/test/bin" />
    <path type="lib" value="./lib" />
  </paths>
  <environment_variables short_var="1">
    <testvar>value1</testvar>
    <othervar>42</othervar>
  </environment_variables>
  <initscripts>
    <initscript>bin/init.sh</initscript>
  </initscripts>
</target>
```

DYNAMIC_LIBRARY_PATH_DICT = {'darwin': 'DYLD_LIBRARY_PATH', 'linux': 'LD_LIBRARY_PATH

__abstractmethods__ = frozenset()

__enter__()
> Set the environment in such a way that the target will be on the path.

__exit__(*exc_type*, *exc_value*, *traceback*)
> Cleanup the environment

__init__(*binary*, *paths=None*, *environment_variables=None*, *initscripts=None*, *modules=None*, *interpreter=None*, ***kwargs*)
> Define a new local binary target. Must be defined either using paths and optionally environment_variables and initscripts, or enviroment modules.

__module__ = 'fastr.plugins'

filename = '/home/docs/checkouts/readthedocs.org/user_builds/fastr/envs/3.0.0/lib/pytho

---

> **module** = <module 'localbinarytarget' from '/home/docs/checkouts/readthedocs.org/user_b

> **paths**

> **run_command**(*command*)
>> Run a command with the target

>>> **Return type** *TargetResult*

class fastr.plugins.**MacroTarget**(*network_file*, *method=None*, *function='main'*)
> Bases: *fastr.core.target.Target*

> A target for MacroNodes. This target cannot be executed as the MacroNode handles execution differently. But this contains the information for the MacroNode to find the internal Network.

> **__abstractmethods__ = frozenset()**

> **__init__**(*network_file*, *method=None*, *function='main'*)
>> Define a new local binary target. Must be defined either using paths and optionally environment_variables and initscripts, or enviroment modules.

> **__module__ = 'fastr.plugins'**

> **filename** = '/home/docs/checkouts/readthedocs.org/user_builds/fastr/envs/3.0.0/lib/pytho

> **module** =**run_command**(*command*)
>> Run a command with the target

> **classmethod test**()
>> Test if singularity is availble on the path

class fastr.plugins.**NipypeInterface**(*id_*, *nipype_cls=None*, *document=None*)
> Bases: *fastr.core.interface.Interface*

> Experimental interfaces to using nipype interfaces directly in fastr tools, only using a simple reference.

> To create a tool using a nipype interface just create an interface with the correct type and set the nipype argument to the correct class. For example in an xml tool this would become:

```
<interface class="NipypeInterface">
  <nipype_class>nipype.interfaces.elastix.Registration</nipype_class>
</interface>
```

> **Note:** To use these interfaces nipype should be installed on the system.

> **Warning:** This interface plugin is basically functional, but highly experimental!

> **__abstractmethods__ = frozenset()**

> **__eq__**(*other*)
>> Return self==value.

> **__getstate__**()
>> Retrieve the state of the Interface

>>> **Returns** the state of the object

>>> **Rtype dict**

__hash__ = None

__init__(*id_*, *nipype_cls=None*, *document=None*)
   The BasePlugin constructor.

   > **Returns** the created plugin

   > **Return type** BasePlugin

   > **Raises** *[FastrPluginNotLoaded](#)* – if the plugin did not load correctly

__module__ = 'fastr.plugins'

__setstate__(*state*)
   Set the state of the Interface

execute(*target*, *payload*)
   Execute the interface using a specific target and payload (containing a set of values for the arguments)

   > **Parameters**

   > - **target** (*[SampleId](#)*) – the target to use

   > - **payload** (*[dict](#)*) – the values for the arguments

   > **Returns** result of the execution

   > **Return type** *[InterfaceResult](#)*

expanding
   Indicates whether or not this Interface will result in multiple samples per run. If the flow is unaffected, this will be zero, if it is nonzero it means that number of dimension will be added to the sample array.

filename = '/home/docs/checkouts/readthedocs.org/user_builds/fastr/envs/3.0.0/lib/pyth

get_type(*trait*)

inputs
   OrderedDict of Inputs connected to the Interface. The format should be {input_id: InputSpec}.

module = <module 'nipypeinterface' from '/home/docs/checkouts/readthedocs.org/user_bui

outputs
   OrderedDict of Output connected to the Interface. The format should be {output_id: OutputSpec}.

classmethod test()
   Test the plugin, interfaces do not need to be tested on import

class fastr.plugins.**Null**
   Bases: *[fastr.core.ioplugin.IOPlugin](#)*

   The Null plugin is create to handle null:// type or URLs. These URLs are indicating the sink should not do anything. The data is not written to anywhere. Besides the scheme, the rest of the URL is ignored.

   __abstractmethods__ = frozenset()

   __init__()
      Initialization for the IOPlugin

      > **Returns** newly created IOPlugin

   __module__ = 'fastr.plugins'

   filename = '/home/docs/checkouts/readthedocs.org/user_builds/fastr/envs/3.0.0/lib/pyth

   module = <module 'null' from '/home/docs/checkouts/readthedocs.org/user_builds/fastr/e

put_url(*inpath*, *outurl*)
>   Put the files to the external data store.

>   > **Parameters**

>   > >   • **inpath** – path of the local data

>   > >   • **outurl** – url to where to store the data, starts with `file://`

put_value(*value*, *outurl*)
>   Put the value in the external data store.

>   > **Parameters**

>   > >   • **value** – value to store

>   > >   • **outurl** – url to where to store the data, starts with `file://`

scheme = 'null'

class fastr.plugins.**PimReporter**
>   Bases: `fastr.plugins.reportingplugin.ReportingPlugin`

>   **SUPPORTED_APIS = {2: <class 'pimreporter.PimAPIv2'>}**

>   **__abstractmethods__ = frozenset()**

>   **__init__**()
>   >   The BasePlugin constructor.

>   >   > **Returns** the created plugin

>   >   > **Return type** BasePlugin

>   >   > **Raises** `FastrPluginNotLoaded` – if the plugin did not load correctly

>   **__module__ = 'fastr.plugins'**

>   **activate**()
>   >   Activate the reporting plugin

>   **configuration_fields = {'pim_batch_size': (<class 'int'>, 100, 'Maximum number of job**

>   **filename = '/home/docs/checkouts/readthedocs.org/user_builds/fastr/envs/3.0.0/lib/pyth**

>   **job_updated**(*job*)

>   **log_record_emitted**(*record*)

>   **module = <module 'pimreporter' from '/home/docs/checkouts/readthedocs.org/user_builds/**

>   **run_finished**(*run*)

>   **run_started**(*run*)

class fastr.plugins.**ProcessPoolExecution**(*finished_callback=None*, *cancelled_callback=None*, *nr_of_workers=None*)
>   Bases: `fastr.plugins.executionplugin.ExecutionPlugin`

>   A local execution plugin that uses multiprocessing to create a pool of worker processes. This allows fastr to execute jobs in parallel with true concurrency. The number of workers can be specified in the fastr configuration, but the default amount is the `number of cores - 1` with a minimum of `1`.

>   > **Warning:** The ProcessPoolExecution does not check memory requirements of jobs and running many workers might lead to memory starvation and thus an unresponsive system.

__abstractmethods__ = frozenset()

__init__(*finished_callback=None*, *cancelled_callback=None*, *nr_of_workers=None*)
   Setup the ExecutionPlugin

> **Parameters**
>
> • **finished_callback** – the callback to call after a job finished
>
> • **cancelled_callback** – the callback to call after a job cancelled
>
> **Returns** newly created ExecutionPlugin

__module__ = 'fastr.plugins'

cleanup()
   Method to call to clean up the ExecutionPlugin. This can be to clear temporary data, close connections, etc.

> **Parameters** **force** – force cleanup (e.g. kill instead of join a process)

configuration_fields = {'process_pool_worker_number': (<class 'int'>, 3, 'Number of w

filename = '/home/docs/checkouts/readthedocs.org/user_builds/fastr/envs/3.0.0/lib/pyth

job_finished_callback(*result*)
   Reciever for the callback, it will split the result tuple and call job_finished

> **Parameters** **result** (*tuple*) – return value of run_job

module = <module 'processpoolexecution' from '/home/docs/checkouts/readthedocs.org/use

classmethod test()
   Test the plugin, default behaviour is just to instantiate the plugin

class fastr.plugins.**RQExecution**(*finished_callback=None*, *cancelled_callback=None*)
   Bases: *fastr.plugins.executionplugin.ExecutionPlugin*

A execution plugin based on Redis Queue. Fastr will submit jobs to the redis queue and workers will peel the jobs from the queue and process them.

This system requires a running redis database and the database url has to be set in the fastr configuration.

Note: This execution plugin required the redis and rq packages to be installed before it can be loaded properly.

__abstractmethods__ = frozenset()

__init__(*finished_callback=None*, *cancelled_callback=None*)
   Setup the ExecutionPlugin

> **Parameters**
>
> • **finished_callback** – the callback to call after a job finished
>
> • **cancelled_callback** – the callback to call after a job cancelled
>
> **Returns** newly created ExecutionPlugin

__module__ = 'fastr.plugins'

check_finished()

cleanup()
>   Method to call to clean up the ExecutionPlugin. This can be to clear temporary data, close connections, etc.

>>   Parameters **force** – force cleanup (e.g. kill instead of join a process)

configuration_fields = {'rq_host': (<class 'str'>, 'redis://localhost:6379/0', 'The u

filename = '/home/docs/checkouts/readthedocs.org/user_builds/fastr/envs/3.0.0/lib/pyth

module = <module 'rqexecution' from '/home/docs/checkouts/readthedocs.org/user_builds/

classmethod run_job(*job_id*, *job_command*, *job_stdout*, *job_stderr*)

classmethod test()
>   Test the plugin, default behaviour is just to instantiate the plugin

class fastr.plugins.**Reference**
>   Bases: *fastr.core.ioplugin.IOPlugin*

>   The Reference plugin is create to handle ref:// type or URLs. These URLs are to make the sink just write a simple reference file to the data. The reference file contains the DataType and the value so the result can be reconstructed. It for files just leaves the data on disk by reference. This plugin is not useful for production, but is used for testing purposes.

>   **__abstractmethods__ = frozenset()**

>   **__init__**()
>>   Initialization for the IOPlugin

>>>   Returns newly created IOPlugin

>   **__module__ = 'fastr.plugins'**

>   **filename = '/home/docs/checkouts/readthedocs.org/user_builds/fastr/envs/3.0.0/lib/pyth**

>   **module = <module 'reference' from '/home/docs/checkouts/readthedocs.org/user_builds/fa**

>   **push_sink_data**(*value*, *outurl*, *datatype=None*)
>>   Write out the sink data from the inpath to the outurl.

>>   **Parameters**

>>>   • **value** (*str*) – the path of the data to be pushed

>>>   • **outurl** (*str*) – the url to write the data to

>>>   • **datatype** (*DataType*) – the datatype of the data, used for determining the total contents of the transfer

>>   Returns None

>   **scheme = 'ref'**

class fastr.plugins.**S3Filesystem**
>   Bases: *fastr.core.ioplugin.IOPlugin*

> ---
> **Warning:** As this IOPlugin is under development, it has not been thoroughly tested.
>
> example url: s3://bucket.server/path/to/resource
> ---

>   **__abstractmethods__ = frozenset()**

>   **__init__**()
>>   Initialization for the IOPlugin

Returns newly created IOPlugin

__module__ = 'fastr.plugins'

cleanup()
> (abstract) Clean up the IOPlugin. This is to do things like closing files or connections. Will be called when the plugin is no longer required.

expand_url(*url*)
> Expand an S3 URL. This allows a source to collect multiple samples from a single url.

> > **Parameters url** (*str*) – url to expand

> > **Returns** the resulting url(s), a tuple if multiple, otherwise a str

> > **Return type** str or tuple of str

fetch_url(*inurl*, *outpath*)
> Get the file(s) or values from s3.

> > **Parameters**

> > > • **inurl** – url to the item in the data store

> > > • **outpath** – path where to store the fetch data locally

fetch_value(*inurl*)
> Fetch a value from S3

> > **Parameters inurl** – url of the value to read

> > **Returns** the fetched value

filename = '/home/docs/checkouts/readthedocs.org/user_builds/fastr/envs/3.0.0/lib/pyth

module = <module 's3filesystem' from '/home/docs/checkouts/readthedocs.org/user_builds

put_url(*inpath*, *outurl*)
> Upload the files to the S3 storage

> > **Parameters**

> > > • **inpath** – path to the local data

> > > • **outurl** – url to where to store the data in the external data store.

put_value(*value*, *outurl*)
> Put the value in S3

> > **Parameters**

> > > • **value** – value to store

> > > • **outurl** – url to where to store the data, starts with `file://`

scheme = ('s3', 's3list')

classmethod test()
> Test the plugin, default behaviour is just to instantiate the plugin

class fastr.plugins.**SimpleReport**
> Bases: *fastr.plugins.reportingplugin.ReportingPlugin*

__abstractmethods__ = frozenset()

__module__ = 'fastr.plugins'

filename = '/home/docs/checkouts/readthedocs.org/user_builds/fastr/envs/3.0.0/lib/pyth

module = <module 'simplereport' from '/home/docs/checkouts/readthedocs.org/user_builds

run_finished(*run*)

class fastr.plugins.**SingularityTarget**(*binary*, *container*, *interpreter=None*)
   Bases: *fastr.core.target.SubprocessBasedTarget*

   A tool target that is run using a singularity container, see the singulary website

   • binary (required): the name of the binary/script to call, can also be called bin for backwards
     compatibility.

   • **container (required): the singularity container to run, this can be in url form for singularity**
        pull or as a path to a local container

   • interpreter: the interpreter to use to call the binary e.g. python

   SINGULARITY_BIN = 'singularity'

   __abstractmethods__ = frozenset()

   __enter__()
        Set the environment in such a way that the target will be on the path.

   __exit__(*exc_type*, *exc_value*, *traceback*)
        Cleanup the environment

   __init__(*binary*, *container*, *interpreter=None*)
        Define a new local binary target. Must be defined either using paths and optionally environment_variables
        and initscripts, or enviroment modules.

   __module__ = 'fastr.plugins'

   filename = '/home/docs/checkouts/readthedocs.org/user_builds/fastr/envs/3.0.0/lib/pytho

   module = <module 'singularitytarget' from '/home/docs/checkouts/readthedocs.org/user_b

   run_command(*command*)
        Run a command with the target

   classmethod test()
        Test if singularity is availble on the path

class fastr.plugins.**SlurmExecution**(*finished_callback=None*, *cancelled_callback=None*)
   Bases: *fastr.plugins.executionplugin.ExecutionPlugin*

   SBATCH = 'sbatch'

   SCANCEL = 'scancel'

   SCONTROL = 'scontrol'

   SQUEUE = 'squeue'

   SQUEUE_FORMAT = '{"id": %.18i, "status": "%.2t"}'

   STATUS_MAPPING = {' F': <JobState.failed: ('failed', 'done', True)>, ' R': <JobState.

   SUPPORTS_CANCEL = True

   SUPPORTS_DEPENDENCY = True

   SUPPORTS_HOLD_RELEASE = True

   __abstractmethods__ = frozenset()

   __init__(*finished_callback=None*, *cancelled_callback=None*)
        Setup the ExecutionPlugin

> **Parameters**
>
> - **finished_callback** – the callback to call after a job finished
>
> - **cancelled_callback** – the callback to call after a job cancelled
>
> **Returns** newly created ExecutionPlugin

__module__ = 'fastr.plugins'

cleanup()
> Method to call to clean up the ExecutionPlugin. This can be to clear temporary data, close connections, etc.
>
> > **Parameters force** – force cleanup (e.g. kill instead of join a process)

configuration_fields = {'slurm_job_check_interval': (<class 'int'>, 30, 'The interval

filename = '/home/docs/checkouts/readthedocs.org/user_builds/fastr/envs/3.0.0/lib/pytho

job_status_check()

module = <module 'slurmexecution' from '/home/docs/checkouts/readthedocs.org/user_buil

classmethod test()
> Test the plugin, default behaviour is just to instantiate the plugin

class fastr.plugins.**StrongrExecution**(*finished_callback=None*, *cancelled_callback=None*)
> Bases: *fastr.plugins.executionplugin.ExecutionPlugin*

A execution plugin based on Redis Queue. Fastr will submit jobs to the redis queue and workers will peel the jobs from the queue and process them.

This system requires a running redis database and the database url has to be set in the fastr configuration.

Note: This execution plugin required the redis and rq packages to be installed before it can be loaded properly.

__abstractmethods__ = frozenset()

__init__(*finished_callback=None*, *cancelled_callback=None*)
> Setup the ExecutionPlugin

> **Parameters**
>
> - **finished_callback** – the callback to call after a job finished
>
> - **cancelled_callback** – the callback to call after a job cancelled
>
> **Returns** newly created ExecutionPlugin

__module__ = 'fastr.plugins'

check_finished()

cleanup()
> Method to call to clean up the ExecutionPlugin. This can be to clear temporary data, close connections, etc.
>
> > **Parameters force** – force cleanup (e.g. kill instead of join a process)

configuration_fields = {}

filename = '/home/docs/checkouts/readthedocs.org/user_builds/fastr/envs/3.0.0/lib/pytho

module = <module 'strongrexecution' from '/home/docs/checkouts/readthedocs.org/user_bu

---

classmethod test()
> Test the plugin, default behaviour is just to instantiate the plugin

class fastr.plugins.**VirtualFileSystem**
> Bases: *fastr.core.vfs.VirtualFileSystem*, *fastr.core.ioplugin.IOPlugin*

The virtual file system class. This is an IOPlugin, but also heavily used internally in fastr for working with directories. The VirtualFileSystem uses the `vfs://` url scheme.

A typical virtual filesystem url is formatted as `vfs://mountpoint/relative/dir/from/mount.ext`

Where the `mountpoint` is defined in the *Config file*. A list of the currently known mountpoints can be found in the `fastr.config` object

```
>>> fastr.config.mounts
{'example_data': '/home/username/fastr-feature-documentation/fastr/fastr/examples/
↪data',
 'home': '/home/username/',
 'tmp': '/home/username/FastrTemp'}
```

This shows that a url with the mount `home` such as `vfs://home/tempdir/testfile.txt` would be translated into `/home/username/tempdir/testfile.txt`.

There are a few default mount points defined by Fastr (that can be changed via the config file).

mountpoint	default location
home	the users home directory (`expanduser('~/')`)
tmp	the fastr temprorary dir, defaults to `tempfile.gettempdir()`
example_data	the fastr example data directory, defaults `$FASTRDIR/example/data`

__abstractmethods__ = frozenset()

__module__ = 'fastr.plugins'

filename = '/home/docs/checkouts/readthedocs.org/user_builds/fastr/envs/3.0.0/lib/pyth

module = <module 'virtualfilesystem' from '/home/docs/checkouts/readthedocs.org/user_b

scheme = 'vfs'

class fastr.plugins.**VirtualFileSystemRegularExpression**
> Bases: *fastr.core.ioplugin.IOPlugin*

The VirtualFileSystemValueList an expand-only type of IOPlugin. No URLs can actually be fetched, but it can expand a single URL into a larger amount of URLs.

A `vfsregex://` URL is a vfs URL that can contain regular expressions on every level of the path. The regular expressions follow the `re module` definitions.

An example of a valid URLs would be:

```
vfsregex://tmp/network_dir/.*/.*/__fastr_result__.pickle.gz
vfsregex://tmp/network_dir/nodeX/(?P<id>.*)/__fastr_result__.pickle.gz
```

The first URL would result in all the `__fastr_result__.pickle.gz` in the working directory of a Network. The second URL would only result in the file for a specific node (nodeX), but by adding the named group id using `(?P<id>.*)` the sample id of the data is automatically set to that group (see Regular Expression Syntax under the special characters for more info on named groups in regular expression).

Concretely if we would have a directory `vfs://mount/somedir` containing:

```
image_1/Image.nii
image_2/image.nii
image_3/anotherimage.nii
image_5/inconsistentnamingftw.nii
```

we could match these files using `vfsregex://mount/somedir/(?P<id>image_\d+)/.*\.nii` which would result in the following source data after expanding the URL:

```
{'image_1': 'vfs://mount/somedir/image_1/Image.nii',
 'image_2': 'vfs://mount/somedir/image_2/image.nii',
 'image_3': 'vfs://mount/somedir/image_3/anotherimage.nii',
 'image_5': 'vfs://mount/somedir/image_5/inconsistentnamingftw.nii'}
```

Showing the power of this regular expression filtering. Also it shows how the ID group from the URL can be used to have sensible sample ids.

> **Warning:** due to the nature of regexp on multiple levels, this method can be slow when having many matches on the lower level of the path (because the tree of potential matches grows) or when directories that are parts of the path are very large.

`__abstractmethods__ = frozenset()`

`__init__`()
> Initialization for the IOPlugin

> > **Returns** newly created IOPlugin

`__module__ = 'fastr.plugins'`

`expand_url`(*url*)
> (abstract) Expand an URL. This allows a source to collect multiple samples from a single url. The URL will have a wildcard or point to something with info and multiple urls will be returned.

> > **Parameters** **url** ([*str*](#)) – url to expand

> > **Returns** the resulting url(s), a tuple if multiple, otherwise a str

> > **Return type** [str](#) or tuple of str

`filename = '/home/docs/checkouts/readthedocs.org/user_builds/fastr/envs/3.0.0/lib/pyth`

`module = <module 'virtualfilesystemregularexpression' from '/home/docs/checkouts/readt`

`scheme = 'vfsregex'`

class fastr.plugins.**VirtualFileSystemValueList**
> Bases: [*fastr.core.ioplugin.IOPlugin*](#)

The VirtualFileSystemValueList an expand-only type of IOPlugin. No URLs can actually be fetched, but it can expand a single URL into a larger amount of URLs. A `vfslist://` URL basically is a url that points to a file using vfs. This file then contains a number lines each containing another URL.

If the contents of a file `vfs://mount/some/path/contents` would be:

```
vfs://mount/some/path/file1.txt
vfs://mount/some/path/file2.txt
vfs://mount/some/path/file3.txt
vfs://mount/some/path/file4.txt
```

Then using the URL `vfslist://mount/some/path/contents` as source data would result in the four files being pulled.

Note: The URLs in a vfslist file do not have to use the `vfs` scheme, but can use any scheme known to the Fastr system.

`__abstractmethods__ = frozenset()`

`__init__`()
> Initialization for the IOPlugin
>
> > **Returns** newly created IOPlugin

`__module__ = 'fastr.plugins'`

`expand_url`(*url*)
> (`abstract`) Expand an URL. This allows a source to collect multiple samples from a single url. The URL will have a wildcard or point to something with info and multiple urls will be returned.
>
> > **Parameters** **url** (*str*) – url to expand
> >
> > **Returns** the resulting url(s), a tuple if multiple, otherwise a str
> >
> > **Return type** str or tuple of str

`filename = '/home/docs/checkouts/readthedocs.org/user_builds/fastr/envs/3.0.0/lib/pyth`

`module = <module 'virtualfilesystemvaluelist' from '/home/docs/checkouts/readthedocs.o`

`scheme = 'vfslist'`

class fastr.plugins.**XNATStorage**
> Bases: *fastr.core.ioplugin.IOPlugin*

Warning: As this IOPlugin is under development, it has not been thoroughly tested.

The XNATStorage plugin is an IOPlugin that can download data from and upload data to an XNAT server. It uses its own `xnat://` URL scheme. This is a scheme specific for this plugin and though it looks somewhat like the XNAT rest interface, a different type or URL.

Data resources can be access directly by a data url:

```
xnat://xnat.example.com/data/archive/projects/sandbox/subjects/subject001/
↪experiments/experiment001/scans/T1/resources/DICOM
xnat://xnat.example.com/data/archive/projects/sandbox/subjects/subject001/
↪experiments/*_BRAIN/scans/T1/resources/DICOM
```

In the second URL you can see a wildcard being used. This is possible at long as it resolves to exactly one item.

The `id` query element will change the field from the default experiment to subject and the `label` query element sets the use of the label as the fastr id (instead of the XNAT id) to `True` (the default is `False`)

To disable `https` transport and use `http` instead the query string can be modified to add `insecure=true`. This will make the plugin send requests over `http`:

```
xnat://xnat.example.com/data/archive/projects/sandbox/subjects/subject001/
↪experiments/*_BRAIN/scans/T1/resources/DICOM?insecure=true
```

For sinks it is import to know where to save the data. Sometimes you want to save data in a new assessor/resource and it needs to be created. To allow the Fastr sink to create an object in XNAT, you have to supply the type as a query parameter:

```
xnat://xnat.bmia.nl/data/archive/projects/sandbox/subjects/S01/experiments/_BRAIN/
↪assessors/test_assessor/resources/IMAGE/files/image.nii.gz?resource_
↪type=xnat:resourceCatalog&assessor_type=xnat:qcAssessmentData
```

Valid options are: subject_type, experiment_type, assessor_type, scan_type, and resource_type.

If you want to do a search where multiple resources are returned, it is possible to use a search url:

```
xnat://xnat.example.com/search?projects=sandbox&subjects=subject[0-9][0-9][0-9]&
↪experiments=*_BRAIN&scans=T1&resources=DICOM
```

This will return all DICOMs for the T1 scans for experiments that end with _BRAIN that belong to a subjectXXX where XXX is a 3 digit number. By default the ID for the samples will be the experiment XNAT ID (e.g. XNAT_E00123). The wildcards that can be the used are the same UNIX shell-style wildcards as provided by the module `fnmatch`.

It is possible to change the id to a different fields id or label. Valid fields are project, subject, experiment, scan, and resource:

```
xnat://xnat.example.com/search?projects=sandbox&subjects=subject[0-9][0-9][0-9]&
↪experiments=*_BRAIN&scans=T1&resources=DICOM&id=subject&label=true
```

The following variables can be set in the search query:

variable	default	usage
projects	`*`	The project(s) to select, can contain wildcards (see `fnmatch`)
subjects	`*`	The subject(s) to select, can contain wildcards (see `fnmatch`)
experiments	`*`	The experiment(s) to select, can contain wildcards (see `fnmatch`)
scans	`*`	The scan(s) to select, can contain wildcards (see `fnmatch`)
resources	`*`	The resource(s) to select, can contain wildcards (see `fnmatch`)
id	`experiment`	What field to use a the id, can be: project, subject, experiment, scan, or resource
label	`false`	Indicate the XNAT label should be used as fastr id, options `true` or `false`
insecure	`false`	Change the url scheme to be used to http instead of https
verify	`true`	(Dis)able the verification of SSL certificates
regex	`false`	Change search to use regex `re.match()` instead of fnmatch for matching
overwrite	`false`	Tell XNAT to overwrite existing files if a file with the name is already present

For storing credentials the `.netrc` file can be used. This is a common way to store credentials on UNIX systems. It is required that the file is only accessible by the owner only or a `NetrcParseError` will be raised. A netrc file is really easy to create, as its entries look like:

```
machine xnat.example.com
        login username
        password secret123
```

See the `netrc module` or the GNU inet utils website for more information about the `.netrc` file.

Note: On windows the location of the netrc file is assumed to be `os.path.expanduser('~/_netrc')`. The leading underscore is because windows does not like filename starting with a dot.

Note: For scan the label will be the scan type (this is initially the same as the series description, but can be updated manually or the XNAT scan type cleanup).

Warning: labels in XNAT are not guaranteed to be unique, so be careful when using them as the sample ID.

For background on XNAT, see the XNAT API DIRECTORY for the REST API of XNAT.

__abstractmethods__ = frozenset()

__init__()
    Initialization for the IOPlugin

        Returns newly created IOPlugin

__module__ = 'fastr.plugins'

cleanup()
    (abstract) Clean up the IOPlugin. This is to do things like closing files or connections. Will be called when the plugin is no longer required.

connect(*server*, *path=''*, *insecure=False*, *verify=True*)

expand_url(*url*)
    (abstract) Expand an URL. This allows a source to collect multiple samples from a single url. The URL will have a wildcard or point to something with info and multiple urls will be returned.

        Parameters url (*str*) – url to expand

        Returns the resulting url(s), a tuple if multiple, otherwise a str

        Return type str or tuple of str

fetch_url(*inurl*, *outpath*)
    Get the file(s) or values from XNAT.

        Parameters

            • **inurl** – url to the item in the data store

            • **outpath** – path where to store the fetch data locally

filename = '/home/docs/checkouts/readthedocs.org/user_builds/fastr/envs/3.0.0/lib/pyth

module = <module 'xnatstorage' from '/home/docs/checkouts/readthedocs.org/user_builds/

parse_uri(*url*)

put_url(*inpath*, *outurl*)
    Upload the files to the XNAT storage

        Parameters

            • **inpath** – path to the local data

            • **outurl** – url to where to store the data in the external data store.

scheme = ('xnat', 'xnat+http', 'xnat+https')

server

static upload(*resource*, *in_path*, *location*, *retries=3*, *overwrite=False*)

> **xnat**

fastr.plugins.**json**
> alias of `fastr.plugins.JsonCollector`

fastr.plugins.**path**
> alias of `fastr.plugins.PathCollector`

fastr.plugins.**stdout**
> alias of `fastr.plugins.StdoutCollector`

executionplugin Module

class fastr.plugins.executionplugin.**ExecutionPlugin**(*finished_callback=None*, *cancelled_callback=None*)
> Bases: `fastr.abc.baseplugin.Plugin`

This class is the base for all Plugins to execute jobs somewhere. There are many methods already in place for taking care of stuff.

There are fall-backs for certain features, but if a system already implements those it is usually preferred to skip the fall-back and let the external system handle it. There are a few flags to enable disable these features:

- `cls.SUPPORTS_CANCEL` indicates that the plugin can cancel queued jobs

- `cls.SUPPORTS_HOLD_RELEASE` indicates that the plugin can queue jobs in a hold state and can release them again (if not, the base plugin will create a hidden queue for held jobs). The plugin should respect the `Job.status == JobState.hold` when queueing jobs.

- `cls.SUPPORTS_DEPENDENCY` indicate that the plugin can manage job dependencies, if not the base plugin job dependency system will be used and jobs with only be submitted when all dependencies are met.

- `cls.CANCELS_DEPENDENCIES` indicates that if a job is cancelled it will automatically cancel all jobs depending on that job. If not the plugin traverse the dependency graph and kill each job manual.

Note: If a plugin supports dependencies it is assumed that when a job gets cancelled, the depending job also get cancelled automatically!

Most plugins should only need to redefine a few abstract methods:

- `__init__` the constructor

- `cleanup` a clean up function that frees resources, closes connections, etc

- `_queue_job` the method that queues the job for execution

Optionally an extra job finished callback could be added:

- `_job_finished` extra callback for when a job finishes

If `SUPPORTS_CANCEL` is set to True, the plugin should also implement:

- `_cancel_job` cancels a previously queued job

If `SUPPORTS_HOLD_RELEASE` is set to True, the plugin should also implement:

- `_hold_job` hold_job a job that is currently held

- `_release_job` releases a job that is currently held

If `SUPPORTED_DEPENDENCY` is set to True, the plugin should:

- Make sure to use the `Job.hold_jobs` as a list of its dependencies

Not all of the functions need to actually do anything for a plugin. There are examples of plugins that do not really need a `cleanup`, but for safety you need to implement it. Just using a `pass` for the method could be fine in such a case.

> **Warning:** When overwriting other functions, extreme care must be taken not to break the plugins working, as there is a lot of bookkeeping that can go wrong.

CANCELS_DEPENDENCIES = False
> Indicates that when a job is cancelled the dependencies

SUPPORTS_CANCEL = False
> Indicates if the plugin can cancel queued jobs

SUPPORTS_DEPENDENCY = False
> Indicate if the plugin can manage job dependencies, if not the base plugin job dependency system will be used and jobs with only be submitted when all dependencies are met.

SUPPORTS_HOLD_RELEASE = False
> Indicates if the plugin can queue jobs in a hold state and can release them again (if not, the base plugin will create a hidden queue for held
>
> > jobs)

__abstractmethods__ = frozenset({'_queue_job', '__init__', 'cleanup'})

__del__()
> Cleanup if the variable was deleted on purpose

__enter__()

__exit__(*type_*, *value*, *tb*)

__init__(*finished_callback=None*, *cancelled_callback=None*)
> Setup the ExecutionPlugin

> > **Parameters**
> >
> > - **finished_callback** – the callback to call after a job finished
> >
> > - **cancelled_callback** – the callback to call after a job cancelled
> >
> > **Returns** newly created ExecutionPlugin

__module__ = 'fastr.plugins.executionplugin'

cancel_job(*job*)
> Cancel a job previously queued

> > **Parameters job** – job to cancel

check_job_requirements(*job_id*)
> Check if the requirements for a job are fulfilled.

> > **Parameters job_id** – job to check

> > **Returns** directive what should happen with the job

> > **Return type** *JobAction*

check_job_status(*job_id*)
> Get the status of a specified job

> **Parameters** `job_id` – the target job
>
> **Returns** the status of the job (or None if job not found)

clean_free_jobs(*job*)

cleanup()
> Method to call to clean up the ExecutionPlugin. This can be to clear temporary data, close connections, etc.
>
> > **Parameters** `force` – force cleanup (e.g. kill instead of join a process)

get_job(*job_id*)

get_status(*job*)

hold_job(*job*)

job_finished(*job*, *errors=None*, *blocking=False*)
> The default callback that is called when a Job finishes. This will create a new thread that handles the actual callback.
>
> > **Parameters**
> >
> > - **job** (`Job`) – the job that finished
> >
> > - **errors** – optional list of errors encountered
> >
> > - **blocking** (`bool`) – if blocking, do not create threads
> >
> > **Returns**

process_callbacks()

queue_job(*job*)
> Add a job to the execution queue
>
> > **Parameters** `job` (`Job`) – job to add

register_job(*job*)

release_job(*job*)
> Release a job that has been put on hold
>
> > **Parameters** `job` – job to release

show_jobs(*req_status=None*)
> List the queued jobs, possible filtered by status
>
> > **Parameters** `req_status` – requested status to filter on
> >
> > **Returns** list of jobs

signal_dependent_jobs(*job_id*)
> Check all depedent jobs and process them if all their dependencies are met. :param job_id: :return:

class `fastr.plugins.executionplugin.`**JobAction**
> Bases: `enum.Enum`

Job actions that can be performed. This is used for checking if held jobs should be queued, held longer or be cancelled.

__module__ = **'fastr.plugins.executionplugin'**

cancel = **'cancel'**

hold = **'hold'**

```
queue = 'queue'
```

reportingplugin Module

class fastr.plugins.reportingplugin.**ReportingPlugin**

    Bases: fastr.abc.baseplugin.Plugin

    Base class for all reporting plugins. The plugin has a number of methods that can be implemented that will be called on certain events. On these events the plugin can inspect the presented data and take reporting actions.

    __abstractmethods__ = frozenset()

    __module__ = 'fastr.plugins.reportingplugin'

    activate()

    deactivate()

    job_updated(*job*)

    log_record_emitted(*record*)

    run_finished(*run*)

    run_started(*run*)

Subpackages

managers Package

managers Package

executionpluginmanager Module

This module holds the ExecutionPluginManager as well as the base-class for all ExecutionPlugins.

class fastr.plugins.managers.executionpluginmanager.**ExecutionPluginManager**(*parent*)

    Bases: *fastr.plugins.managers.pluginmanager.PluginSubManager*

    Container holding all the ExecutionPlugins known to the Fastr system

    __abstractmethods__ = frozenset()

    __init__(*parent*)

        Initialize a ExecutionPluginManager and load plugins.

            Parameters

                • **path** – path to search for plugins

                • **recursive** – flag for searching recursively

            Returns newly created ExecutionPluginManager

    __module__ = 'fastr.plugins.managers.executionpluginmanager'

interfacemanager Module

This module holds the ExecutionPluginManager as well as the base-class for all ExecutionPlugins.

class fastr.plugins.managers.interfacemanager.**InterfacePluginManager**(*parent*)

   Bases: *fastr.plugins.managers.pluginmanager.PluginSubManager*

   Container holding all the CollectorPlugins

   __abstractmethods__ = frozenset()

   __init__(*parent*)
      Create the Coll :param path: :param recursive: :return:

   __module__ = 'fastr.plugins.managers.interfacemanager'

iopluginmanager Module

class fastr.plugins.managers.iopluginmanager.**IOPluginManager**(*parent*)

   Bases: *fastr.plugins.managers.pluginmanager.PluginSubManager*

   A mapping containing the IOPlugins known to this system

   __abstractmethods__ = frozenset()

   __init__(*parent*)
      Create the IOPluginManager and populate it.

         Returns newly created IOPluginManager

   __iter__()
      Get an iterator from the BaseManager. The iterator will iterate over the keys of the BaseManager.

         Returns the iterator

         Return type dictionary-keyiterator

   __keytransform__(*key*)
      Identity transform for the keys. This function can be reimplemented by a subclass to implement a different key transform.

         Parameters **key** – key to transform

         Returns the transformed key (in this case the same key as inputted)

   __module__ = 'fastr.plugins.managers.iopluginmanager'

   cleanup()
      Cleanup all plugins, this closes files, connections and other things that could be left dangling otherwise.

   static create_ioplugin_tool(*tools*, *interfaces*)
      Create the tools which handles sinks and sources. The command of this tool is the main of core.ioplugin.

   expand_url(*url*)
      Expand the url by filling the wildcards. This function checks the url scheme and uses the expand function of the correct IOPlugin.

         Parameters **url** (*str*) – url to expand

         Returns list of urls

         Return type list of str

pull_source_data (*url*, *outdir*, *sample_id*, *datatype=None*)

    Retrieve data from an external source. This function checks the url scheme and selects the correct IOPlugin to retrieve the data.

        Parameters

- **url** – url to pull

- **outdir** (`str`) – the directory to write the data to

- **datatype** (`DataType`) – the datatype of the data, used for determining the total contents of the transfer

        Returns None

push_sink_data (*inpath*, *outurl*, *datatype=None*)

    Send data to an external source. This function checks the url scheme and selects the correct IOPlugin to retrieve the data.

        Parameters

- **inpath** (`str`) – the path of the data to be pushed

- **outurl** (`str`) – the url to write the data to

- **datatype** (`DataType`) – the datatype of the data, used for determining the total contents of the transfer

put_url (*inpath*, *outurl*)

    Put the files to the external data store.

        Parameters

- **inpath** – path to the local data

- **outurl** – url to where to store the data in the external data store.

static register_url_scheme (*scheme*)

    Register a custom scheme to behave http like. This is needed to parse all things properly with urlparse.

        Parameters **scheme** – the scheme to register

url_to_path (*url*)

    Retrieve the path for a given url

        Parameters **url** (`str`) – the url to parse

        Returns the path corresponding to the input url

        Return type str

networkmanager Module

This module contains the tool manager class

class fastr.plugins.managers.networkmanager.**NetworkManager** (*path*)

    Bases: *fastr.plugins.managers.objectmanager.ObjectManager*

    __abstractmethods__ = frozenset()

    __module__ = 'fastr.plugins.managers.networkmanager'

    get_object_version (*obj*)

        Get the version of a given object

> **Parameters object** – the object to use
>
> **Returns** the version of the object

object_class
> The class of the objects to populate the manager with

objectmanager Module

This module contains the object manager class

class fastr.plugins.managers.objectmanager.**ObjectManager**(*path*)
> Bases: fastr.abc.basemanager.BaseManager
>
> Class for managing all the objects loaded in the fastr system
>
> **__abstractmethods__ = frozenset({'get_object_version', 'object_class'})**
>
> **__contains__**(*key*)
> > Check if an item is in the ObjectManager
> >
> > **Parameters key** (*str or tuple*) – object id or tuple (Objectid, version)
> >
> > **Returns** flag indicating the item is in the manager
>
> **__getitem__**(*key*)
> > Retrieve a Object from the ObjectManager. You can request by only an id, which results in the newest version of the Object being returned, or request using both an id and a version.
> >
> > **Parameters key** (*str or tuple*) – object id or tuple (Objectid, version)
> >
> > **Returns** the requested Object
> >
> > **Raises** *FastrObjectUnknownError* – if a non-existing Object was requested
>
> **__init__**(*path*)
> > Create a ObjectManager and scan path to search for Objects
> >
> > **Parameters path** (*str or iterable of str*) – the path(s) to scan for Objects
> >
> > **Returns** newly created ObjectManager
>
> **__keytransform__**(*key*)
> > Key transform, used for allowing indexing both by id-only and by (id, version)
> >
> > **Parameters key** – key to transform
> >
> > **Returns** key in form (id, version)
>
> **__module__ = 'fastr.plugins.managers.objectmanager'**
>
> **get_object_version**(*obj*)
> > Get the version of a given object
> >
> > **Parameters object** – the object to use
> >
> > **Returns** the version of the object
>
> **object_class**
> > The class of the objects to populate the manager with
>
> **objectversions**(*obj*)
> > Return a list of available versions for the object
> >
> > **Parameters object** – The object to check the versions for. Can be either a *Object* or a *str*.

> **Returns** List of version objects. Returns *None* when the given object is not known.

todict()
> Return a dictionary version of the Manager
>
> > **Returns** manager as a dict

pluginmanager Module

This module contains the Manager class for Plugins in the fastr system

class fastr.plugins.managers.pluginmanager.**PluginManager**(*path=None*)
> Bases: fastr.abc.basepluginmanager.BasePluginManager

> **__abstractmethods__ = frozenset()**

> **__init__**(*path=None*)
> > Create a BasePluginManager and scan the give path for matching plugins
> >
> > **Parameters**
> >
> > - **path** (*str*) – path to scan
> >
> > - **recursive** (*bool*) – flag to indicate a recursive search
> >
> > - **module** (*module*) – the module to register plugins into
> >
> > **Returns** newly created plugin manager
> >
> > **Raises** *FastrTypeError* – if self._plugin_class is set to a class not subclassing BasePlugin

> **__module__ = 'fastr.plugins.managers.pluginmanager'**

> **__setitem__**(*key*, *value*)
> > Store an item in the BaseManager, will ignore the item if the key is already present in the BaseManager.
> >
> > **Parameters**
> >
> > - **name** – the key of the item to save
> >
> > - **value** – the value of the item to save
> >
> > **Returns** None

> **plugin_class**
> > The plugin manager contains any Plugin subclass

class fastr.plugins.managers.pluginmanager.**PluginSubManager**(*parent*, *plugin_class*)
> Bases: fastr.abc.basepluginmanager.BasePluginManager

> A PluginManager that is a selection of a parent plugin manger. It uses the PluginsView to only expose part of the parent PluginManager. This is used to create plugin plugins.managers for only certain types of plugins (e.g. IOPlugins) without loading them multiple times.

> **__abstractmethods__ = frozenset()**

> **__init__**(*parent*, *plugin_class*)
> > Create a BasePluginManager and scan the give path for matching plugins
> >
> > **Parameters**
> >
> > - **path** (*str*) – path to scan
> >
> > - **recursive** (*bool*) – flag to indicate a recursive search

- **module** (`module`) – the module to register plugins into

Returns newly created plugin manager

Raises [**FastrTypeError**](#) – if self._plugin_class is set to a class not subclassing BasePlugin

__module__ = 'fastr.plugins.managers.pluginmanager'

data
> The actual data dict underlying this Manager

plugin_class
> PluginSubManagers only expose the plugins of a certain class

class fastr.plugins.managers.pluginmanager.**PluginsView**(*parent*, *plugin_class*)
> Bases: [collections.abc.MutableMapping](#)

A collection that acts like view of the plugins of another plugin manager. This is a proxy object that only gives access the plugins of a certain plugin class. It behaves like a mapping and is used as the data object for a PluginSubManager.

__abstractmethods__ = frozenset()

__delitem__(*key*)

__dict__ = mappingproxy({'__module__': 'fastr.plugins.managers.pluginmanager', '__doc

__getitem__(*item*)

__init__(*parent*, *plugin_class*)
> Constructor for the plugins view

> Parameters

>> - **parent** (`BasePluginManager`) – the parent plugin manager

>> - **plugin_class** (`class`) – the class of the plugins to expose

__iter__()

__len__()

__module__ = 'fastr.plugins.managers.pluginmanager'

__setitem__(*key*, *value*)

__weakref__
> list of weak references to the object (if defined)

filter_plugin(*plugin*)

targetmanager Module

This module holds the ExecutionPluginManager as well as the base-class for all ExecutionPlugins.

class fastr.plugins.managers.targetmanager.**TargetManager**(*parent*)
> Bases: [fastr.plugins.managers.pluginmanager.PluginSubManager](#)

Container holding all the ExecutionPlugins known to the Fastr system

__abstractmethods__ = frozenset()

__init__(*parent*)
> Initialize a ExecutionPluginManager and load plugins.

> Returns newly created ExecutionPluginManager

```
__module__ = 'fastr.plugins.managers.targetmanager'
```

`toolmanager` Module

This module contains the tool manager class

class `fastr.plugins.managers.toolmanager.`**`ToolManager`**(*path*)

   Bases: *`fastr.plugins.managers.objectmanager.ObjectManager`*

   `__abstractmethods__` **= frozenset()**

   `__module__` **= 'fastr.plugins.managers.toolmanager'**

   `get_object_version`(*obj*)
      Get the version of a given object

         Parameters **`object`** – the object to use

         Returns the version of the object

   `object_class`
      The class of the objects to populate the manager with

   `populate`()
      Populate the manager with the data. This is a method that will be called when the Managers data is first
      accessed. This way we avoid doing expensive directory scans when the data is never requested.

   `toolversions`(*tool*)
      Return a list of available versions for the tool

         Parameters **`tool`** – The tool to check the versions for. Can be either a *Tool* or a *str*.

         Returns List of version objects. Returns *None* when the given tool is not known.

resources Package

resources Package

Subpackages

datatypes Package

datatypes Package

`AnalyzeImageFile` Module

class `fastr.resources.datatypes.AnalyzeImageFile.`**`AnalyzeImageFile`**(*value=None*,
                                                                                      *for-
                                                                                      mat_=None*)

   Bases: *`fastr.datatypes.URLType`*

   `__abstractmethods__` **= frozenset()**

   `__module__` **= 'fastr.resources.datatypes.AnalyzeImageFile'**

   classmethod **`content`**(*invalue*, *outvalue=None*)
      Give the contents of a URLType, this is generally useful for filetypes that consists of multiple files (e.g.

AnalyzeImageFile, DICOM). The value will indicate the main file, and the contents function can determine all files that form a single data value.

> **Parameters**
>
> - **inval** – a value to figure out contents for this type
> - **outval** – the place where the copy should point to
>
> **Returns** a list of all files part of the value (e.g. header and data file)
>
> **Return type** list

description = 'Analyze Image file formate'

extension = 'hdr'

Boolean Module

class fastr.resources.datatypes.Boolean.**Boolean**(*value=None*, *format_=None*)
    Bases: *fastr.datatypes.ValueType*

    Datatype representing a boolean

    __abstractmethods__ = frozenset()

    __module__ = 'fastr.resources.datatypes.Boolean'

    __str__()
        Returns the string version of the BaseDataType

> **Returns** string version
>
> **Return type** str

    description = 'A boolean value (True of False)'

    value
        The value of object instantiation of this DataType.

Directory Module

class fastr.resources.datatypes.Directory.**Directory**(*value=None*, *format_=None*)
    Bases: *fastr.datatypes.URLType*

    DataType representing a directory.

    __abstractmethods__ = frozenset()

    __eq__(*other*)
        Directories are equal by default as long as the validatity matches.

> **Parameters other** (Directory) – other to compare against
>
> **Returns** equality flag

    __hash__ = None

    __module__ = 'fastr.resources.datatypes.Directory'

    action(*name*)
        This method makes sure the Directory exists. A Tool can indicate an action that should be called for an Output which will be called before execution.

Parameters **name** (*str*) – name of the action to execute

Returns None

description = 'A directory on the disk'

extension = None

FilePrefix Module

class fastr.resources.datatypes.FilePrefix.**FilePrefix**(*value=None*, *format_=None*)
Bases: *fastr.datatypes.URLType*

__abstractmethods__ = frozenset()

__module__ = 'fastr.resources.datatypes.FilePrefix'

checksum()
Return the checksum of this URL type

Returns checksum string

Return type str

description = 'Prefix for another file, including the path'

extension = None

Float Module

class fastr.resources.datatypes.Float.**Float**(*value=None*, *format_=None*)
Bases: *fastr.datatypes.ValueType*

__abstractmethods__ = frozenset()

__module__ = 'fastr.resources.datatypes.Float'

description = 'A floating point value'

value
The value of object instantiation of this DataType.

ITKImageFile Module

class fastr.resources.datatypes.ITKImageFile.**ITKImageFile**(*value=None*)
Bases: *fastr.datatypes.TypeGroup*

__abstractmethods__ = frozenset()

__module__ = 'fastr.resources.datatypes.ITKImageFile'

description = 'Text file to store point coordinates'

Int Module

class fastr.resources.datatypes.Int.**Int**(*value=None*, *format_=None*)
Bases: *fastr.datatypes.ValueType*

__abstractmethods__ = frozenset()

> **__module__** **= 'fastr.resources.datatypes.Int'**

> **description = 'an integer value'**

> **value**
>> The value of object instantiation of this DataType.

`JsonFile` Module

class fastr.resources.datatypes.JsonFile.**JsonFile**(*value=None*, *format_=None*)
> Bases: *fastr.datatypes.URLType*

> **__abstractmethods__** **= frozenset()**

> **__module__** **= 'fastr.resources.datatypes.JsonFile'**

> **checksum**()
>> Return the checksum of this URL type

>>> **Returns** checksum string

>>> **Return type** str

> **description = 'json file'**

> **extension = 'json'**

`MetaImageFile` Module

class fastr.resources.datatypes.MetaImageFile.**MetaImageFile**(*value=None*, *format_=None*)
> Bases: *fastr.datatypes.URLType*

> **__abstractmethods__** **= frozenset()**

> **__eq__**(*other*)
>> Test the equality of two DataType objects

>>> **Parameters** **other** (URLType) – the object to compare against

>>> **Returns** flag indicating equality

>>> **Return type** bool

> **__hash__** **= None**

> **__module__** **= 'fastr.resources.datatypes.MetaImageFile'**

> **checksum**()
>> Return the checksum of this MetaImageFile

>>> **Returns** checksum string

>>> **Return type** str

> **classmethod content**(*invalue*, *outvalue=None*)
>> Give the contents of a URLType, this is generally useful for filetypes that consists of multiple files (e.g. AnalyzeImageFile, DICOM). The value will indicate the main file, and the contents function can determine all files that form a single data value.

>>> **Parameters**

>>>> • **inval** – a value to figure out contents for this type

- **outval** – the place where the copy should point to

> **Returns** a list of all files part of the value (e.g. header and data file)
>
> **Return type** list

```
description = 'Meta Image file format'
```

```
extension = 'mhd'
```

NiftiImageFile Module

class fastr.resources.datatypes.NiftiImageFile.**NiftiImageFile**(*value=None*)

Bases: *fastr.datatypes.TypeGroup*

```
__abstractmethods__ = frozenset()
```

```
__module__ = 'fastr.resources.datatypes.NiftiImageFile'
```

```
description = 'NiftiTypeGroup'
```

NiftiImageFileCompressed Module

class fastr.resources.datatypes.NiftiImageFileCompressed.**NiftiImageFileCompressed**(*value=None*,
*for-
mat_=None*)

Bases: *fastr.datatypes.URLType*

```
__abstractmethods__ = frozenset()
```

```
__module__ = 'fastr.resources.datatypes.NiftiImageFileCompressed'
```

checksum()

Return the checksum of this URL type

> **Returns** checksum string
>
> **Return type** str

```
description = 'Compressed Nifti Image File format'
```

```
extension = 'nii.gz'
```

NiftiImageFileUncompressed Module

class fastr.resources.datatypes.NiftiImageFileUncompressed.**NiftiImageFileUncompressed**(*value=*
*for-
mat_=*

Bases: *fastr.datatypes.URLType*

```
__abstractmethods__ = frozenset()
```

```
__module__ = 'fastr.resources.datatypes.NiftiImageFileUncompressed'
```

```
description = 'Nifti Image File format'
```

```
extension = 'nii'
```

NrrdImageFile Module

class fastr.resources.datatypes.NrrdImageFile.**NrrdImageFile**(*value=None,* *format_=None*)

   Bases: *fastr.datatypes.URLType*

   __abstractmethods__ = frozenset()

   __module__ = 'fastr.resources.datatypes.NrrdImageFile'

   description = 'Nrrd Image File format'

   extension = 'nrrd'

Number Module

class fastr.resources.datatypes.Number.**Number**(*value=None*)
   Bases: *fastr.datatypes.TypeGroup*

   __abstractmethods__ = frozenset()

   __module__ = 'fastr.resources.datatypes.Number'

   description = 'an numeric value'

ProvNFile Module

class fastr.resources.datatypes.ProvNFile.**ProvNFile**(*value=None, format_=None*)
   Bases: *fastr.datatypes.URLType*

   __abstractmethods__ = frozenset()

   __module__ = 'fastr.resources.datatypes.ProvNFile'

   description = 'Provenance file in the prov N format'

   extension = 'provn'

String Module

class fastr.resources.datatypes.String.**String**(*value=None, format_=None*)
   Bases: *fastr.datatypes.ValueType*

   __abstractmethods__ = frozenset()

   __module__ = 'fastr.resources.datatypes.String'

   description = 'A simple string value'

TifImageFile Module

class fastr.resources.datatypes.TifImageFile.**TifImageFile**(*value=None,* *format_=None*)

   Bases: *fastr.datatypes.URLType*

   __abstractmethods__ = frozenset()

   __module__ = 'fastr.resources.datatypes.TifImageFile'

```
description = 'Tif Image File format'

extension = ('tiff', 'tif')
```

`TxtFile` Module

class fastr.resources.datatypes.TxtFile.**TxtFile**(*value=None*, *format_=None*)
    Bases: *fastr.datatypes.URLType*

    __abstractmethods__ = frozenset()

    __module__ = 'fastr.resources.datatypes.TxtFile'

    description = 'General text file'

    extension = 'txt'

`UnsignedInt` Module

class fastr.resources.datatypes.UnsignedInt.**UnsignedInt**(*value=None*, *format_=None*)
    Bases: *fastr.datatypes.ValueType*

    __abstractmethods__ = frozenset()

    __module__ = 'fastr.resources.datatypes.UnsignedInt'

    description = 'an unsigned integer value'

    value
        The value of object instantiation of this DataType.

tools Package

`tools` Package

Subpackages

fastr Package

Subpackages

util Package

Subpackages

1.0 Package

Subpackages

bin Package

`auto_prefix` Module

`auto_prefix_negate` Module

`delay` Module

`fail` Module

`passthroughauto` Module

`range` Module

`stdout` Module

test Package

`test` Package

`test_datatypes` Module

utils Package

`utils` Package

A collections of utils for fastr (command line tools or non-core functionality)

`compare` Module

Module to compare various fastr specific things such as a execution directory or a reference directory.

fastr.utils.compare.**compare_execution_dir**(*path1*, *path2*)

fastr.utils.compare.**compare_job_dirs**(*sample*, *node*, *node_dir1*, *node_dir2*)

fastr.utils.compare.**compare_job_output_data**(*output*, *job1*, *job2*)

fastr.utils.compare.**compare_set**(*set1*, *set2*, *path*, *sub_compare_func*, *f_args=None*, *f_kwargs=None*)
Compare two sets and dispatch each item to a sub comparison function

> **Parameters**
>
> > - **set1** (`Iterable`) – first set of items
> > - **set2** (`Iterable`) – second set of items
> > - **path** (`str`) – identifier of the data location
> > - **sub_compare_func** – function to apply to items
> > - **f_args** – args to pass to sub_compare_func
> > - **f_kwargs** – kwargs to pass to sub_compare_func
>
> **Returns** generator that iterates over the differences

> **Return type** generator

`fastr.utils.compare.`**`compare_value_dict_item`**(*key*, *data1*, *data2*, *path*)

`fastr.utils.compare.`**`compare_value_list`**(*data1*, *data2*, *path*, *key=None*)

`dicteq` Module

Some helper function to compare dictionaries and find the parts of the dict that are different. This is mostly to help in debugging.

`fastr.utils.dicteq.`**`dicteq`**(*self*, *other*)

> Compare two dicts for equality
>
> > **Parameters**
> >
> > > - **`self`** – the first object to compare
> > >
> > > - **`other`** – the oth
> >
> > **Returns**

`fastr.utils.dicteq.`**`diffdict`**(*self*, *other*, *path=None*, *visited=None*)

> Find the differences in two dictionaries.
>
> > **Parameters**
> >
> > > - **`self`** – the first object to compare
> > >
> > > - **`other`** (*dict*) – other dictionary
> > >
> > > - **`path`** (*list*) – the path for nested dicts (too keep track of recursion)
> >
> > **Returns** list of messages indicating the differences
> >
> > **Return type** list

`fastr.utils.dicteq.`**`diffobj`**(*self*, *other*, *path=None*, *visited=None*)

> Compare two objects by comparing their __dict__ entries
>
> > **Parameters**
> >
> > > - **`self`** – the first object to compare
> > >
> > > - **`other`** – other objects to compare
> > >
> > > - **`path`** (*list*) – the path for nested dicts (too keep track of recursion)
> >
> > **Returns** list of messages
> >
> > **Return type** list

`fastr.utils.dicteq.`**`diffobj_str`**(*self*, *other*)

> Compare two objects by comparing their __dict__ entries, but returns the differences in a single string ready for logging.
>
> > **Parameters**
> >
> > > - **`self`** – the first object to compare
> > >
> > > - **`other`** – other object to compare to
> >
> > **Returns** the description of the differences
> >
> > **Return type** str

`gettools` Module

fastr.utils.gettools.**main**()

`multiprocesswrapper` Module

fastr.utils.multiprocesswrapper.**function_wrapper**(*filepath*, *fnc_name*, **args*, ***kwargs*)

`verify` Module

fastr.utils.verify.**verify_resource_loading**(*filename*)

fastr.utils.verify.**verify_tool**(*filename*)
> Verify that a file

Subpackages

cmd Package

`cmd` Package

fastr.utils.cmd.**add_parser_doc_link**(*parser*, *filepath*)

fastr.utils.cmd.**find_commands**()

fastr.utils.cmd.**get_command_module**(*command*)

fastr.utils.cmd.**main**()

fastr.utils.cmd.**print_help**(*commands=None*)

`cat` Module

fastr.utils.cmd.cat.**fastr_cat**(*infile*, *path*)

fastr.utils.cmd.cat.**get_parser**()

fastr.utils.cmd.cat.**main**()
> Print information from a job file

`dump` Module

fastr.utils.cmd.dump.**create_zip**(*directory*, *output_file*)

fastr.utils.cmd.dump.**get_parser**()

fastr.utils.cmd.dump.**main**()
> Dump the contents of a network run tempdir into a zip for remote assistance

`execute` Module

`fastr.utils.cmd.execute.`**`get_parser`**`()`

`fastr.utils.cmd.execute.`**`main`**`()`
    Execute a fastr job file

`extract_argparse` Module

`fastr.utils.cmd.extract_argparse.`**`cardinality_from_nargs`**`(`*value*`)`

`fastr.utils.cmd.extract_argparse.`**`datatype_from_type`**`(`*type_*, *metavar*`)`

`fastr.utils.cmd.extract_argparse.`**`extract_argparser`**`(`*filepath*`)`

`fastr.utils.cmd.extract_argparse.`**`find_argparser`**`(`*entry*, *basename='/home/docs/checkouts/readthedocs.org/user_*
        build'`)`

`fastr.utils.cmd.extract_argparse.`**`get_parser`**`()`

`fastr.utils.cmd.extract_argparse.`**`main`**`()`
    Create a stub for a Tool based on a python script using argparse

`provenance` Module

`fastr.utils.cmd.provenance.`**`get_parser`**`()`

`fastr.utils.cmd.provenance.`**`get_prov_document`**`(`*result*`)`

`fastr.utils.cmd.provenance.`**`main`**`()`
    Get PROV information from the result pickle.

`pylint` Module

`fastr.utils.cmd.pylint.`**`get_parser`**`()`

`fastr.utils.cmd.pylint.`**`main`**`()`
    Dump the contents of a network run tempdir into a zip for remote assistance

`fastr.utils.cmd.pylint.`**`run_pylint`**`(`*out_file*, *pylint_args*`)`

`run` Module

`fastr.utils.cmd.run.`**`create_network_parser`**`(`*network*`)`

`fastr.utils.cmd.run.`**`get_parser`**`()`

`fastr.utils.cmd.run.`**`main`**`()`
    Run a Network from the commandline

`sink` Module

`fastr.utils.cmd.sink.`**`get_parser`**`()`

`fastr.utils.cmd.sink.`**`main`**`()`
    Command line access to the IOPlugin sink

`fastr.utils.cmd.sink.`**`sink`**`()`

source **Module**

`fastr.utils.cmd.source.`**`get_parser`**`()`

`fastr.utils.cmd.source.`**`main`**`()`
    Command line access to the IOPlugin source

`fastr.utils.cmd.source.`**`source`**`()`

test **Module**

`fastr.utils.cmd.test.`**`check_network`**`(`*args*`)`

`fastr.utils.cmd.test.`**`check_networks`**`(`*args*`)`

`fastr.utils.cmd.test.`**`check_tool`**`(`*args*`)`

`fastr.utils.cmd.test.`**`check_tools`**`(`*args*`)`

`fastr.utils.cmd.test.`**`directory`**`(`*path*`)`
    Make sure the path is a valid directory for argparse

`fastr.utils.cmd.test.`**`get_parser`**`()`

`fastr.utils.cmd.test.`**`main`**`()`
    Run the tests of a tool to verify the proper function

`fastr.utils.cmd.test.`**`tool`**`(`*value*`)`
    Make sure the value is a correct tool for argparse or reference directory

trace **Module**

`fastr.utils.cmd.trace.`**`get_parser`**`()`

`fastr.utils.cmd.trace.`**`main`**`()`
    Trace samples/sinks from a run

`fastr.utils.cmd.trace.`**`print_job_result`**`(`*job_file*`)`

`fastr.utils.cmd.trace.`**`print_sample_sink`**`(`*sink_data*, *dirname*, *sample_sink_tuples*, *verbose*`)`

`fastr.utils.cmd.trace.`**`print_samples`**`(`*sink_data*, *sample_ids*, *verbose*`)`

`fastr.utils.cmd.trace.`**`print_sinks`**`(`*sink_data*, *sink_ids*, *verbose*`)`

`fastr.utils.cmd.trace.`**`read_sink_data`**`(`*infile*`)`

`fastr.utils.cmd.trace.`**`switch_sample_sink`**`(`*sink_data*`)`

upgrade **Module**

`fastr.utils.cmd.upgrade.`**`find_tool`**`(`*toolspec*`)`

`fastr.utils.cmd.upgrade.`**`get_parser`**`()`

`fastr.utils.cmd.upgrade.`**`main`**`()`
    Print information from a job file

`fastr.utils.cmd.upgrade.`**`upgrade`**`(`*infile*, *outfile*`)`

`verify` Module

`fastr.utils.cmd.verify.`**`get_parser`**`()`

`fastr.utils.cmd.verify.`**`main`**`()`
    Verify fastr resources, at the moment only tool definitions are supported.

`webapp` Module

`fastr.utils.cmd.webapp.`**`get_parser`**`()`

`fastr.utils.cmd.webapp.`**`main`**`()`
    Start the fastr webapp and open in a new browser tab

`fastr.utils.cmd.webapp.`**`open_url`**`(`*url*`)`

secrets Package

`secrets` Package

`secretprovider` Module

`secretservice` Module

Subpackages

exceptions Package

`exceptions` Package

`couldnotdeletecredentials` Module

`couldnotretrievecredentials` Module

`couldnotsetcredentials` Module

`notimplemented` Module

`providernotfound` Module

providers Package

`providers` Package

keyringprovider **Module**

netrcprovider **Module**

web Package

web Package

api **Module**

class fastr.web.api.**NetworkApi**(*api=None*, **args*, ***kwargs*)
 Bases: flask_restplus.resource.Resource

 __module__ = 'fastr.web.api'

 endpoint = 'api_network'

 get(*id_*)
  Get a Network json description from the server

 mediatypes()

 methods = {'GET'}

class fastr.web.api.**NetworkListApi**(*api=None*, **args*, ***kwargs*)
 Bases: flask_restplus.resource.Resource

 __module__ = 'fastr.web.api'

 endpoint = 'api_networks'

 get()
  Get a list of the networks

 mediatypes()

 methods = {'GET'}

class fastr.web.api.**ObjectUrl**(*object_classs*, ***kwargs*)
 Bases: flask_restplus.fields.Raw

 __init__(*object_classs*, ***kwargs*)
  Initialize self. See help(type(self)) for accurate signature.

 __module__ = 'fastr.web.api'

 __schema_type__ = 'string'

 __slotnames__ = []

 format(*value*)
  Formats a field's value. No-op by default - field classes that modify how the value of existing object keys should be presented should override this and apply the appropriate formatting.

   Parameters **value** – The value to format

   Raises **MarshallingError** – In case of formatting problem

  Ex:

```
class TitleCase(Raw):
    def format(self, value):
        return unicode(value).title()
```

class `fastr.web.api.`**Run**(*id_*, *network*, *source_data*, *sink_data*)

> Bases: `object`
>
> **__dict__ = mappingproxy({'__module__': 'fastr.web.api', '__init__': <function Run.__**
>
> **__init__**(*id_*, *network*, *source_data*, *sink_data*)
>
> > Initialize self. See help(type(self)) for accurate signature.
>
> **__module__ = 'fastr.web.api'**
>
> **__weakref__**
>
> > list of weak references to the object (if defined)
>
> **abort**()
>
> **run_network**(*network*, *source_data*, *sink_data*, *abort_lock*)
>
> **status**()

class `fastr.web.api.`**RunApi**(*api=None*, **args*, ***kwargs*)

> Bases: `flask_restplus.resource.Resource`
>
> Run API documentation
>
> **__module__ = 'fastr.web.api'**
>
> **delete**(*id_*)
>
> > Abort a Network run and stop all associated execution
>
> **endpoint = 'api_run'**
>
> **get**(*id_*)
>
> > Get information about a Network run
>
> **mediatypes**()
>
> **methods = {'DELETE', 'GET'}**

class `fastr.web.api.`**RunListApi**(*api=None*, **args*, ***kwargs*)

> Bases: `flask_restplus.resource.Resource`
>
> **__module__ = 'fastr.web.api'**
>
> **endpoint = 'api_runs'**
>
> **get**()
>
> > Get a list of all Network runs on the server
>
> **mediatypes**()
>
> **methods = {'GET', 'POST'}**
>
> **post**()
>
> > Create a new Network run and start execution
>
> **request_parser = <flask_restplus.reqparse.RequestParser object>**

class `fastr.web.api.`**StatusApi**(*api=None*, **args*, ***kwargs*)

> Bases: `flask_restplus.resource.Resource`
>
> **__module__ = 'fastr.web.api'**
>
> **endpoint = 'api_status'**

get(*id_*)
> Get the status of a Network Run on the server

mediatypes()

methods = {'GET'}

class fastr.web.api.**SubUrl**(*object_classs*, *subfield*, ***kwargs*)
> Bases: flask_restplus.fields.Raw

__init__(*object_classs*, *subfield*, ***kwargs*)
> Initialize self. See help(type(self)) for accurate signature.

__module__ = 'fastr.web.api'

__schema_type__ = 'string'

format(*value*)
> Formats a field's value. No-op by default - field classes that modify how the value of existing object keys should be presented should override this and apply the appropriate formatting.

> > **Parameters value** – The value to format

> > **Raises MarshallingError** – In case of formatting problem

> Ex:

```python
class TitleCase(Raw):
    def format(self, value):
        return unicode(value).title()
```

class fastr.web.api.**ToolApi**(*api=None*, **args*, ***kwargs*)
> Bases: flask_restplus.resource.Resource

__module__ = 'fastr.web.api'

endpoint = 'api_version_tool'

get(*id_*, *version=None*)
> Get a Tool json description from the server

mediatypes()

methods = {'GET'}

class fastr.web.api.**ToolListApi**(*api=None*, **args*, ***kwargs*)
> Bases: flask_restplus.resource.Resource

__module__ = 'fastr.web.api'

endpoint = 'api_tools'

get()
> Get a list of all Tools known to the server

mediatypes()

methods = {'GET'}

fastr.web.api.**network_lock_thread**(*lock*, *network*)

fastr.web.api.**network_runner**(*network*, *source_data*, *sink_data*, *chuck_status*, *job_status*, *job_results*, *abort_lock*)

fastr.web.api.**update_job_result**(*job*, *job_status*, *job_results*)

fastr.web.api.**update_status**(*job*, *job_status*)

`run` Module

fastr.web.run.**main**()

fastr.web.run.**runapp**(*debug=False*)

`views` Module

fastr.web.views.**doc**()

fastr.web.views.**index**()

fastr.web.views.**network**(*name=None*)

fastr.web.views.**networks**()

fastr.web.views.**prov**()

fastr.web.views.**shutdown**()

fastr.web.views.**shutdown_server**()

fastr.web.views.**tool**(*toolname=None*, *version=None*)

fastr.web.views.**websocket_client**()

Indices and tables

- genindex
- modindex
- search

Python Module Index

Index

Symbols

G

GE_NATIVE_SPEC (*fastr.plugins.DRMAAExecution attribute*), 203

generate_jobs() (*fastr.execution.networkrun.NetworkRun method*), 151

get() (*fastr.web.api.NetworkApi method*), 249

get() (*fastr.web.api.NetworkListApi method*), 249

get() (*fastr.web.api.RunApi method*), 250

get() (*fastr.web.api.RunListApi method*), 250

get() (*fastr.web.api.StatusApi method*), 251

get() (*fastr.web.api.ToolApi method*), 251

get() (*fastr.web.api.ToolListApi method*), 251

get_arguments() (*fastr.plugins.FastrInterface method*), 208

get_base_version() (*in module fastr.version*), 84

get_command() (*fastr.plugins.FastrInterface method*), 208

get_command_module() (*in module fastr.utils.cmd*), 245

get_cpu_usage() (*in module fastr.helpers.sysinfo*), 169

get_deferred() (*fastr.execution.job.Job method*), 139

get_drmaa_info() (*in module fastr.helpers.sysinfo*), 169

get_field() (*fastr.helpers.configmanager.Config method*), 161

get_hg_info() (*in module fastr.version*), 84

get_hostinfo() (*in module fastr.helpers.sysinfo*), 169

get_job() (*fastr.plugins.executionplugin.ExecutionPlugin method*), 229

get_memory_usage() (*in module fastr.helpers.sysinfo*), 169

get_message() (*in module fastr.exceptions*), 83

get_mounts() (*in module fastr.helpers.sysinfo*), 169

get_object_version() (*fastr.plugins.managers.networkmanager.NetworkManager method*), 232

get_object_version() (*fastr.plugins.managers.objectmanager.ObjectManager method*), 233

get_object_version() (*fastr.plugins.managers.toolmanager.ToolManager method*), 236

get_os() (*in module fastr.helpers.sysinfo*), 169

get_output_datatype() (*fastr.execution.job.Job method*), 139

get_output_datatype() (*fastr.execution.job.SourceJob method*), 144

get_output_info() (*fastr.execution.macronoderun.MacroNodeRun method*), 148

get_output_info()

(*fastr.planning.node.MacroNode method*), 196

get_parent_provenance() (*fastr.core.provenance.Provenance static method*), 97

get_parser() (*in module fastr.utils.cmd.cat*), 245

get_parser() (*in module fastr.utils.cmd.dump*), 245

get_parser() (*in module fastr.utils.cmd.execute*), 246

get_parser() (*in module fastr.utils.cmd.extract_argparse*), 246

get_parser() (*in module fastr.utils.cmd.provenance*), 246

get_parser() (*in module fastr.utils.cmd.pylint*), 246

get_parser() (*in module fastr.utils.cmd.run*), 246

get_parser() (*in module fastr.utils.cmd.sink*), 246

get_parser() (*in module fastr.utils.cmd.source*), 247

get_parser() (*in module fastr.utils.cmd.test*), 247

get_parser() (*in module fastr.utils.cmd.trace*), 247

get_parser() (*in module fastr.utils.cmd.upgrade*), 247

get_parser() (*in module fastr.utils.cmd.verify*), 248

get_parser() (*in module fastr.utils.cmd.webapp*), 248

get_path_from_url() (*in module fastr.data.url*), 114

get_processes() (*in module fastr.helpers.sysinfo*), 169

get_prov_document() (*in module fastr.utils.cmd.provenance*), 246

get_python() (*in module fastr.helpers.sysinfo*), 169

get_result() (*fastr.execution.job.InlineJob method*), 137

get_result() (*fastr.execution.job.Job method*), 139

get_result() (*fastr.execution.job.SinkJob method*), 144

get_saved_version() (*in module fastr.version*), 84

get_sourced_nodes() (*fastr.execution.inputoutputrun.InputRun method*), 128

get_sourced_nodes() (*fastr.execution.inputoutputrun.SubInputRun method*), 134

get_sourced_nodes() (*fastr.execution.noderun.NodeRun method*), 153

get_sourced_nodes() (*fastr.planning.inputoutput.Input method*), 179

get_sourced_nodes() (*fastr.planning.inputoutput.SubInput method*), 184

get_sourced_nodes() (*fastr.planning.node.Node method*), 197

get_sourced_outputs()

W

X