

# Command

Created by John Flavin, last modified on Feb 15, 2018

If you want XNAT to execute your docker image, you will need a Command. The Command is a collection of properties that describe your docker image, and which XNAT can read to understand what your image is and how to run it:

- What kind of image is it? (Currently only docker images are supported.)
- Which docker image is it? What is its name? ID?
- Does it have a human-friendly name we can use for it?
- How do you run it? What does the command-line string look like?
- Does it need files? Where should they go? How do you want to get those out of XNAT?
- Does it produce any files at the end? Those have to get back into XNAT, so where should they go?

## Command object format

```
{  
    "name": "",  
    "label": "",  
    "description": "",  
    "version": "",  
    "schema-version": "1.0",  
    "type": "docker",  
    "info-url": "",  
    "image": "",  
    "index": "", // only valid for docker images  
    "hash": "", // only valid for docker images  
    "working-directory": "",  
    "command-line": "", pet_pipeline_run(XNAT_object_properties) # see XNAT object properties section below  
    "reserve-memory": "", 16 GB  
    "limit-memory": "", 32 GB  
    "limit-cpu": "",  
    "override-entrypoint": false,  
    "mounts": [  
        {  
            "name": "",  
            "writable": false,  
            "path": ""  
        }  
    ],  
    "environment-variables": {  
        "envName1": "envVal1",  
        "envName2": "#inputReplacementKey#"  
    },  
}
```

**Files needed by the PET pipeline are listed in CNDAs → rawdata and Scans → DICOM.** For each pipeline instance, I will need to parse header files for both rawdata and DICOMs headers and use header field values to uniquely identify the needed files. Since the rawdata set for a Session is ~30 GB, the parsing needs to be done before attempting to retrieve rawdata files from storage. I don't know what mechanisms are needed for the retrieval.

The results of a pipeline instance will be similar to the data objects that you've been working with in your front-end work for the PPG. I can adjust naming/filesystem conventions as needed/recommended. Some intermediate data objects can get deleted but the storage requirements of output will be of the same order as the storage size of the inputs. These will need to go back into XNAT.

```
"ports": {
    "80": "8080",
    "22": "52222"
}, // "ports" only valid for docker images
"inputs": [
    {
        "name": "",
        "description": "",
        "type": "",
        "required": true,
        "matcher": "",
        "default-value": "",
        "replacement-key": "",
        "command-line-flag": "",
        "command-line-separator": "",
        "true-value": "",
        "false-value": ""
    }
],
"outputs": [
    {
        "name": "",
        "description": "",
        "required": true,
        "mount": "",
        "glob": ""
    }
],
"xnat": [
    {
        "name": "",
        "description": "",
        "contexts": [""],
        "external-inputs": [
            {
                "name": "",
                "description": "",
                "type": "",
                "matcher": "",
                "default-value": "",
                "user-settable": true,
                "replacement-key": "",
                "provides-value-for-command-input": "",
                "provides-files-for-command-mount": "",
                "via-setup-command": "repo/image:version[:commandname]"
            }
        ]
    }
]
```

```

        }
    ],
    "derived-inputs": [
        {
            "name": "",
            "description": "",
            "type": "",
            "matcher": "",
            "default-value": "",
            "user-settable": true,
            "replacement-key": "",
            "provides-value-for-command-input": "",
            "provides-files-for-command-mount": "",
            "via-setup-command": "repo/image:version[:commandname]",
            "derived-from-wrapper-input": "",
            "derived-from-xnat-object-property": ""
        }
    ],
    "output-handlers": [
        {
            "name": "",
            "type": "",
            "accepts-command-output": "",
            "via-wrapup-command": "",
            "as-a-child-of-wrapper-input": "",
            "label": ""
        }
    ]
}
]
}
}

```

- **name** - The name of the command. Should be unique, but that is not required.
- **label** - A short human-friendly name of the command. If none is provided, the name will be used.
- **description** - A longer, human-friendly description of the command.
- **version** - The version of the command document you are writing.
- **schema-version** - The version of this schema you are reading, which is 1.0. All commands should use "schema-version": "1.0".
- **type** - The image type. Currently only "docker" is supported.
- **info-url** - A URL where more info on the command, or the image, or both, can be found.
- **image** - An identifier of the image this command describes. For Docker images, this should be in "repository/image-name:tag" format. If the command JSON is embedded in the labels of a docker image, then this field may be omitted.
- **index** - (Docker images only) The index, or hub, where this image can be found. For instance, if the image is on the public Docker Hub, the index value should be "https://index.docker.io/v1".
- **hash** - (Docker images only) A sha hash value for the image.
- **working-directory** - The working directory in which the command line should be executed.
- **command-line** - This string is a templated version of the command-line string that will be executed inside the container. The templated portions will be resolved at launch time with the values of the command's inputs. See the section on [template strings](#) below for more detail.

- **reserve-memory** Integer value in MB to ensure containers have at least this much memory available
- **limit-memory** Integer value in MB to ensure containers don't use more memory than this
- **limit-cpu** Float value to ensure containers don't use more cycle shares than this. For example, a value of 1.5 would mean the container can't use more cycles than 1.5 CPUs are capable of.
- **override-entrypoint** - A boolean value (true/false). If false, the entrypoint will not be touched; any entrypoint that the underlying image has will be used. If true, the entrypoint will be overridden with an empty string.
- **mounts** - A list of mount points that will be created for your container.
  - **name** - The name of the mount. You can use this to refer to the mount elsewhere in the command, e.g. when creating an output.
  - **writable** - true/false: whether the mount should be writable. Output mounts are always writable, whereas input mounts are typically read-only but can be made writable with this flag.
  - **path** - The absolute path inside your container at which the mount will be created.
- **environment-variables** - Key/value pairs of environment variables to set in the container. Both keys and values can be templates that will be filled by input values at runtime.
- **ports** - (Docker images only) String key/value pairs of ports to expose. The key is the port inside the container, the value is the port to expose out on the host. In other words, entries in this map should be of the form "container\_port": "host\_port". Keys and values can be templates.
- **inputs** - A list of inputs that will be used to resolve the command and launch the container. See [Command Inputs](#).
  - **name** - The name of the input. You can use this to refer to the input elsewhere in the command.
  - **description** - A human-friendly description of the input.
  - **type** - One of string, boolean, number, or file. See the section on [input types](#) below for more. Default: string.
  - **required** - A boolean value (true/false) whether this input is required. If a required input does not have a value at runtime, an error is thrown. Default: false.
  - **matcher** - A [JSONPath filter](#) used to determine if an input value is valid or not.
  - **default-value** - A value that will be used if no other value is provided at runtime.
  - **replacement-key** - A shorthand way to refer to this input's value elsewhere in the command. Default: the input's name bracketed by "#"; e.g. for an input named "foo" the default replacement-key is "#foo#".
  - **command-line-flag** - When this input's value is replaced in the command-line string, it is preceded by this flag. E.g. an input with value "foo" and command-line-flag "--flag" will appear in the command-line string as "--flag foo".
  - **command-line-separator** - The character separating the command-line-flag from the value in the command-line. Default: " ".
  - **true-value** - The string to use in the command line for a boolean input when its value is true. Some examples: "true", "T", "Y", "1", "--a-flag". Default: "true".
  - **false-value** - The string to use in the command line for a boolean input when its value is false. Some examples: "false", "F", "N", "0", "--some-other-flag". Default: "false".
  - **mount** - (Only for inputs of type "file") The name of a mount—which must be defined in this command—into which container service will .
- **outputs** - A list of outputs that will be used to upload files produced by the container. See [Command Outputs](#).
  - **name** - The name of the output.
  - **description** - A human-friendly description of the output.
  - **required** - A boolean value (true/false) whether this output is required. If a required output does not match any files when container is finished, an error is thrown. Default: true.
  - **mount** - The name of a mount, which must be defined in this command and must have type "output", into which your container wrote whatever file(s) you intend to upload.
  - **path** - The relative path within a mount at which output files can be found. Value can be templated with input replacement keys.
  - **glob** - A glob-style matcher for the files to upload. If "glob" is blank, then all files found at relative path "path" within the mount will be uploaded.
- **xnat** - A list of XNAT Command Wrappers, or just "Wrappers" for short, in which you can define how to pull files and properties from XNAT objects into your containers, and upload the containers' outputs back.
  - **name** - A user-friendly name. Example: "dcm2niix on a scan".
  - **description** - A longer description of what this command wrapper does: What XNAT object(s) does it take as inputs? How does it use those to fill the command's inputs? Where does it upload the command's outputs?
  - **contexts** - A list of [XNAT data types](#) on which this command wrapper can be run. If none are provided, the list of contexts is populated from the data types of the external inputs.
  - **external-inputs** - A List of Inputs to the Command Wrapper that will come in when a launch is requested. See [Wrapper Inputs](#) for more.
    - **name**
    - **description**
    - **type** - One of the basic types (string, boolean, number, file) or the XNAT object types (Project, Subject,

Session, Scan, Assessor, Resource, Config). See the section on [input types](#) below for more.

- **matcher** - A [JSONPath filter](#) used to determine if an input value is valid or not. For instance, if the parent input is a Session, and this input is a Scan, we can make sure that this input only matches scans with a DICOM resource by setting the matcher to "DICOM" in `@.resources[*].label`, or only matches scans of a certain type by setting the matcher to `@.scan-type == "MPRAGE"`.
- **default-value**
- **user-settable** - true/false. Should this Input be exposed to users who are launching via a UI? See the section [User-settable or not?](#) for the use-cases where one might want to set this to "false".
- **replacement-key** - A shorthand way to refer to this input's value elsewhere in the command. Default: the input's name bracketed by "#"; e.g. for an input named "foo" the default replacement-key is "#foo#".
- **provides-value-for-command-input** - The name of a Command Input, which will receive its value from this input.
- **provides-files-for-command-mount** - The name of a Command Mount, which will receive files from this input.
- **via-setup-command** - A reference to a setup command image (format: `repo/image:version[:commandname]` where the `commandname` is optional). See the page on [Setup Commands](#) for more.
- **derived-inputs** - A List of Inputs to the Command Wrapper that will not come in from outside, but instead will be derived from other inputs as parents or children. See [Wrapper Inputs](#) for more.
  - **name**
  - **description**
  - **type** - One of the basic types (string, boolean, number) or the XNAT object types (Project, Subject, Session, Scan, Assessor, Resource, Config, File, File[], Directory). See the section on [input types](#) below for more.
  - **matcher** - A [JSONPath filter](#) used to determine if an input value is valid or not. For instance, if the parent input is a Session, and this input is a Scan, we can make sure that this input only matches scans with a DICOM resource by setting the matcher to "DICOM" in `@.resources[*].label`, or only matches scans of a certain type by setting the matcher to `@.scan-type == "MPRAGE"`.
  - **default-value**
  - **user-settable** - true/false. Should this Input be exposed to users who are launching via a UI? See the section [User-settable or not?](#) for the use-cases where one might want to set this to "false".
  - **replacement-key** - A shorthand way to refer to this input's value elsewhere in the command. Default: the input's name bracketed by "#"; e.g. for an input named "foo" the default replacement-key is "#foo#".
  - **provides-value-for-command-input** - The name of a Command Input, which will receive its value from this input.
  - **provides-files-for-command-mount** - The name of a Command Mount, which will receive files from this input.
  - **via-setup-command** - A reference to a setup command image (format: `repo/image:version[:commandname]` where the `commandname` is optional). See the page on [Setup Commands](#) for more.
  - **derived-from-wrapper-input** - Name of a Wrapper input which is a "parent" to this input. See [Deriving values](#).
  - **derived-from-xnat-object-property** - Property of an XNAT object that will be used for the value of this input. See [Deriving values](#).
- **output-handlers** - A list of [output handlers](#). You use these to instruct the container service how and where to upload your container's outputs.
  - **name**
  - **type** - The type of object that will be created in XNAT. Currently only "Resource" is accepted.
  - **accepts-command-output** - The name of a [command output](#) whose files will be handled.
  - **via-wrapup-command** - A reference to a wrapup command image (format: `repo/image:version[:commandname]` where the `commandname` is optional). See the page on [Wrapup Commands](#) for more.
  - **as-a-child-of-wrapper-input** - The name of a [wrapper input](#)—either external or derived—that refers to an XNAT object. The output files will be uploaded as a new child of that object.
  - **label** - The label of the new Resource that will be created from these files.

## Mounts

Mounts are the way to get files into and back out of your container. If you need the container service to stage files from XNAT

into your container, or you want container service to upload any files your container creates back into XNAT, then you need to use mounts.

A mount only has a few properties; we will summarize those here, and go into more detail on each below. When you create a mount, you give it a...

- "name" so that you can refer to it in other parts of the command
- "path", which is the path at which it will be found *inside* the container. You don't need to specify the path outside the container; we will manage that for you.
- "writable" boolean, which specifies whether the mount is read-only or writable.

A little more detail on the `writable` flag: Mounts that are referenced by [command outputs](#) are always `writable`. If a mount is not referenced by any output—i.e. it is only used to mount input files—it will typically be `read-only`. This is so that files can be mounted directly from the XNAT archive, which should never be written to directly. However, this means that if a container does try to write anything to a location that is a `read-only` mount, the container will fail with a runtime error. If you know that a container will write to a location where you want to place input files, you can explicitly set an input mount to "`writable=true`". This means that before the container is launched, any input files will be copied out of the archive into a `writable` directory which will then be mounted.

A mount can be used for both an input and an output. That means the input files will be copied into the directory before launch, and the same directory will be searched for output files upon container completion. If you aren't careful, the input files will be re-uploaded along with the output files. The `output.path` and `output.glob` properties can be carefully crafted to avoid this effect.

## Getting files into a mount

Define a [wrapper input](#) for the kind of object that will provide the files you need. Typically this will be a [Resource](#), but that resource input will almost certainly need to be derived from some higher-level input object like a [Session](#) or a [Scan](#). On the input that provides the files, set the mount's name as the value of the "`provides-files-for-command-mount`" property.

More info to come.

## Getting files out of a mount

[I can generate a file catalog from XNAT\\_object\\_properties.](#)

- Define a [command output](#) that details the files you expect to see in your mount.
- Define an [XNAT output handler](#) that will take the files from the output, and attach them to some XNAT object.

More info to come. # Inputs

[We'll need to ask John Flavin how to define the handler.](#)

## Command Inputs

What information does your container need?

Command Inputs allow you define what information needs to be provided in order to launch your container: files, command-line arguments, environment variables, etc. If you need some bit of information to be variable and set at launch time, it should be a command input.

[Sessions have variable numbers and types of scans so the files needed and files generated will depend on information contained in XNAT\\_object\\_properties.](#)

## Wrapper Inputs

How do you get information from XNAT into your Command's inputs?

At the level of command inputs, it does not necessarily matter where the runtime values come from. They could in principle come from some XNAT object, or from a user's input, or from some other contextual information.

The XNAT inputs—i.e. the `command.xnat.external-inputs` and `command.xnat.derived-inputs` objects—give you a way to tell XNAT how to use XNAT objects—their properties, files, and hierarchical relationships—to provide values to the Command inputs.

[I need more information on how to define/instantiate these input objects. Would you have familiarity with these?](#)

## Deriving values

If you have a derived wrapper input, it can derive its value from a parent wrapper input.

Info dump:

- The parent can either be an external input or another derived input. But it must be an input on the same wrapper, not on the command or a different wrapper.
- The type of the parent input must be one of the XNAT object types: Project, Subject, Session, Scan, Assessor, or Resource.
- If the type of the child input is also an XNAT type, it must obey the usual XNAT hierarchy.
  - Project -> Subject
  - Subject -> Session
  - Session -> Scan
  - Session -> Assessor
  - Any of Project, Subject, Session, Scan, or Assessor -> Resource
- Subject parents cannot have children of type Directory. All the other XNAT types can.
- If the child type is not one of the XNAT types—e.g. string or number—you can derive the value by specifying the property on the parent object that you want to pull out using the input's derived-from-xnat-object-property field. For a list of the properties you can use, see [XNAT object properties](#).

## XNAT object properties

A list of all the properties for all the XNAT object types:

- Project
  - id
  - label
  - xsiType
  - uri
  - directory
- Subject
  - id
  - label
  - xsiType
  - uri
  - project-id
- Session
  - id
  - label
  - xsiType
  - uri
  - directory
  - project-id
- Scan
  - id
  - label
  - xsiType
  - uri
  - directory
  - integer-id
  - scan-type
- Assessor
  - id
  - label
  - xsiType
  - uri
  - directory
- Resource
  - id

I'll try to expose these in some kind of API for the pipeline.

- label
- xsiType
- uri
- directory
- integer-id

This initial list is small. There are many more properties that could be added over time.

**Note:** The uri property is the REST-style identifier of the object. For instance, a project's URI will be /projects/{projectId}, subject's may be /subjects/{subjectId} or /projects/{project}/subjects/{subjectLabel}, etc.

## User-settable or not? Let's skip user-settings in the prototypes.

When a user brings up a user interface to launch a container using a Command, they will see some appropriate interface element for each Command input and external XNAT Wrapper input. These interface elements give the user a chance to change the input values before launching the container. If, for some reason, this is not appropriate for a particular input, and users should not be allowed to change the default value, the input can be defined with the property user-settable=false. When the user brings up an interface to launch a container, they will see the input and its value but will not be able to change it.

The more common use-case for this parameter is in XNAT-project-specific settings. Perhaps all the inputs in a Command definition have user-settable=true, but a project owner may choose to configure particular inputs on the Command to have user-settable=false. In this way, they ensure that all containers on their project will be executed with the chosen input value.

## Input Types

Dynamic image sizes ~2GB and there will be 7 PET images per session. Auxiliary images have negligible sizes. A single session requiring 14-20 GB of RAM makes sense for understanding the analysis.

Command input types: string, boolean, number

XNAT Wrapper input types: string, boolean, number, Directory, File, File[], Project, Subject, Session, Scan, Assessor, Resource, Config

**Note:** Some inputs are possible to make but aren't currently functional.

- number inputs should be validated that they are within a range, or that they are in fact numbers and not strings. But currently they are not. They are treated the same as string inputs.
- Directory, File, and File[ ] inputs are possible to make, but the files they refer to cannot be mounted.
- Config inputs are intended to pull their values from the XNAT Config Service. They currently do not.

**Note** A Project input is kind of a bad idea to use on its own. (Probably Subject too.) The performance is really very bad. The entire project, all its subjects, all their sessions, all their scans, and all the resources and files on every one of those things all get loaded into memory and stored in an object. It takes a long time!

If you want to use a Project input, you should set the property "load-children" on the input to false. That will prevent the entire project from being loaded. But, on the other hand, it will prevent you from deriving any child inputs from the Project input. So only use if you need to derive some property from the Project itself.

## Outputs

### Command Outputs

For the PPG, I've heretofore been using folder structures HOME/HYGLY99/Vall for per-subject analysis. For the container, a structure more usable across projects may be: HOME/sessionid/scanid.

If you want your container to produce files that get imported into XNAT, you need to define one or more output objects. You need to define where the files can be found (which output mount they are in, and what is the path within that mount) and where the new to-be-created object will live within XNAT. For the latter, you provide the name of an input, which must be an XNAT object type; the output files will be a new child of that parent input.

The folder structure should accomodate the front-end, but, admittedly, the front-end is most useful when it groups all scans and sessions of a given subject together. So the pipeline will generate results by sessionid, because that is the most that can reasonably be managed in a single pipeline run (~20 GB RAM & ~24 cpu hours). Comparing all sessions for a subject may need to be a distinct pipeline.

# Output Handling

More info to come.

## Template Strings

Most template strings will be derivable from the XNAT object properties and the actual scans that were acquired on the session day, viz., DICOM and rawdata header files.

When you define a Command, you can leave many of the values as “templates”. These templates are placeholder strings, also known as “replacement keys”, which tell the container service “When you launch a container from this Command, you will have values for your inputs; I want you to use one of those values here.”

Lots of properties in the Command can use template strings:

- command-line - See a simple example in the [Hello world example](#), but also see the caveats in the [complex example](#) below.
- environment-variables - Both the environment variable name and value can be templates.
- ports - Both the container port and host port can be templates.
- output.path - The relative path within a mount at which output files can be found.

Should we make a list/API of all the objects you need for your front-end? I'm thinking I should generate JSON objects from the pipeline that you could use. This would be for a subject-wise analysis with image registrations on a distinct pipeline from the session reconstruction pipeline that uses builds scan images using NiftyPET.

## JSONPath

JSONPath is an expression syntax for searching through a JSON object, similar to the way you can use XPath to search through an XML document. The syntax and operators are documented here at the source repository:  
<https://github.com/jayway/JsonPath>.

You can use JSONPath strings as values in several Command fields. When the Command is resolved before it is used to launch a container, those JSONPath strings will be replaced with whatever values they refer to. This is similar to the way you can use a [template string](#) in the Command definition, which gets replaced by a value when the Command is resolved. In fact, anywhere in the Command that you can use a template string, you can also use a JSONPath expression.

When you use a JSONPath expression as a value, you must surround it with carets (^ . . . ^) to signal to the Container Service that it needs to invoke the JSONPath interpreter.

JSONPath expressions start at the root of the Command, which is referred to as “\$”. For instance, you could get the path of a particular mount with name `foo` using the JSONPath expression

```
$ .mounts [ ?(@.name = "foo") ].path
```

Again, remember to surround the JSONPath expression with carets (^) to signal that it should be evaluated.

You can search through the XNAT Command wrappers in a JSONPath expression just like any other part of the Command: `^$.xnat[ ?(@.name = "wrapper-name") ] . . .`. To find in the list the particular wrapper that is being evaluated, you would have to hard-code the wrapper name into such an expression, because there is no easy way to find the name at runtime. However, there is a shorthand way to search through the particular command wrapper that is being evaluated. Instead of surrounding the JSONPath expression with carets (`^$.thing1.thing2...^`), surround it with carets and the word “wrapper” as such: `^wrapper:$ .thing3.thing4^`. Now the root of the JSONPath expression (\$) will refer to the Wrapper, not the Command.

## JSONPath filters

The `input.matcher` property uses a special subset of the JSONPath syntax called a “filter”. In JSONPath expressions that can return a list, you can use one of these expressions to filter out non-matching elements. As a simple example, let's say I have the JSON

```
{
```

```

"ice-cream": [
    {"name": "Strawberry", "flavor": "yummy"}, 
    {"name": "Spaghetti", "flavor": "yucky"} 
]
}

```

The PET data have many “flavor” attributes.  
I’ll try to expose them clearly.

Say I want just the “name”s. The JSONPath expression “\$.ice-cream[\*].name” would give back a list:

```
["Strawberry", "Spaghetti"]
```

But if I filter that list, I can get back just the elements that are “yummy”:

```
$ .ice-cream[?(@.flavor == "yummy")].name
["Strawberry"]
```

See the [source documentation](#) for the filter syntax. The filter is the part inside the parentheses. In the filter expression, we use the character “@” to refer to the element that we are checking.

We can use these filter expressions as the value of the `input.matcher`. When the Command is being resolved, a potential input value must not be rejected by the filter, i.e. it must match the filter condition, to be assigned to the input value. In that way, if an input can receive its value from a [parent input](#) which has many children of a certain type, we can select just the one child that we want to be the value for our input.

For example, say we have a Scan input, with a Session input as “parent”. That Session may have many child scans, but we can use the “matcher” to select one that has a certain scantype by setting it to a filter expression, something like

```
@.scan-type in ["T1", "MPRAGE"]
```

Here’s another example from [xnat/dcm2niix-scan](#).

```

"inputs": [
    {
        "name": "scan",
        "description": "Input scan",
        "type": "Scan",
        "required": true,
        "matcher": "DICOM in @.resources[*].label"
    },
    {
        "name": "scan-dicoms",
        "description": "The scans dicom resource",
        "type": "Resource",
        "parent": "scan",
        "matcher": "@.label == DICOM"
    }
]
```

There are two types of running processes in the pipeline for scan reconstructions with NiftyPET:  
1. python + CUDA calls to the GPU  
2. Matlab runtime

It’s looking to me that I have to expose that there are these two types because the container support requirements are different for the two.

Your front-end would be great to add to the CNDA front-end, but CNDA can be so slow that I’m thinking it would work better to have your front-end have initialization codes that pull everything needed from CNDA, do renderings that your presentation codes need. It seems to me that the front-end should be optimized for the user experience of exploring and finding patterns in the data. Your thoughts?

This Command expects that it will be given a scan as an input, but it wants to run a matcher anyway just to be sure the scan has a DICOM resource ("matcher": "'DICOM' in @.resources[\*].label"). The second input is a child to the first, and it matches the scan's DICOM resource ("matcher": "@.label == 'DICOM'").

## XNAT Command Wrapper

How do you take in one or more XNAT objects and use their properties and files to launch your Command?

More info to come.

## Examples

### Hello world example

Let's go over a simple use case. You have a string input, and you want whatever value gets put into that input to be printed to the screen. We could define a command like this:

```
{  
    "name": "hello-world",  
    "description": "Prints a string to stdout",  
    "type": "docker",  
    "image": "busybox:latest",  
    "command-line": "echo #my_cool_input#",  
    "inputs": [  
        {  
            "name": "my_cool_input",  
            "description": "The string that will be printed",  
            "type": "string",  
            "default-value": "Hello world"  
        }  
    ]  
}
```

In this example, you have one input, which is a string, and is named "my\_cool\_input". The value of command-line is what will be executed inside the container when it is launched. However, before container service launches that container, it will look at that command-line value and see that it recognizes the string #my\_cool\_input# as a replacement key for the input with name my\_cool\_input. Each input has a replacement key, which is by default just the input's name surrounded by hash marks (#). But you can customize an input's replacement key by setting the replacement-key property on the input.

When the container-service sees a replacement key, it will replace it with the input's value. In this case, that means #my\_cool\_input# will be replaced by Hello world. (Well, technically, that is only partially correct. See the [more complex example below](#).)

### More complex command-line example

What I said above is that when the container service looks at one of the strings in the list above and sees an input's

~~you could set input.true-value to some flag and set input.false-value to a blank string, or vice versa. In that way, you could set a flag on the command line only if a certain input is true but do not set any flag if the input is false; for example, you could have the input named "recursive" have false-value="" and true-value="-r".~~

- When the input's value is replaced into the command-line, you can optionally prefix the value with a flag by setting `input.command-line-flag`. So if my input `foo` has value `bar` and `command-line-flag="--my-input-value"`, the flag and value will be smashed together and will replace the replacement key as `--my-input-value bar`. If you want the flag and value to be joined by something other than a space, for instance an equals sign, you can set that as `input.command-line-separator`.

Let's see an example Command with these properties set.

```
{  
  "name": "complex-example",  
  "description": "An example Command with more complex inputs",  
  "type": "docker",  
  "image": "busybox:latest",  
  "command-line": "/run/my_script.sh [THE_BOOLEAN] a-string",  
  "environment-variables": {  
    "STR_VAL": "a-string",  
    "BOOL_VAL": "[THE_BOOLEAN]"  
  },  
  "inputs": [  
    {  
      "name": "the_boolean",  
      "description": "A boolean input",  
      "type": "boolean",  
      "default-value": false,  
      "replacement-key": "[THE_BOOLEAN]",  
      "true-value": "T",  
      "false-value": "F",  
      "command-line-flag": "--bool",  
      "command-line-separator": "="  
    },  
    {  
      "name": "the_string",  
      "description": "A string input",  
      "type": "string",  
      "replacement-key": "a-string",  
      "command-line-flag": "--str",  
    }  
  ]  
}
```

Let's say this Command is used to launch a container, and let's say that no input values are passed in (i.e. only the defaults will be used). Then `the_boolean` will have value `false` and `the_string` will have no value. The command line would

become "/run/my\_script.sh --bool=F ". The environment variables would be STR\_VAL= and BOOL\_VAL=F.

Now let's say we launch a container from the Command again, and this time we pass in input values the\_boolean=true and the\_string=Hey. Then the command line would become "/run/my\_script.sh --bool=T --str Hey"

No labels

## 1 Comment

---



Charlie Moore

@John Flavin, in case you aren't aware of it, the "[XNAT output handler](#)" and "[parent input](#)" page anchors don't seem to be functional.

Powered by a free Atlassian Confluence Open Source Project License  
granted to XNAT.org. Evaluate Confluence today.