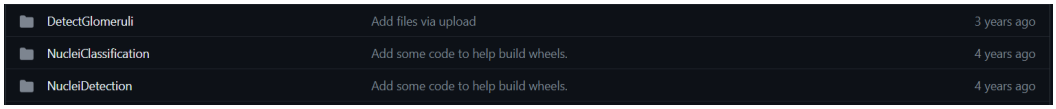
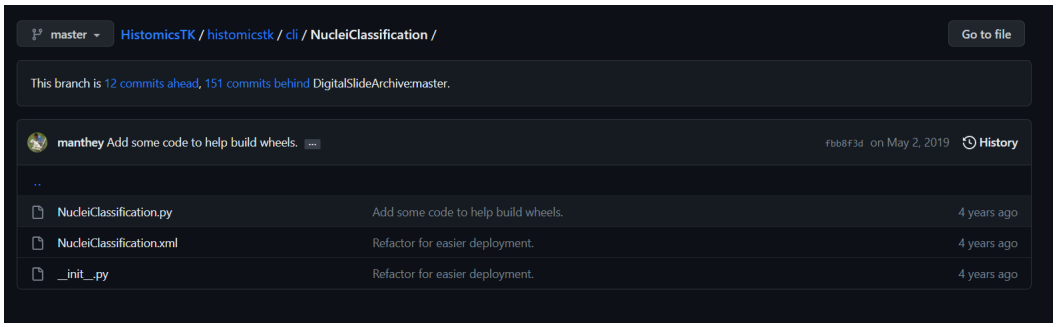
Creating a Plugin

Architecture hierarchy, the role of XML files and the CLI (Command Line Interface), and the overall workflow of a plugin:

- Folder Architecture Hierarchy

  Plugins are organized in a folder hierarchy. Each plugin has its own folder under histomicstk/cli and contains two files: a Python file and an XML file. Example: histomicstk/cli contains the plugin "Nuclei Classification," which has an associated XML file specifying inputs and outputs; NucleiClassification.xml and a main script python file; NucleiClassification.py.





- XML File and CLI

  The XML file defines the inputs and output arguments for the plugin, and the CLI (Command-Line Interface) is used to create an interface for the plugin within the container. The CLI utilizes the information provided in the XML file to specify the CLI arguments and their corresponding functionality. By utilizing the XML file, the CLI knows which arguments to expect, how to parse them, and how to pass them to the Python script for execution.

- Plugin Script:

  Each plugin has a main script (.py file). The main script is responsible for executing the specific functions of the plugin. Other scripts and utility packages can be imported as needed.

- Argument Parsing:

  The CLI script uses a utility called the CLI argument parser (CLIArgumentParser) to parse the arguments specified in the XML file. The parsed arguments are then passed to the main script for further processing.

- Input and Output Handling:

  The XML file defines the input and output parameters for the plugin. Inputs can include image files, directories, or other data types. The XML file's input specifications are used to create the corresponding user interface (UI)

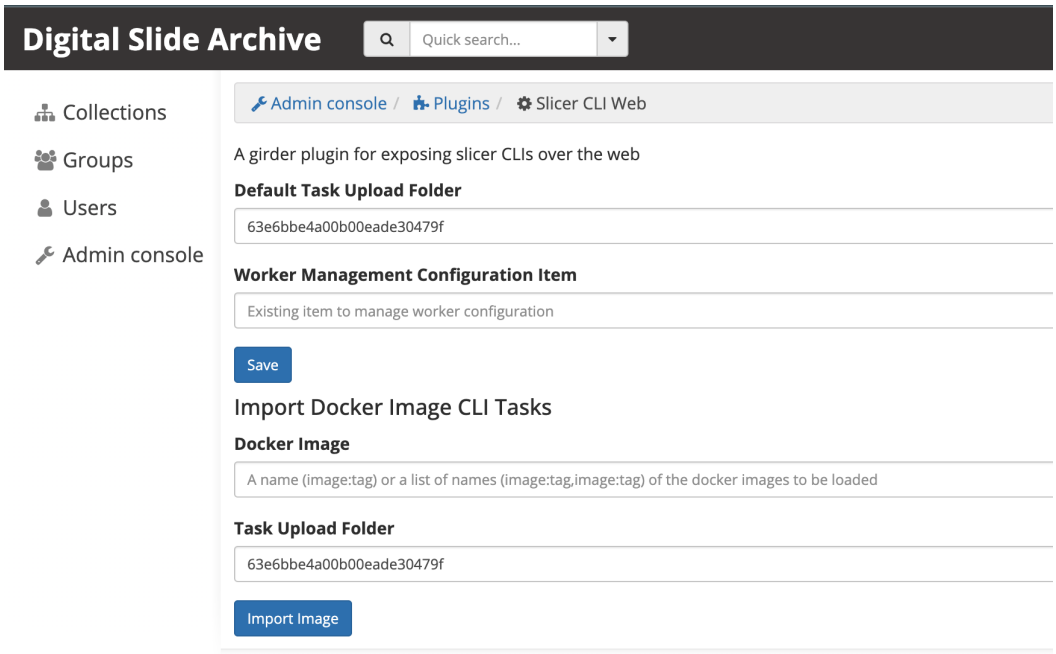
elements in the web interface. The values provided by the user through the UI are passed as arguments to the main Python script for processing.

- Output Storage:

  The output generated by the plugin is stored in a directory inside Girder, a data management system. The plugin specifies the output directory within the container. Output files, such as annotations, are saved to this directory. The saved files can be automatically displayed or accessed through the web interface.

- Docker Image and Deployment:

  After creating the Python and XML files for the plugin, a Docker image needs to be created. Upload the docker image in Adminconsle/plugins/upload-dockerimage



Designing the input interface and parameters based on the XML file format.

In the XML file the section contains all the input and output parameters.

Here are some examples of input and output parameters and how they appear on the user interface:

- Input Parameter on the XML file:

```
    <image>
      <name>inputImageFile</name>
      <label>Input Image</label>
      <description>Input image</description>
```

```
        <channel>input</channel>
        <index>0</index>
      </image>
```

<image>: This tag represents an input parameter that expects an image file as input.

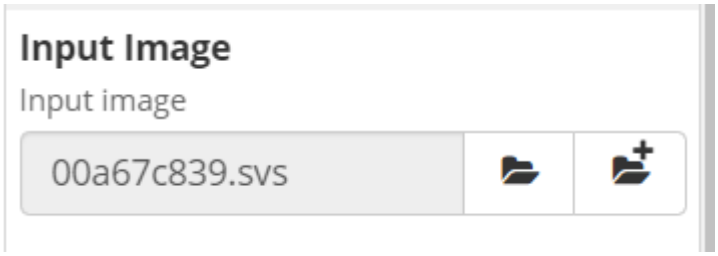
<name>: Specifies the internal name or identifier of the parameter.

<label>: It is displayed to users in the interface to identify the parameter.

<channel>: Specifies the type of the parameter. It helps in categorizing the parameters based on their role.

<index>: Specifies the position of the input/output parameter. It can be used when there are multiple input/output parameters to define the sequence in which the parameters should appear.

- Input Parameter on the UI:



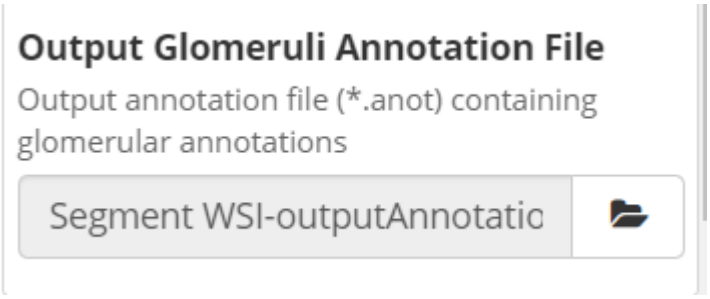
- Output Parameter on the XML file:

```
      <file fileExtensions=".anot" reference="inputImageFile">
        <name>outputAnnotationFile</name>
        <label>Output Glomeruli Annotation File</label>
        <channel>output</channel>
        <index>3</index>
        <description>Output annotation file (*.anot) containing glomerular
annotations</description>
      </file>
```

<file>: This tag represents an output parameter that expects a file as an output.

<description>: In description we can specify additional details about the parameter's purpose, format, or expected content. In this case, it clarifies that the parameter represents an output annotation file with the ".anot" extension, which contains glomerular annotations.

- Output Parameter on the UI:



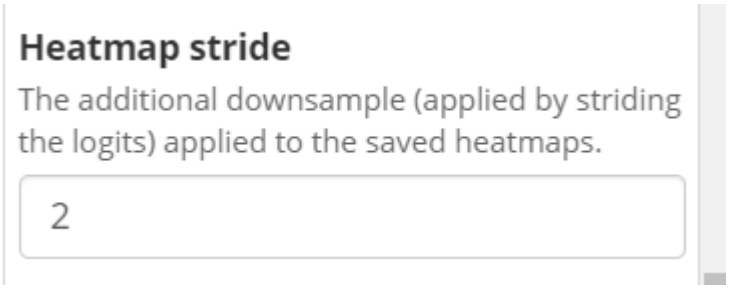
- Integer input parameter on the XML file:

```
      <integer>
        <name>heatmap_stride</name>
        <label>Heatmap stride</label>
        <description>The additional downsample (applied by striding the logits)
applied to the saved heatmaps.</description>
        <longflag>heatmap_stride</longflag>
        <default>2</default>
      </integer>
```

<default>: Specifies the default value assigned to the input parameter. If a user does not explicitly provide a value for this parameter, the default value will be used.

<longflag>: is used to define a long flag for a parameter in the executable.

- Integer input parameter on the UI:

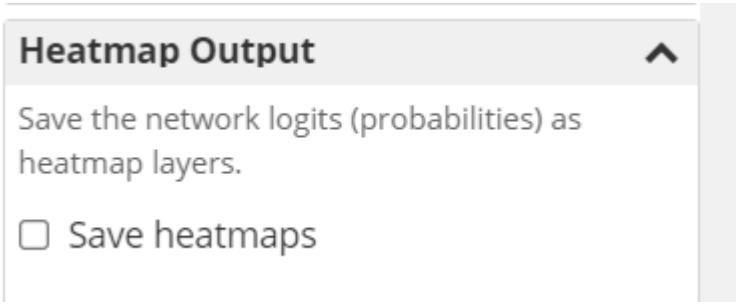


- Boolean input parameter on the XML file:

```
      <label>Heatmap Output</label>
          <description>Parameters for saving network predictions as
heatmaps</description>
          <boolean>
            <name>save_heatmap</name>
            <label>Save heatmaps</label>
            <longflag>save_heatmap</longflag>
            <description>Save the network logits (probabilities) as heatmap
layers.</description>
            <default>false</default>
          </boolean>
```

<boolean>: This tag reperesents that the parameter can either be true or false.

- Boolean input parameter on the UI:



Passing the arguments to the Python script for execution.

- Parsing

  The CLIArgumentParser() is instantiated in the main script to parse the command-line arguments. The XML file is used by the CLIArgumentParser to understand the expected arguments and their types.

```
    if __name__ == "__main__":
        main(CLIArgumentParser().parse_args())
```

- Validation

  If any of the input files/images specified in the XML file doesn't exist 'IO error' is raised.

```
    def check_args(args):

        if not os.path.isfile(args.inputImageFile):
            raise IOError('Input image file does not exist.')

        if not os.path.isfile(args.inputModelFile):
            raise IOError('Input model file does not exist.')
```

Updating Slicer_cli_list.json

slicer_cli_list.json file typically includes the names of all the CLI modules (plugins) available. After creating the .py and xml files for the plugin, add plugin name and type under histomicstk/cli/Slicer_cli_list.json.

Here is an example of slicer_cli_list.json file with 'SegmentWSI', 'TrainNetwork', 'ExportAperioXML', 'IngestAperioXML' and 'ExtractFeaturesFromAnnotations' Plugins.

```
{
  "SegmentWSI": {
    "type"    : "python"
  },
```

```
      "TrainNetwork": {
        "type"    : "python"
      },

      "ExportAperioXML": {
        "type"    : "python"
      },

      "IngestAperioXML": {
        "type"    : "python"
      },

      "ExtractFeaturesFromAnnotations": {
        "type"    : "python"
      }
    }
```

Testing entrypoints - Dockerfile

The slicer_cli_list.json file is used within the Dockerfile to verify the functionality of the histomicstk CLI entry points. The Dockerfile runs tests on the CLI entry points using the python -m slicer_cli_web.cli_list_entrypoint command followed by the CLI module name and the --help flag. These tests ensure that the CLI entry points are working correctly and that the plugins are available for use within the Docker container.

```
    RUN python -m slicer_cli_web.cli_list_entrypoint --list_cli
    RUN python -m slicer_cli_web.cli_list_entrypoint SegmentWSI --help
    RUN python -m slicer_cli_web.cli_list_entrypoint TrainNetwork --help
    RUN python -m slicer_cli_web.cli_list_entrypoint ExtractFeaturesFromAnnotations --
help
    RUN python -m slicer_cli_web.cli_list_entrypoint IngestAperioXML --help
```