# Non-Motorized Toolkit – Tool Server Communication Protocol

Version 0.2; April 2, 2014

Summary of configuration and protocol for NMTK Server and Tools

[Basic system architecture 2](#_Toc376428019)

[Configuration – Server Side 3](#_Toc376428020)

[Tool Registration 3](#_Toc376428021)

[Tool Interaction with NMTK Server 4](#_Toc376428022)

[Posting a Request to the Tool Server 5](#_Toc376428023)

[Headers for /analyze and /cancel requests 5](#_Toc376428024)

[POST data for /analyze 5](#_Toc376428025)

[POST data for /cancel 7](#_Toc376428026)

[Posting a Request to the NMTK Server 7](#_Toc376428027)

[Headers for /result and /update Requests 7](#_Toc376428028)

[POST data for /result and /update Requests 7](#_Toc376428029)

[Computing an HMAC signature 8](#_Toc376428030)

[NMTK Tool Configuration Object 9](#_Toc376428031)

[“info” section 9](#_Toc376428032)

[“documentation” section 10](#_Toc376428033)

[“input” section 10](#_Toc376428034)

[Input specification 11](#_Toc376428035)

[“ConfigurationPage” elements 13](#_Toc376428036)

[“File” elements 15](#_Toc376428037)

[“output” section 16](#_Toc376428038)

[“output” elements 17](#_Toc376428039)

[“sample” section 19](#_Toc376428040)

[Processing the NMTK Configuration 20](#_Toc376428041)

[Transmitting data to the Tool 21](#_Toc376428042)

[Receiving results from the Tool 21](#_Toc376428043)

## System architecture

The system consists of three applications that communicate through APIs. This document focuses on the interaction between the NMTK Server and the Tool Server, and is primarily intended for Tool developers.

1. Web Application: the end user web application

* Uses the NMTK API to make tool and data requests
* The initial implementation contains a Javascript-based “pure client” interacting with NMTK layer
* This interaction is not discussed further in this document

1. NMTK Server: the toolkit manager

* Presents an API for web applications to make tool and data requests (not discussed here)
* Presents an API for tools to process tool requests and return results (data, below)
* Uses the Tool Server API to make tool requests (below)

1. Tool Server: a web application that hosts one or more tools that can process tool requests

* Presents a RESTful API for the NMTK Server to make tool requests (below)
* Uses the NMTK tool processing API to communicate results and status information to the NMTK Server (above)

The initial Tool Server implementation can be used as boilerplate to construct a new Tool Server, or tools within a Tool Server. Eventually, this functionality will be moved to a Python package and a Python API will be constructed to support simple tool creation in a manner that is resistant to disruption if the Tool Server implementation is changed. An additional API is planned for tools written for the R statistical environment, and further languages are under consideration (2014/04/02).

## Configuration – Server Side

A security protocol is maintained for communications between the NMTK Server and any Tool Server that it is configured to work with.

The following information is maintained on the NMTK Server to identify a Tool Server, and must be maintained cooperatively by the NMTK Server administrator and the Tool Server administrator.

|  |  |  |
| --- | --- | --- |
| **Key** | **Description** | **Mandatory?** |
| tool\_server\_name | User-facing identification of the tool server (textual description) | Yes |
| tool\_server\_contact | Administrative record (name, email, phone, etc.) for tool server administrator (comparable to the administrative contact in a DNS domain name whois record) | No |
| tool\_server\_id | Unique identifier for the tool server | Yes |
| tool\_server\_url | The master URL for the tool server | Yes |
| shared\_secret | “Secret” shared key for HMAC.SHA1 protocol | Yes |

### Tool Registration

The tool registration process enables the NMTK Server administrator to complete the server side configuration (above) using information from the tool server administrator (*tool\_server\_contact*, *tool\_server\_url*, and *tool\_server\_name*) and to transmit required information (*tool\_server\_id*, *shared\_secret*) back to the tool server administrator.

The sequence for registration:

1. Tool server administrator submits (e.g. via email) to the NMTK Server administrator these items:
   1. tool\_server\_url
   2. tool\_server\_contact
2. NMTK Server administrator and tool server administrator agree on this item (or NMTK Server administrator unilaterally assigns):
   1. tool\_server\_name
3. NMTK Server administrator generates these items and communicates them to the tool server administrator:
   1. tool\_server\_id
   2. shared\_secret
   3. nmtk\_server\_url

The existing NMTK Server and Web Application interface contain administrative functions that allow the NMTK Server administrator to configure a Tool Server and generate the necessary security items.

The *tool\_server\_id* and *shared\_secret* are installed in the tool server by the tool server administrator (this presently requires editing a few lines in the tool server application code) and are used in subsequent protocol-based communications (described below). The *nmtk\_server\_url* is used by the tool server to allow tools to send requests back to the NMTK Server.

The information returned to the Tool Server Administrator is stored in the tool server settings and is used by the tool server to receive requests from the NMTK server and to return status messages and results.

### Tool Interaction with NMTK Server

The Tool serveridentifies hosted tools by providing top level routing information to each hosted tool. That information is used by the NMTK Server to present a list of available tools to the end user.

The NMTK Server will access the /index route at the tool server to retrieve a list of available tools. The request is submitted through a GET request using this format (which is publicly available):

$tool\_server\_url/index

where *$tool\_server\_url* is replaced with the contents of that server side configuration parameter (see above).

The result is a JSON array of text strings where each string is a route to an installed tool:

[ “tool1”, “tool2”, … , “tooln” ]

The /index route can also be used to perform a de facto “ping” on the tool server: the tool server will respond if it is available, and will (obviously) remain silent if it’s not working.

It is legal for these routes to include sub-routes (e.g. “tool1/variantA” or “tool1/variantB”), with the tool server responsible for parsing these sub-routes.

Each of the individual tools is accessed through a standard set of routes (plus optional additional routes identified in the tool configuration):

* /config
* /analyze
* /cancel (optional)

The routes are built into URIs by concatenating the server, tool root name (from the /index result), and operation as follows (“tool1” in this example is simply one entry from the /index):

$tool\_server\_url/tool1/config

The /config route is accessed via an HTTP GET request, which returns a JSON tool configuration object whose structure is documented in a subsequent section below.

The /analyze route is accessed via an HTTP multi-part POST request whose format is described in detail in the following pages. Responses to the POST request are also explained below. POST requests are encrypted as explained below, and they require that both the NMTK Server and the Tool Server use the same shared secret key.

The /cancel route is optional (so that POSTing to it may return a 404 HTTP status code, which will be handled gracefully by the NMTK Server). If available, the /cancel route kills an existing task (identified in the POST data as explained below) that may be underway on tool server, and if it is processed, returns a 200 status code with some short message (e.g. “Job ID XXXXX was cancelled by the user”).

### Posting a Request to the Tool Server

The NMTK Server initiates interaction with a tool by POSTing a request to the tool’s /analyze route (or the /cancel route)**.** When the POST request is received by the tool server, the tool server will validate the request (as described below) and issue an HTTP response status code:

* 200 – Request Accepted (if all is in order and the request was valid); could be more specific, e.g. 202
* 400 – Request rejected due to ill-formed data; could be more specific (e.g. 422)
* 409 – (“Edit Conflict”) Request rejected because the job ID has already been issued; or we could just use 400, or we could just accept with 202 and proceed to return the same job information again (or a status). Only a problem if the NMTK Server has changed the original request.

As of this document, the responses from the Tool Server are limited to 200 and 400. This document is possibly inconsistent with what has actually been implemented and will be revised as the code “settles”.

If the resource that the NMTK Server is attempting to POST to cannot be reached, or receives any HTTP status error (404, 500, etc) this will be communicated back to the user. This information will also be logged internally by the NMTK Server.

Once the /analyze request has been POSTed by the NMTK Server to the tool server, the NMTK Server will not issue further requests except possibly to /cancel a running job. The tool server will generate requests to the NMTK Server to post results or errors, or (optionally) to issue status updates.

Note that the NMTK Server will assign each job a unique identifier that will be issued one time when an /analyze request is posted, and only used again if a /cancel request is issued. Tool servers will use that job identifier in all requests posted back to the NMTK Server. When a tool server receives an /analyze request, it should verify that the unique job identifier has not previously been issued (and if it has previously been issued, then issue a 400-series HTTP error such as 409 “Edit Conflict”).

#### Headers for /analyze and /cancel requests

The /analyze and /cancel POST requests are both presented as multipart MIME posts. The POST request must include the following headers:

|  |  |
| --- | --- |
| **Header** | **Value** |
| Content type | “multipart/form-data” (literal; required) |
| Authorization | An HMAC.SHA1 signature computed from the first part of the POST data |

The /analyze route expects at least two parts in the POSTed data: (1) job specification and (2) data to be analyzed. There may be more than one data part. The formats for these are described below.

The /cancel route expects a single part in the POSTed data, which contains job identification information as described below.

#### POST data for /analyze

The /analyze route expects data POSTed in two or more parts. Currently, each of the two parts will have a content type of “application/json” and consist of JSON-encoded data, and the “charset” for each part should be “utf-8”.

The first part contains the job description. The subsequent parts will contain the actual data (files) to be processed by the analysis tool. For now, all of these are JSON objects. How many subsequent parts are present depends on the specific needs of the tool as represented in the tool configuration. There will always be at least one additional part beyond the job description.

JSON data provided must pass basic JSON lint-style validation or it will be rejected. The JSON data will conform to the input schema in the tool configuration file (see below).

First Part: Job Description

The job description is the first part of the /analyze POST request. It consists of a JSON object with this overall structure:

{

“job” : {

“tool\_server\_id” : <unique identifier for the current toolserver

on the issuing NMTK Server>

“job\_id” : “<unique job identifier>”,

“timestamp” : “<request timestamp>”

}

“analysis settings” {

<key/value pairs as described in the text>

}

}

The key/value pairs in the analysis\_settings are structured according to the input schema contained in the tool configuration file (whose format is documented below). These will communicate parameters and options describing the analysis to be performed.

Subsequent Parts: Data to Process

The data to process is placed in subsequent parts of the /analyze POST request. Each of these parts corresponds to a "namespace" (a file or page of data to analyze) in the tool configuration, and is named after the namespace (see the configuration description below). The namespace consists of a JSON sequence of key/value pairs that will be interpreted according to the configuration information in the analysis\_settings. This would typically be a table or a set of geographic features (or possibly some binary data type such as a shapefile or a geoTIFF raster) that will be analyzed.

#### POST data for /cancel

The /cancel route expects a single data part consisting of a JSON object with this format:

{

“job” : {

“tool\_server\_id” : <unique identifier for the current toolserver

on the issuing NMTK Server>

“job\_id” : “<unique job identifier>”,

“timestamp” : “<timestamp from the original request, optional>”

}

“

}

The tool server may respond with a 404 page not found error if it does not handle the /cancel request (the NMTK Server will handle that transparently). If it handles the /cancel route, the tool server will issue a response with a 200 status code and some short text message (e.g. “Job ID XXXXX was cancelled by the user”).

### Posting a Request back to the NMTK Server

The routes available at the NMTK Server to handle communications from the Tool Server are the following:

|  |  |  |
| --- | --- | --- |
| **Resource** | **Method** | **Use Case** |
| /tools/result | POST | Used by analysis tools to return (POST) results or fatal errors |
| /tools/update | PUT | (optional) Used by analysis tools to provide status updates in tool processing. \*N.B.\* this is only for status updates of \*ongoing\* analyses, not for terminal failures or errors. |
| /tools/ping | GET | (optional) This resource can be used by tool developers to test if they are able to communicate with the NMTK instance. |

These routes are appended to the NMTK Server URL (as provided by the NMTK Server administrator to the tool server administrator when the tool is registered).

#### Headers for /result and /update Requests

When a tool issues one of the resource requests to the NMTK Server, it should include the following HTTP header:

|  |  |
| --- | --- |
| **Header** | **Description** |
| Authorization | An HMAC signature that can be verified by the NMTK Server. This is provided so the server can verify that this request was made by a validated source; the checksum is computed from the first section of the POST or PUT data |

#### POST data for /result and /update Requests

Both the /result and /update requests include a JSON data object in multipart/form-data requests (like the /analyze request coming from the tool server). The result request may include multiple sections if the results are returned in several parts (e.g. a summary file, and a complete set of line items).

The first part is a JSON object that describes the request, and the second part includes data associated with the request. The first part of both the /result and the /update request must have a form field name of “config” (identified through the “name=’config’” parameter in the Content-Disposition section) and is structured as follows:

{

“job” : {

“tool\_server\_id” : <unique identifier for the current toolserver

on the issuing NMTK Server>

“job\_id” : “<unique job identifier>”,

“timestamp” : “<timestamp from original /analyze request>”

}

“status” : { <a string, one of [“results”,”error”,”status”] },

“results”: {“field”: <field containing tool result>,

“file”: <POST field containing the primary results file>,

“units”: <the units of the result field> }

}

The second part of the request will differ in structure depending on the job status. If the status is “results”, the subsequent part is a JSON object (or possibly some other file format) consistent with the output description contained in the tool configuration file (see below).

It should be noted that the tool can return the response to the server in any of the server’s recognized formats (numerous Geo-formats, xls, csv, dbf, etc.) The field specified in the result will be used to style the output for geospatial results, and the units will appear alongside a legend graphic. The bundled UI will automatically display the specified field in the output as an orderable column (but other UI’s may not do so.) This data is all tied to the datafile object (via the REST API) and is accessible using that API element.

If the status is “error” or “status”, then the second part could be plain text or HTML-encoded text intended to be presented to the end user through the web application (describing the tool situation).

### Computing an HMAC signature

The HMAC checksum is calculated on the first JSON-encoded configuration component of the payload (the part named “config”) for all PUT and POST requests. Using the private key issued to the tool and the SHA1 algorithm, an HMAC checksum is computed for the payload using the following method (Python example):

**import hmac**

**import hashlib**

**def getSignature(private\_key, payload):**

**'''**

**Generate an HMAC signature for the entire payload using the**

**private key provided to the tool.**

**This will go in the HTTP protocol standard authorization header.**

**'''**

**digest\_maker =hmac.new(private\_key,**

**payload,**

**hashlib.sha1)**

**digest=digest\_maker.hexdigest()**

**return digest**

## NMTK Tool Configuration Object

The NMTK Tool Configuration Object describes a Tool, providing information about what analysis it can perform, linking to additional documentation, specifying required input information, and describing the outputs that are returned. The Tool Configuration Object is a JSON object with the following keys (“sections”):

* “info” section
* “documentation” section
* “input” section
* “output” section
* “sample” section

Each section is described in detail below.

### “info” section

The “info” section is a JSON object that provides a descriptive overview of this version of the tool.

The “info” section is **required** and has the following keys:

* “name” – **required**: A string containing a human readable “one line” name that identifies the tool. This will be presented in lists of available tools by the end user application.
* “version” – **required**: A string stating the version of the tool. The version should be changed whenever any part of the tool configuration object is changed. There is no required format for the version, which may be a GUID, a build number, or a more typical dot-separated version string.
* “text” – **required**: A block of text that is simple (formatted) HTML and that may be displayed to an end user on the information page for this tool. The text is expected to provide a simple narrative description of the tool. Information should include the purpose of the tool, intended applications, the kind of input it requires, the kind of results it returns, the tool developer (both theoretical and practical), and other things a potential user might like to know to help them decide if using the tool is appropriate or feasible for them. **Future:** Eventually, this item may be extended to allow (or to require) a different markup language (e.g. Markdown or Wikitext), and some parsing capability may be added to allow images to be loaded from the tool server.

### “documentation” section

The “documentation” section is a JSON object that provides access to additional resources related to the theory and practical application of the tool. Many tools anticipated to be included in the NMTK are based on published books, articles and reports, and this section provides the tool developer a mechanism to serve documents through the tool server, and to provide links to external public websites.

The “documentation” section is **recommended (NOT required)**, and the section has the following keys:

* “docs” – **optional, unimplemented**: A JSON array of JSON objects, each of which has the following keys:
  + “name” – **required:** A human readable string that names the document, for use in creating a document link in the end-user web application (the document should be presented as a downloadable attachment with the indicated MIME type). The end-user application may choose to download the document when the tool configuration is loaded or updated and serve the document from its own location.
  + “url” – **required:** A URL, ***relative to the “<toolserver>/<tool>/docs” path,*** that can be retrieved via a GET request to the tool server. When this URL is requested, the document should be returned with the given MIME type (next item). It is up to the NMTK Server administrator to periodically verify that the document returned is “safe”.
  + “mimetype” – **required:** The MIME type of the document. **Future**: we may be able to make this optional if the file extension (e.g. PDF) is unambiguous.
* “links” – **optional**: A JSON array of JSON objects, each of which has the following keys:
  + “name” – **required:** A human readable string that names the document URL, for use in creating a hyperlink to the resource
  + “url” – **required:** A complete URL, typically on a server other than the tool server, that can be retrieved via a GET request. When this URL is requested, the end user application should open the URL in a new tab or page. It is up to the tool developer to decide whether these links should point to a home page, a document hosted elsewhere, or some other resource. It is up to the NMTK Server administrator to periodically check that the URL provided leads to a “safe” destination.

### “input” section

The “input” section of the Tool configuration file specifies the information that must be provided in order for the Tool to perform an analysis. The input section is structured with two goals in mind: (1) to permit the end user web application to gather the necessary input information from the Tool user comprehensibly and efficiently; and (2) to describe how the information requested from the user is to be submitted to the Tool.

Two types of parameters can be configured in the input section:

1. Files, which describe a file containing a series of features to be analyzed by the tool (notably by identifying which file attributes contain required input data for the Tool)
2. General parameters, which are named and have a single value that is either provided by the user or defaulted

How the parameter values are ultimately communicated to the Tool Server (and the Tool) is explained in the document entitled “Processing the NMTK Configuration”. In a nutshell, all the general parameters and file metadata (e.g. field identification) is submitted through a multi-part HTTP POST request, in a part called “analysis specification” that is formatted as a JSON object consisting of key:value pairs where the key is the parameter name, and the value is what the user has supplied. The actual file data is submitted in subsequent parts with names specified as described below.

#### Input specification

The “input” section is a JSON array, each of whose elements describes a “page”, or collection, of input data.

There are two types of “pages”:

1. “ConfigurationPage”
2. “File”

Implicitly a File page generates a “two-page” input structure, first selecting the file, and then specifying the field elements for that file.

The end user application will process the “input” section by first associating each file described by a “File” page with specific file data managed by the NMTK system. The end user application will then gather the specific parameters for the File (“property” maps as described below). After collecting the File parameters, the end user application will gather the general parameters requested on each of the “ConfigurationPage” pages. The pages will be presented to the user in the order the pages are listed in the configuration file (so if a “File” page is first, its parameters will appear ahead of the parameters on later pages in the end user display, but if it is last, its parameters will be requested below the other parameters). Within each page, the parameters will be requested in the order they appear in the “elements” list described in detail below.

Both types of pages (“File” and “ConfigurationPage”) contain a set of objects indexed by these keys:

* “type” (required)
* “namespace” (required)
* “name” (required)
* “label” (required)
* “description” (recommended)
* “required” (optional)
* “readonly” (optional)
* “elements” (required)
* “expanded” (optional)
* “primary” (assumed to be false unless provided for “File”; ignored for “ConfigurationPage”)
* “mime\_type” (optional for “File”; ignored for “ConfigurationPage”)

Here are detailed descriptions of each key:

* “type” – **required:** a literal string, either “File” or “ConfigurationPage”
* “namespace” – The namespace used for configuration data for this particular ConfigurationPage. This field is required, and must be unique across all ConfigurationPage/File elements.) This is the “key” used to identify this set of configuration elements when sending configuration to the server. In the case of a File configuration, this is also the form field used in the POST to identify the file associated with the File type.
* “name” – **required:** a short name identifying the page; for pages of type “File”, the value of this key is used as the name of the POSTed part that contains the corresponding file data when a job is submitted to the Tool (and should be alphanumeric with no spaces). For both types of pages (“File” and “ConfigurationPage”), the name is also used for an object in the “analysis\_settings” passed to the Tool, and the individual elements on the page are keys within that object.
* “label” – **required:** a human-readable string describing the page; this string may be used by the end user web application as a heading when presenting the parameters on the page.
* “description” – **recommended (if missing, presumed an empty string):** a longer block of unformatted human-readable text explaining general help text for parameters on this page; this string may be presented as help or introductory text to the end user ahead of the input widgets used to collect the values of the parameters
* “required” – **optional (if missing, presumed “true”):** with values “true” or “false; if true the page (or the corresponding file) must be presented to the user for input by the web application; if “false” and the page is type “File”, the file is not required to be submitted (the Tool will presumably only use general parameters to perform an analysis, effectively of a single “feature”).
* “readonly” – **optional (if missing, presumed “false”):** with values “true” or “false”; if true, all elements on the page *must*  have “default” values, and those values may be displayed as literal text, rather than through an input widget. The values must still have unique names that can be submitted through an HTTP POST request, but if the values are posted, the Tool will always use the element default value from the configuration file and will ignore any changes made by the user. This flag may be used, for example, for model parameters such as regression coefficients that the Tool developer wishes to display to the user for their information, but that the user will not be expected to change.
* “elements” – **required:** a JSON array of JSON objects, each of which has keys detailed in the following paragraphs. The primary difference between a “File” page and a “ConfigurationPage” is the type of field specification. Possible “elements” for the two page types are described separately in the next two sections.
* “expanded” – optional, defaults to true for a File type with “primary” set to true, indicates that the dialog section on the configuration page should be expanded (settings visible) by default. When false, the user would need to open the accordion box in the config to see/change the setting.
* “primary” – **optional for “File”; ignored for “ConfigurationPage”:** a Boolean value (“true”/”false”); ignored for ConfigurationPages. If true, then prompt the user with a dropdown for the file fields to select a field to use in results displays. Also suggests that this file to be “primary” in the results and its features and properties will be included in any results returned from the Tool. The default assumed value is “false” (therefore, if a file should be used for input selection, it must be set to True in at least one case.) If this property is “true” for more than one file, it will be ignored for every file after the first. Non-primary files should omit the “elements” list of objects, since the list is not supported. The tool server will upload a file that matches the “name” specified in the configuration object. As such the tool should rely on the MIME type of the uploaded file to determine the file type (not the file name and/or extension.) A list of supported file formats can be used to limit the MIME types allowed for upload.
* “mime\_type” – **optional for “File”; ignored for “ConfigurationPage”.** The value may be a single string value or a list identifying the file formats that can be received. The formats are listed using GDAL/OGR standard format codes, as present in the lists at <http://www.gdal.org/formats_list.html> (raster formats) or <http://www.gdal.org/ogr/ogr_formats.html> (vector formats). The NMTK Server must encode data in this format when it is sent to the tool. If this key is not present for a “File” page, the Tool is expected to handle geoJSON files (or a tabular file consisting of just “features” with “properties” and no “geometry” or “CRS” information). A typical application of this key would be to force the NTMK server to supply a Shapefile or some other format. The format should be in the form of a JSON list of the allowed MIME types of the files.

#### “ConfigurationPage” elements

Each element (parameter) on a “ConfigurationPage” is a JSON object with these keys:

* “name” (required) (already there)
* “label” (recommended) (already there)
* “description” (recommended) (already there)
* “type” (required) (already there)
* “default” (required if “readonly” is true, otherwise optional)
* “validation” (optional)
* “choices” (optional)
* “required” (optional)

The keys are described in detail in the following paragraphs:

* “name” - **required:** a “slug” used for form processing – must be constructed from [-a-zA-Z0-9\_]. Values posted by the user are associated with this name.
* “label” – **recommended** **(if missing, “name” will be used):** A human readable short name for this element (unformatted text) that may be displayed by the end user application.
* “description” – **recommended (if missing, presumed an empty string):** an unformatted block of text that provides help or guidance to the user on the value they should enter for this element (e.g. whether a decimal or fraction is expected, the range of legitimate values, etc.)
* “type” – **required:** a string which specifies the type of value that the Tool expects to be associated with “name”. The end user application may use the type to determine what kind of input widget to present to the end user, or to format or validate the results. Valid types include:
  + “integer” – a number with no fractional part
  + “number” – a number that may have a fractional part
  + “string” – a string of characters
  + “Boolean” – a string whose value is either true (represented as an empty string, the literal case-insensitive word “true, or the digit 0 (zero)), or false (anything that is not represented as “true”)
* “default” –**required if “readonly” for either the element or the ConfigurationPage is true; required if “hidden” is true, otherwise optional:** a value consistent with the “type” and “validation” (if any) of the element that will be used by the Tool if no user-supplied value is furnished. The “default” may be used by the end-user application to pre-fill an input widget. If the element or ConfigurationPage is “readonly”, a default value is required and will be displayed to the user. If the end user application does not post a parameter that has a “default” value (that is, the parameter is missing from the posted data), the Tool should act as if the parameter was posted with its default value. In the case of “choices” (below) the default should be the value, not the key (if an object is provided to enumerate choices.)
* **Future**: “validation” – **optional, mutually exclusive with “choices”:** provides hints to the end user application regarding the values that may be legitimately entered as the value of this element. The value of this option is a JSON object with two optional key:value pairs (that is, either one or both can be specified): “minimum”, which the smallest value to be allowed, and “maximum” which is the largest value to be allowed. Typically, this validation only makes sense for elements with a “type” of “integer” or “number”.
* **Future**: “choices” – **optional, mutually exclusive with “validation”:** provides hints to the end user application regarding valid values for this element. These may be used to construct a <SELECT> input widget. The value of this option is a JSON list either of values (a set of strings or integers consistent with the element “type”) or of key:value pairs, where the key will be posted as the value of the element, and the value will be (probably a string) presented to the user. Typically, this option makes sense for elements with type “string” or “integer”. Unless a default is provided, the first option in the choice list will be used as a default.
* “required” – **optional, presumed “false”:** if true, the field must be posted with a non-empty value. If the field has a default value, making it “required” says it must be posted even if the value is set to the default. In typical applications, a “required” element is one that does not have a sensible default, and thus may not have a “default” value in the configuration (though the field may still have a “default” value that the end user application can post back to the Tool).
* “elements” – **required:** A JSON list of the elements (fields and defaults) that must be supplied in the corresponding file in order to perform the Tool analysis. The elements are described in detail in the next section.

#### “File” elements

File elements describe the fields that are required in an input file, associating the Tool’s conventional name of the field with the field name that actually appears in the data file. They should be presented to the end user in the order listed in the File elements list. Each field that is sought in the File is described by an element that is a JSON object with the following key value pairs:

* “name” (required)
* “type” (required)
* “label” (recommended)
* “description” (recommended)
* “default” (optional)
* “required” (optional)
* “spatial\_types” (optional)
* “mime\_types” (optional)

These keys are described in detail in the following paragraphs.

* “name” – **required**: the “name” is used in the POST request to identify the part of the request that contains this file’s data, it must be present
* “type” - **required** the allowable data type for the data in this field. The Tool Server will validate the data provided by the client using this field. If the specified type is “property”, then a constant may not be specified for this field. However, if a type of “number”, “integer”, or “string” is allowed, the user will be allowed to provide a value.
* “label” – **recommended:** a short human readable string describing the field. If not supplied, it will default to “name”
* “description” – **recommended:** an unformatted block of text that provides help or guidance to the user on what this field in the file is expected to contain. It is especially important to describe validation characteristics (though in the current tool configuration specification, there is no way to formally require validation; the tool should check the parameters and return errors to the user as necessary). Typical information would include whether the field needs to be integer, number, string, whether a decimal or fraction is expected, what is the range of legitimate values, etc.
* “default” – **optional, if the type is not “property”, the value here will be the default value provided when the user decides to use a constant. This would, ideally, be the “recommended constant” for a field.**
* “required” – **optional, presumed “true”:** if false, the end user application should present the user with an option to “ignore” or “discard” this field and the Tool should be able to proceed without referring to this value. When building a Tool, marking a field as not “required” means that if the field ***is*** provided, then some “extra” analysis should be undertaken. This option may also be a ***string***, which must be distinct from any legal value that the user might enter as a default for this field (e.g. an alpha string if the field is expected to be numeric). If the option is a “string”, then “required” is presumed ***false,*** and if the literal string is posted explicitly as the “value” of this element, then the Tool will treat the posted field as if it were not supplied. It may make more sense to call this option “optional” and reverse the true/false logic so it can be presumed “false” if not provided, and usual programming shortcuts like presuming any non-empty string is “true” can be applied without confusion. Consequently, this functionality may be implemented later under a different name.
* “spatial\_types”- optional: If provided, this should be a list of allowed geometry types. Valid values are “POINT”, “LINESTRING”, and “POLYGON” . Note that the NMTK Server does not differentiate between MULTIPOINT and POINT, MULTILINESTRING and LINESTRING, and MULTIPOLYGON and POLYGON; as such the tool should be either able to understand the “MULTI” variants of these types and/or return an appropriate error message. It is recommended that your documentation reflect the allowable file types, as the tool server will limit the users choices of allowable files, but not provide reasons for limiting allowable types.
* “mime\_types” – optional: If provided this should be a list of allowed files specified by MIME type (i.e., application/json would limit the content to JSON files.) Generally speaking, if the NMTK Server understands a file type, it will be passed as a JSON structure to the tool, therefore the tool should only provide this for types that are not recognized by the server (like a image/tif, application/msword, etc.)

### “output” section

The “output” section describes (and optionally allows the user to configure) the fields that are returned in the result set when the Tool analysis is complete (presuming there are no errors), and provides metadata about the information returned by the Tool.

For simplicity in reusing code to parse the sections, the “output” section consists of a JSON list of “ConfigurationPage” objects (but NOT “File” pages). Output configuration pages should be grouped together and presented separately from input pages by the end user application. Each output configuration page can have these keys:

* “type” (optional, presumed “ConfigurationPage”)
* “name” (required)
* “label” (recommended)
* “description” (recommended)
* “readonly” (optional)
* “elements” (required)
* “expanded” (optional, default true)
* “hidden” (optiona, default false)

Here are detailed descriptions of each key:

* “type” - **optional, presumed “ConfigurationPage”:** This key is currently ignored but is documented to allow for future expansion
* “name” – **required:**  a short text string used to identify this page. Used in place of “label” if label is not specified.
* “label” – **recommended:** a human readable string describing the elements of this configuration page (e.g. global (per-file) results versus field (per-feature) results)
* “description” – **recommended:** a longer block of unformatted human-readable text explaining what kind of information will be returned by elements on this page.
* “readonly” – **optional, presumed “false”:** if true, simply display the default values of each of the “elements” to document for the user what they can expect to have returned from the Tool (and where to find it) Ignored for ‘type’ of File, unless the user chooses to use a constant (i.e., readonly does not apply when the user must choose a file mapping.)
* **Future**: “fileformat” – **optional, presumed “geoJSON”**: The value is a single string value identifying the file formats that can be received. The formats are listed using GDAL/OGR standard format codes, as present in the lists at <http://www.gdal.org/formats_list.html> (raster formats) or <http://www.gdal.org/ogr/ogr_formats.html> (vector formats). The NMTK Server must encode data in this format when it is sent to the tool. If this key is not present, the Tool is expected to return a geoJSON object (or a reduced version without geographic information).
* “elements” – **required:** a JSON list of JSON objects describing the result fields that will be returned by the Tool. Possible “elements” are described in the next section. **Future**: May become optional for a raster “fileformat” where there are no feature properties.
* “expanded” – optional: a Boolean value indicating whether the contents of this section should be displayed.
* “hidden” – Optional (default False), indicates that the field should not be shown to the user. Generally, this also implies readonly

#### “output” elements

Each output element is a JSON object that describes a field that is generated by the Tool it performs an analysis. An element may allow the user to specify an alternate name for the field, but the Tool developer may choose to use a fixed name, in which case the element may be “readonly” and the end user will simply display it to document for the user where they should look for the tool results.

Each output element has these keys:

* “name” (required)
* “label” (recommended)
* “description” (recommended)
* “type” (recommended)
* “units” (recommended)
* “readonly” (optional)
* “default” (required)

These keys are described in detail in the following paragraphs:

* “name” – **required:**  a short text string used to identify this element. Used to post alternate output field names provided by the user. Entering an alternate field name should be allowed by the end-user application provided this element is not “readonly”, and the output Configuration Page for this element is not read-only
* “label” – **recommended:** a human readable string describing the output element; if not present, “name” may be used
* “description” – **recommended:** a longer block of unformatted human-readable text explaining in greater detail what the output element will contain when the Tool analysis is performed, including the data type or units.
* **“**global” – **optional, presumed “false”:** if true, this element describes a result that will be attached to the entire returned data set (that is, not to the individual feature properties). If false, the element describes a field that will be attached to each feature that is analyzed by the Tool. A “global” output element may indicate a range of result values, a summary statistic (e.g. mean, maximum, minimum, standard deviation), a tool version or some other information applicable to all features.
* “type” – **optional, presumed readily convertible to text:** if specified, should name a standard scalar data type (“integer”, “float”, “double”, etc.). This is an advisory field that may help the end-user web application better format or present the result value.
* “units” – **optional**:if specified, contains a human-readable string describing the units of the result (e.g. “pedestrians per 12-hour period” or “bicyclists per peak hour”). This string is intended to be displayed as part of a legend or data display
* “readonly” – **optional, presumed “false”; if “true”, “default” is required; ignored for ‘type’ of File:** if true, this element’s default value will simply be displayed to the user, with no opportunity to change it. The intention is to provide a place to document what outputs are returned from the Tool even if the Tool developer has chosen not to allow the end user to configure the name (for example, providing global fields that identify the tool in different ways, such as a variable called “MyTool\_Version” whose value will be the version of MyTool that generated the results).
* “default” – **optional, defaults to “name”:** A string that is the name of the result field that will be used by the Tool if the user does not post an alternative. If the element is not “readonly”, this value may also be suggested to the user in an input widget allowing them to override it. If the element is “readonly”, this value is simply displayed.

### “sample” section

The sample section provides sample data and a tool “analysis\_setup” that can be inspected by the user and processed by the tool.

“sample”: {

“files”:[{“checksum”: <sha1 checksum of file>,

“uri”: <fully qualified URI for the file>,

“namespace”: <namespace this file is used as>,

“content-type”: <content type for the file (i.e., text/csv)>

},

...],

“config”: <job configuration JSON object>,

“description”: <string describing this sample job>

}

The end user application provides one button on the Tool Explorer page that will initiate a job to process the sample data. This button does two things: (1) load a copy of the sample data from the URL into the user’s “Library” space; and (2) set up a configuration for the job in the user’s Job space using the analysis\_setup and the file that was just loaded. Note that the file referenced using the URI must match the sha1 checksum provided, or the file will fail to load.

When the sample data is loaded in the library, it is given a standard name built from the tool name and the tool’s internal file name plus the text “sample” and optionally a date or version when the file was loaded. The same naming convention should apply to the sample job.

## Processing the NMTK Configuration

How the NMTK processes configuration inputs and outputs and transmits data to and from a tool. **This section is a work in progress, and needs some discussion and additional prototyping. The sample tool and tool server code remains the definitive documentation (4/2/2014).**

Building the “analysis\_settings” object:

* The “analysis\_settings” object consists of a JSON object with two key:value pairs:
  + “input” – **required**: a JSON object containing key:value pairs corresponding to each of the configuration pages in the tool input configuration
  + “output” – **required:** a JSON object containing key:value pairs corresponding to each of the configuration pages in the tool output configuration
* The “input” object contains key:value pairs of two types corresponding to the two types of configuration pages (ConfigurationPage and File), where the key is the “name” of the configuration page or file. The internal object structure varies for the two types of pages:
  + For a standard ConfigurationPage, the object contains key:value pairs, with each “key” being one of the configuration element names, and each “value” being the value for that element (consistent with default and validation requirements for the element laid out in the tool input configuration).
  + For a File configuration page, a more elaborate JSON object is constructed
    - The object has these two keys, which may both be present, but **at least one of which must be supplied**:
      * “global” – **optional:** a JSON object consisting of key:value pairs, where the key is one of the File element names and the value is a literal global value to be applied to each row or feature.
      * “properties” – **optional:** a JSON object consisting of key:value pairs, where the key is one of the File element names, and the value is the name of the field in the file that will furnish that element.
    - If an element name appears in both “global” and “properties”, the “global” value will be used. **Future**: We may want to reconsider this behavior, for example having a tool use the property if it exists in a feature, but using the global value if the property does not exist for a particular feature. This behavior should be globally specified here (rather than leaving it up to the tool developer) in order to avoid bugs stemming from failure to communicate between developer and user.
    - If an element name does not appear in either the “global” or “properties” section, the Tool is expected to supply the default value as documented in the input configuration, or if there is no indicated default, the value will be treated as “missing” (which will either generate an error, or perform some suitable null value processing such as dropping that variable from a regression equation).
* The “output” object contains key:value pairs corresponding to each output configuration page with the key being the “name” of the configuration page and the value being a JSON object describing the page elements.
  + The output configuration page object contains key:value pairs, one for each element in the corresponding output configuration page. Only elements that are not “readonly” (see the output configuration specification) must be included, but the end user application may choose to include any or all of the elements with their default values even if they are “readonly”.
* Note that tools which accept multiple files for input should always generate a single file for output. How the input files are combined is up to the tool (they may be concatenated, or overlaid geographically, or merged row-by-row, or merged by some key field, or whatever)

### Transmitting data to the Tool

For each “File” configuration page, if there are any “properties” (versus “global”) elements in the input (configured in “analysis\_settings”:”input”:”properties”), the properties map the expected tool name for a field into the field name in the data file.

**Future**: Certain input file types (specifically, “raster”) may not have properties. The files themselves are posted with the File “name” field identifying the posted part and the file data included as a part of the multipart POST request sent to the toolserver.

Two kinds of objects (files) are currently handled: a **geoJSON** **object** with a set of features, each of which will be analyzed by the tool in turn, or a JSON object with a structure similar to geoJSON that contains tabular data.

The **tabular object** is a degenerate geoJSON object without spatial information (bbox, CRS, etc.) or geometry. The tabular format consists at a minimum of a JSON object with a single key:value pair whose key is “features” and whose value is a JSON list of JSON objects. Each of the feature objects consists, at a minimum of a key:value pair where the key is “properties” and the value is a JSON object. Each “properties” object consists of key:value pairs where the key is the user-supplied property name (i.e. the name to which the tool property name is mapped) and the value is the value of that property for the corresponding feature.

### Receiving results from the Tool

The Tool will return a single JSON object containing the results of its analysis as a file attached to POST request back to the NMTK Server. This JSON object will have the basic structure of a geoJSON file, with one additional global key:value pair called “global\_results” (see below). A tabular file will not contain the geoJSON spatial metadata, and the features will not have a “geometry” key. The minimum returned JSON object will have these two keys:

* “global\_results” – **optional:**  present if the output includes global (whole file summary) results (such as a range of values, a minimum or maximum, an aggregate performance measure). The value of this object is a JSON object containing result elements. Each result element is a key:value pair, where the key is the result name configured by the user according to the output configuration for the Tool, or the default or immutable name provided by the tool.
* “features” – **optional:** present if the “input” section of “analysis\_settings” included any “properties” for this page and a suitable file was provided for analysis by the tool, then the “features” element will be a list of JSON objects (one per primary input feature) that contains, at a minimum, a “properties” section consisting of the same properties that were submitted in the input file for analysis.