Publish and Share

Developing Tools for the Publish & Share Platform

Version 1.0
2019-08-07

Email: contact@its4land.com
Web: https://its4land.com/

Table of Contents

Introduction

The Publish & Share Platform is created to host tools from external sources. The platform therefore provides a system to envelope any tools like plugins which can be loaded to the Publish & Share server component and fill their own space in the web GUI.

Since a tool can do anything, it is necessary to put it into a standard environment, which the platform can handle in a uniform way.

The intended audience for this manual are software developers aiming to program tools for the Publish and Share platform. While tools may be developed in any programming language, this manual uses Python as the preferred language of choice to demonstrate the usage of the tool wrapper to create tools. Usage of the wrapper is optional but recommended, as it provides utility functions to interact with the platform. It is possible to start the main tool executable or script from Python as a subprocess. This functionality can be exploited to execute tools written in other languages.

First Steps

The tool framework we provide for you is developed in **Python 3.7**, so you also have to do some coding in Python. But, that does not mean that your entire tool has to consist of Python modules. You can rather write Python code that serves also as a kind of wrapper, starting external components. So, you need at least these few Python lines to embed an external executable.

The tool you are going to create will run in the context of a process, which itself is generated in the context of a project. So, as the process defines which tool will be started with which parameters and at which entry point, the first step on executing a tool will be the request for the process's data. Our wrapper does this work for you, so your tool can use the information about the running process provided by the wrapper. The wrapper also passes command line parameters to the tool, if there are any.

The start function of you tool will be determined by three statements, delivered by the process data: the tool's **name**, its **version**, and the name of the chosen **entry point**. Your Python project has to reproduce this hierarchic structure in folders: the main folder with the tool's name, below a folder with the tool's version. In the version's folder the wrapper expects a Python module named "**entrypoints.py**". This module has to define a function for each registered entry point, using the entry point's name. These entry point functions will be called by the wrapper, who passes the process data and command line parameters to them.

The content of the entry point functions is completely in your responsibility. For your convenience, we have prepared a simple base class in Python ("BasicProcessing"), which provides basic functions to start, finish and cancel a tool run. This base class shows the current state of the processing and logs it to the Public API, that's all. You can either use this class to create a subclass, which executes your specific tool, or do the work by your own.

! Important: take care that your tool (main folder) is in the system's path, so Python can find your package and import it dynamically.

Tools, Entrypoints and Processes

A ***tool*** is, abstractly spoken, a thing which helps the user or the system to manage a certain task. When it is requested, the tool can be activated to perform something and then finish its work. A tool consists of one or more software components - typically Python scripts or external executables. It has to be registered in the database to be accessible for the Public API.

Tools provide at least one ***entrypoint***, what stands for a kind of function to be called on starting the tool. So, a tool with different entrypoints can handle multiple functions. Also, on starting a tool, certain parameters can be submitted to the tool and its entrypoint, in order to configure the tool's runtime. Entrypoints are also registered in the database.

Every time when the user requests the service of a certain tool, a ***process*** is started to maintain the liftetime of a tool run. The process holds information about the project, in whose context the tool is running, name and version of the tool, a timestamp, the current process state, and the selected entrypoint. The process's metadata are stored in the database and accessible by the Public API. When the tool's work is done, the process terminates and sets its status to *finished*.

Docker Images and Containers

By default, the entire tool components are packed together into a *Docker image*, which also conatins the necessary environment. Docker images can be deployed to any host machine, on which the Docker software is installed, independent from the operation system of the host. When an image starts executing, a *Docker container* is created, what means that the execution takes place inside of an own protected environment, separated from the host system. The container will be removed after finishing the tool's work. Images and containers have a similar relationship as tools and processes.

A Docker image can contain multiple versions of the tool. If the user wants to use a certain version, he has to subscribe this version explicitely, otherwise the newest version (highest number) will be accessed inside the container.

The tool running inside a Docker container has access to the Public API and therefore indirectly also to the database, but it has only restricted access to the host's file system. So the tool should use the file system at most for temporary files. They will be deleted when the container is closed.

In consequence, the input and output is handled exclusively via the Public API. All data can be requested through the API endpoints, and can be stored there as the result of a tool's run.

The Tool Wrapper

The Tool Wrapper is a Python package to be included in the Docker image, together with the tool itself. The wrapper retrieves all available data regarding the process from the Public API and provides a simple way to handle a tool call inside a Docker container. The tool itself has to meet some requirements concerning the folder structure, the Python module to be called by the wrapper, and the entrypoint function definitions - even if it does not use any further Python scripts.

The wrapper starts automatically, when the Docker container is created.

Two things are required, to let the wrapper start its work: the URL of the **Public API** to establish a connection for accessing the database, and the UID of the **process** in whose context the tool shall run. Both parameters are expected to be set als environment variables:

Environment Variable	Description
`I4L_PUBLICAPIURL`	URL of the Public API
`I4L_PROCESSUID`	UID of the running process
`I4L_PROJECTUID`	UID of the current project (the wrapper does not use it)

There is also a configuration file `config.json` in the wrapper folder, which specifies some customizations of the wrapper. Amongst others, the URL of the Public API can be set in this file as `PublicAPI` variable. It will be used only if the environment variable is not set.

The Wrapper Modules

- `main.py`: parses the command line, loads the configuration, starts the tool wrapper
- `configuration.py`: loads the configuration file if available, provides the configuration values - including environment variables, which configure the wrapper
- `publicapi.py`: library to load specific data for the tool and the wrapper, calling the endpoints of the Public API
- `requestcontroller.py`: low level library to send requests to the public API server and handle the response, used by `publicapi.py`
- `toolclasses.py`: declares specific object classes and and loads objects from `publicapi.py` the data
- `basicprocessing.py`: base class for the tool's processing for convenience, should be derived by your own subclass

main.py

The main module has only one sole public function, "`startWrapper`". If the `autostart` **mode** is switched to True, this function will be executed automatically, as soon as the module is imported anywhere.

The `startWrapper` function starts with a check if the **Public API URL** and the **Process UID** are present. Without the URL the wrapper will not be able to load the process data and to start the tool, so it terminates immediately with the error message "URL of the Public API missing!".

If the `I4L_PROCESSUID` variable is not set, the wrapper switches to an **interactive mode**, which allows a user to enter a valid process UID (and other parameters) in the command line. The wrapper terminates, when the tool has been executed successfully, or if the user enters the quit comand `-q`.

After loading the process data from the Public API, the wrapper has the needed information about the tool, the tool's version, and the entrypoint. It tries now to import the tool package and to execute the tool by calling the entrypoint function. After the tool has finished, the wrapper also terminates.

function	result	description
`main.startWrapper()`	-	executes the wrapper as described above

configuration.py

This module holds all configuration details of the wrapper and makes them available to the tool. First, it reads the `config.json` file. If the config variable `LogfileName` is set, the wrapper's logging will start then. The tool can access the logging es well by calling the function `print`. However, logging inside a Docker container makes not so much sense...

function	result	description
`Configuration.load(configFileName=None, basePath=None)`	-	loads a config file and sets the Public API url in the `publicapi.py` module
`Configuration.configValue(key, logError=False)`	any	gets the value of a config variable, logs an error (key not found) if desired
`Configuration.publicApiURL()`	string	gets the value of the environment variable `I4L_PUBLICAPIURL`
`Configuration.setPublicApiURL(url)`	-	sets the value of the environment variable `I4L_PUBLICAPIURL`
`Configuration.processUID()`	string	gets the value of the environment variable `I4L_PROCESSUID`

function	result	description
`Configuration.setProcessUID(processUID)`	-	sets the value of the environment variable `I4L_PROCESSUID`
`Configuration.projectUID()`	string	gets the value of the environment variable `I4L_PROJECTUID`
`Configuration.setProjectUID(projectUID)`	-	sets the value of the environment variable `I4L_PROJECTUID`
`Configuration.setPublicApiURL(url)`	-	sets the environment variable `I4L_PUBLICAPIURL` to the given value (should be called before `startWrapper`)
`Configuration.setProcessUID(processUID)`	-	sets the environment variable `I4L_PROCESSUID` to the given value
`Configuration.setProjectUID(projectUID)`	-	sets the environment variable `I4L_PROJECTUID` to the given value
`Configuration.inputFilePath(fileName)`	string	combines a filename with the input path
`Configuration.outputFilePath(fileName)`	string	combines a filename with the output path
`Configuration.tempFilePath(fileName)`	string	combines a filename with the temp path
`Configuration.print(message, level=logging.DEBUG)`	-	prints the message text on the screen and writes it into the logging file, if available. Levels: `DEBUG, INFO, WARNING, ERROR, FATAL`
`Configuration.startLogging(fileName, level=logging.DEBUG)`	-	creates or opens a log file and starts logging
`Configuration.stopLogging()`	-	stops logging and closes the log file (if started before)
`Configuration.processingModule()`	string	provides the tool's mudule
`Configuration.toolFunction()`	string	provides the tool's start function for the entry point

publicapi.py

This module contains functions to access the public API, using the server URL set by the main module before.

function	result	description
`PublicAPI.serverURL()`	string	gets the URL of the Public API
`PublicAPI.setServerURL(url)`	-	sets the URL of the Public API
`PublicAPI.serverConnected()`	boolean	checks if the Public API url is set
`PublicAPI.sendGetRequest(endpoint, uid = None, add = None)`	dictionary	sends a `GET` request to a specific endpoint of the Public API; combines the URL with the endpoint name, an optional UID and an optional extension
`PublicAPI.sendPostRequest (endpoint, uid = None, add = None, data = {})`	dictionary	sends a `POST` request to a specific endpoint of the Public API; combines the URL with the endpoint name, an optional UID and an optional extension; the data to be posted is committed in the `data` dictionary parameter (JSON structure)
`PublicAPI.loadProcess(processUID)`	dictionary	loads data of a process from the API, including project data and log entries
`PublicAPI.loadLogEntries(processUID)`	list	loads log entries of a process from the API
`PublicAPI.saveLogEntry(processUID, logEntry)`	dictionary	saves a new log entry of a process
`PublicAPI.loadTool(toolUID)`	dictionary	loads tool data from the API
`PublicAPI.getEntryPoints(toolUID)`	list	retrieves all entry points of a tool from the API
`PublicAPI.loadProject(projectUID)`	dictionary	loads data of a project from the API

requestcontroller.py

This module holds all configuration details of the wrapper and makes them available to the tool. First, it reads the `config.json` file. If the config variable `LogfileName` is set, the wrapper's logging will start then. The tool can access the logging es well by calling the function `print`. However, logging inside a Docker container makes not so much sense...

function	result	description
`RequestController.createCurrentSession()`	Session	creates a new session if there is none before, returns the current session
`RequestController.resetCurrentSession()`	-	closes and removes the current session
`RequestController.sendGetRequest(url, requestData, useCurrentSession=True)`	string	sends a GET request to the given server url
`RequestController.sendPostRequest(url, requestData, useCurrentSession=True)`	string	sends a POST request to the given server url
`RequestController.sendJsonRequest(url, requestData, useCurrentSession=True)`	string	sends a POST request in JSON format to the given server url
`RequestController.sendPatchRequest(url, requestData)`	string	sends a PATCH request to the given server url
`RequestController.sendPostFormDataRequest(url, requestData, uploadFile=None)`	string	sends a POST request from a HTML form to the given server url, uploads a file if it is specified
`RequestController.sendPutRequest(url, requestData)`	string	sends a PUT request to the given server url
`RequestController.sendDeleteRequest(url, requestData)`	string	sends a DELETE request to the given server url
`RequestController.downloadFile(url, fileName, useCurrentSession=True)`	string	downloads a file from the given server url, returns the file name
`RequestController.strToDict(content)`	dictionary	converts a JSON string to a dict
`RequestController.connectNotification(func)`	string	sets the notification to an external function (signature `(title, message, exception = None)`), used for notifications

toolclasses.py

The toolclasses module contains a couple of classes which are used to represent the process object, loaded by the wrapper before starting the tool. The top level class is `Process`; the wrapper creates one single instance of this class from the process data, loaded from the Public API's `processes` endpoint by submitting the UID of the current process. This object is passed to the desired entrypoint of the tool, so that the tool can use the process information. Subordinated instances of the other classes appear in properties of the process object, like `Project`, `Tool`, `EntryPoint`, `LogEntry`.

Implemented classes:

- `Process`: represents the currently running process
- `Tool`: represents the tool specification
- `EntryPoint`: the entry point, which the process wants to trigger
- `ProcessResult`: a result specificcation of this process
- `LogEntry`: a log entry of this process
- `Project`: the prject, in whose context this process is started

`Process` **class**	**result**	**description**
`loadProcess(processUID)` `(static)`	`Process`	loads a process object with a certain ID from the Public API, including tool, log entries, process results, and project
`uid(self)`	string	ID of the process
`name(self)`	string	Name of the process
`projectName(self)`	string	Name of the current project
`toolName(self)`	string	Name of the tool
`toolVersion(self)`	string	Version of the tool
`entryPointName()`	string	Name of the entrypoint
`toolModule(self)`	string	path to `entrypoints.py` of the tool's version, to be called by the wrapper
`getToolFunction(self)`	function	tries to import the tool package and to find the entrypoint function
`project(self)`	`Project`	object showing project's properties

`Process` **class**	**result**	**description**
`tool(self)`	`Tool`	object showing the tool's properties
`entryPoint(self)`	`EntryPoint`	object showing entrypoint's properties
`parameters(self)`	dictionary	key/value pairs of the entrypoints's parameters
`logs(self)`	list of `LogEntry`	list of log entries, latest first
`addLog(self, logMessage, level, source)`	new `LogEntry`	adds a new log entry to the database and the logs list
`results(self)`	list of `ProcessResult`	list of stored process results

`Tool` **class**	**result**	**description**
`loadTool (toolUID) (static)`	`Tool`	loads a tool from the API, including entry point(s)
`uid(self)`	string	ID of the tool
`name(self)`	string	Name of the tool
`version(self)`	string	Version of the tool
`entryPointName()`	string	Name of the entrypoint
`entryPoint(self)`	`EntryPoint`	object showing entrypoint's properties
`parameters(self)`	dictionary	key/value pairs of the entrypoints's parameters

`EntryPoint` **class**	**result**	**description**
`loadEntryPoints (toolUID) (static)`	list of `EntryPoint`	retrieves all entry points of a tool from the API, including parameters
`uid(self)`	string	ID of the entrypoint
`name(self)`	string	Name of the entrypoint
`getEntryPointFunction()`	function	tries to find the function referenced by the entry point

`EntryPoint` class	result	description
`parameters(self)`	dictionary	key/value pairs of the entrypoints's parameters

`ProcessResult` class	result	description
`uid(self)`	string	ID of the result
`name(self)`	string	Name of the result
`resultType(self)`	string	result type of this process result

`LogEntry` class	result	description
`uid(self)`	string	ID of the log entry
`name(self)`	string	Name of the log entry
`message(self)`	string	message of this log entry
`level(self)`	string	level of this log entry (`DEBUG, INFO, WARNING, ERROR, FATAL`)
`sequenceNumber(self)`	int	sequence number of this log entry
`date(self)`	string	creation date of this log entry
`source(self)`	string	source of this log entry

`Project` class	result	description
`uid(self)`	string	ID of the project
`name(self)`	string	Name of the project
`description(self)`	string	description of this project
`projectDir(self)`	string	directory of this project
`areaOfInterest(self)`	string	area of interest in the project

basicprocessing.py

This module defines a class called `BasicProcessing`, which the tool developer can easily use as an abstract base class for the tool's own processing. It declares basic functions to start, finsish and abort processing, and also a `state` property, which shows the current state the processing is in. In particular, the tool developer should extend the `start` method, in order to do the tool's specific tasks. For convenience, this class provides also the process's methods.

method of the `BasicProcessing` **class**	**result**	**description**
`BasicProcessing(process, cmdParameters)`	new Processing object	Constructor of this class, same parameters as entrypoint function call: `Process` object and the list of command line parameters; sets state to initiated, if the process is valid, else state = error
`state(self)`	integer	current state: 0 = *None*, 1 = *initiated*, 2 = *running*, 3 = *finished*, 4 = *aborted*, -1 = *error*
`start(self)`	boolean	if `state` = *initiated*, sets state to *running*, creates a log entry ("Tool processing started"), should be extended
`finish(self)`	boolean	if `state` = *running*, sets state to *finished*, creates a log entry ("Tool processing terminated")
`abort(self)`	boolean	if `state` = *running*, sets state to *aborted*, creates a log entry ("Tool processing aborted")
`commandParameters(self)`	list	list of command line parameters, passed by the wrapper
`process(self)`	`Process`	the `Process` object as committed to the constructor
`processName(self)`	string	Name of this process, a combination of project name and tool name
`projectName(self)`	string	Name of the current project
`toolName(self)`	string	Name of the tool
`toolVersion(self)`	string	Version of the tool

method of the `BasicProcessing` **class**	**result**	**description**
`entryPointName(self)`	string	Name of the entrypoint
`toolParameters(self)`	dictionary	key/value pairs of the entrypoints's parameters
`toolModule(self)`	string	path to `entrypoints.py` of the tool's version, to be called by the wrapper
`logs(self)`	list of `LogEntry`	list of log entries, latest first
`addLog(self, logMessage, level, source)`	new `LogEntry`	adds a new log entry to the database and the logs list
`results(self)`	list of `ProcessResult`	list of results of this process

The Folder Structure

The Docker container provides at the runtime a certain internal folder structure, which guarantees that the wrapper is able to localize the desired tool entrypoint. The path to the entrypoint can be described as a combination of the **tool name**, the **tool version**, and the **entrypoint name**. As these parts will be combined to a valid Python path, they have to consider the name conventions for Python. This means that the names may only constist of **lower case alphanumeric characters**; especially space and dots are not allowed.

At the root level a folder with the **tool's name** is expected. This tool's folder contains subfolders named by the available **tool's versions**. In the folder name, dots must be replaced by the underscore character, e.g. the folder for version "3.1" has to be named "3_1".

Each of the version folders must contain a module called `entrypoints.py`, which consists of functions named by the **entrypoint names**. The entire folder structure looks like this example, a tool called "dummydemo":

```
/ (root folder)  | __init__.py  | wrapper (wrapper's folder)  |
dummydemo (tool's folder)    | 1_0 (version 1.0's
folder)       |  entrypoints.py (module containing the entrypoint
functions of version 1.0)       |  toolmain.py (example tool
module)    | 1_1 (version 1.1's folder)       |  entrypoints.py
(module containing the entrypoint functions of version
1.1)       |  toolmain.py (example tool module)
```

The Entrypoints Module and Functions

The version's folder can hold further Python modules to be used by the tool, if necessary (libraries etc.) - like the module `toolmain.py` in the example above. In this case the entrypoints module should add the version's folder to the system's search path, otherwise the tool's modules cannot be found:

```
import sys, ossys.path.append(os.path.split(__file__)[0])
```

Each entrypoint function declares two parameters: the **process object** and the command line **parameters**, which are passed to it by the wrapper and can be accessed as needed.

```
def <name_of_the_entrypoint> (process, parameters):
   (...)
```

Processing

If the tool consists totally or partially of Python scripts, the wrapper provides some useful features the tool developer can make use of. For accessing the Public API the module publicapi should be imported:

```
from wrapper.publicapi import PublicAPI
```

Calling the functions `PublicAPI.sendGetRequest` or `PublicAPI.sendPostRequest` the tool can retrieve data from or send data to the database. The tool does not have to care about the actual Public API, because the API calls are encapsuled by the `PublicAPI` module.

Importing the `basicprocessing` module allows to use the `BasicProcessing` as an abstract framework. The tool should declare an own subclass and at least an extension of the `start` method, in order to do the tool's specific tasks. A simple "dummy" example of the `entrypoints.py` module could look like this:

```
from wrapper.basicprocessing import BasicProcessing

def dummyentrypoint (process, parameters):    processing =
DummyProcessing(process, parameters)    if
processing.start():        processing.finish()        return
True    processing.abort()    return False

class DummyProcessing (BasicProcessing):

    def __init__(self, process, parameters):
        super(DummyProcessing, self).__init__(process, parameters)

    def start (self):
        if super(DummyProcessing, self).start():
            # do your work
            return True
        return False
```

The Demo Tool "publishandsharedemo"

We have created a simple demo tool, which can show you, how your tool could be implemented for the Publish & Share platform. It only consists of its main folder ("publishandsharedemo"), the version folder ("1_0") and two modules. And, of course, the empty `__init__.py` module in the version folder - don't forget! Also very important: the tool's main folder must be **located in the system path**; otherwise Python can't find it.

The first modulue is the mandatory file `entrypoints.py`, defining the function `demo(process, parameters)` for the entry point "demo". It also introduces a second module `toolmain.py`, This modules contains the class `MyProcessing`, which is derived from `BasicProcessing` - similar to the description above. The demo function creates an instance of this class, starts the processing and also finishes it.

The class `MyProcessing` overrides the base methods `start()`, `finish()` and `abort()`. But, they do not extend the inherited methods very much - in addition to the logging they only print messages on the console screen.

You can use this demo tool as a template to create your own tool. It's your task to extend the three methods, especially the method `start()`. Good luck!

Deploying the Tool on Publish and Share

Once the tool has been created and tested, the next steps are to package it using Docker, deploying the image on a host machine on the platform and registering it in the database.

Packaging the tool using Docker

We will not cover the basics of Docker image creation and running of containers here. Please refer to the official Docker's documentation for this. We will focus more on what the platform expects from a tool packaged in the form of a Docker image.

The tool writer can provide the tool to the platform operator/administrator in two ways:

1. As a source archive containing the Dockerfile (https://docs.docker.com/engine/reference/builder/) along with any necessary scripts & executable files. This requires that any parent image or URLs used in the Dockerfile are publicly accessible. The administrator should be able to reproduce the Docker image just by running the `docker build` command after decompressing the archive contents in a directory.
2. Prepackaged as a (optionally compressed) tar archive. This can be done in the following way for a Docker image named **publishandsharedemo**

```
docker save -o demo.tar publishandsharedemo:latest
```

The above command will create a tar archive named demo.tar which contains the image. Tar archives can be huge in size. It is generally a good idea to compress it using some format such as zip, before sharing it with the administrator. In Linux like systems, a compressed gzip tar archive can be created in one step as shown below

```
docker save publishandsharedemo:latest | gzip -c > demo.tar.gz
```

The system administrator upon receiving a packaged image can load it in the host machine in the manner shown below. If the image is compressed, it will need to be decompressed first.

```
docker load -i demo.tar


# Use the command below to load an image compressed as a gzip tar archive in
one go
gunzip -c demo.tar.gz | docker load
```

Upon listing the images using `docker images` the `publishandshare:latest` image should be available.

Registering the tool in the Publish and Share database

In order to use a tool in the Publish and Share platform, it needs to be registered in the database. The tool author needs to provide the following information to the platform administrator. Items in bold are mandatory:

- **Tool name**
- Short description
- Extended description
- Tool supplier name
- URL for tool information (e.g. a landing page or website for the tool)
- URL containing external documentation
- **Version Information**
- Release Date
- For each entrypoint
    - **Entrypoint name**
    - Entrypoint short description
    - Extended description (including parameters/arguments that can be passed to the entrypoint)
    - Entrypoint argument (i.e. the command line argument to the passed to the tool to invoke the entrypoint functionality, if needed)

> This rest of this section is mainly aimed at Publish and Share platform administrators and requires write access to the its4land database

Registration of the tool consists of adding the tool metadata to the database. Please refer to the Publish and Share concepts manual to lookup more information about the database schema. Entries need to be made to the following tables as indicated below. Green cells indicate that an entry should be added or updated.

Operation	i4ldata.t_tools	i4ldata.t_toolimages	i4ldata.t_entrypoints
Add/remove new tool			
Add/update version of existing tool			

The following SQL script is provided as an example of registering a new Image Registration tool. The docker image id is obtained by first adding the Docker image to the host machine and viewing the id shown in `docker images`

```
-- Add main tool info. Skip this if tool already exists


INSERT INTO "i4ldata"."t_tools"( "uid", "name", "description",
"longdescription", "extdescriptionurl", "supplier", "toolurl") VALUES
(uuid_generate_v4(),'Image Registration', 'GeoTIFF metadata and registration
tool', NULL, NULL, 'HansaLuftbild', NULL);

-- Add tool docker image info
-- Make sure to get the tool name, version and docker image id correct
WITH imgtab AS (
INSERT INTO "i4ldata"."t_toolimages"( "uid", "tooluid", "version",
"releasedate", "image")
VALUES (
     uuid_generate_v4(),
     (SELECT uid FROM i4ldata.t_tools WHERE name='Image Registration'),
     '1.0',
     CURRENT_TIMESTAMP,
     'b1b027c98071'
 ) RETURNING "uid")


-- Add entrypoint info corresponding to the newly add version


INSERT INTO "i4ldata"."t_entrypoint"("uid", "toolimageuid", "name",
"description", "entrypoint") VALUES
(uuid_generate_v4(), (SELECT uid FROM imgtab) , 'Register', 'Register GeoTIFF
with GeoServer','--register'),
(uuid_generate_v4(), (SELECT uid FROM imgtab) , 'Info', 'Get GeoTIFF
Metadata','--info'),
(uuid_generate_v4(), (SELECT uid FROM imgtab) , 'Check', 'Check if GeoTIFF is
Cloud Optimized', '--check'),
(uuid_generate_v4(), (SELECT uid FROM imgtab) , 'Expermaps', 'Add GeoTIFF layer
to Expermaps', '--expermaps');
```

After the database entries have been added, simply use the `/tools` API endpoint to check if the entry has been added.

```
curl -L -X GET "http://platform.its4land.com/api/tools?name=Image
Registration&version=*"
```

The response should contain information about the newly added version in addition to other versions if any.