SpecTcl REST plugin

Ron Fox

SpecTcl REST plugin

by Ron Fox

Revision History

Revision 1.0 May 1, 2015 Revised by: RF
Original Release

Table of Contents

List of Examples

Chapter 1. Introduction

This plugin provides a REST-like method to obtain information from SpecTcl. The plugin can be used to build remote control panels, web browser based interfaces to SpecTcl and alternative spectrum viewers.

This document describes

- How to incorporate the plugin into your SpecTcl at run time (in the `SpecTclRC.tcl` file).
- How to make requests of the REST interface and what to expect in return.

A few notes are important:

- The REST plugin is stateless, that is it has no concept of a client session.
- The REST plugin responses are returned in JavaScript Object Notation (JSON) format. If you intend to write a client, this document assumes you are familiar with JSON.
- The REST plugin is built on top of the tclhttpd pure Tcl web server.

How you format, issue and obtaint the response to REST operations depends on the language you program in. For example in a shell script you can use the `curl` command. In Tcl, the rest package and so on.

Chapter 2. Incorporating the REST plugin

This chapter assumes that the REST plugin has been installed in the SpecTcl installation directory tree, or at least is installed somewhere in the Tcl package load path (`auto_path`). At the NSCL, the REST plugin is normally installed in: `$SpecTclHome/TclLibs/rest`.

In this chapter we will see Tcl code that you can incorporate in your `SpecTclRC.tcl` script. This code will:

- Load the REST plugin
- Select a port on which the REST server will listen for client requests
- Start the REST server.

If the REST plugin was installed in the `$SpecTclHome/TclLibs` directory tree, SpecTcl will ensure that it is visible to the Tcl package auto load path. If not you will need to make an approprate addition to `auto_path`.

Example 2-1. Incorporating and starting the REST server

```
package require SpecTclHttpdServer          ❶
set port [::SpecTcl::findFreePort 8000]     ❷
startSpecTclHttpdServer $port               ❸
....
Starting 8080 charlie.nscl.msu.edu          ❹
```

❶ The `SpecTclHttpdServer` package contains the REST interface. Since REST is a web-based protocol, the REST plugin is built on top of a Tcl web server called tclhttpd that has custom, active pages that implement the interface.

❷ A system can run several instances of SpecTcl. Each REST server, however must listen for connections on a distinct TCP/IP port. The package provides a proc `::SpecTcl::findFreePort` that probes for an unused port starting at a base port.

This line locates the first unused port above port number `8000`, and returns that port number. You can display the port number you used at stdout or in a custom user interface so that you know how to point your clients at the server.

❸ Starts the REST server. Once this has been done REST requests can be made of the server.

❹ As the server starts it outputs a few lines of text on Tcl's stdout file (in most cases for SpecTcl this is redirected to the `TkCon` window. As this line shows, one of the bits of output identifies the port and host on which the server is running.

Chapter 3. REST requests supproted.

3.1. General Request format

Rest requests are made by performing an httpd GET operation. The actual request is a combination of the URL and the query parameters added to the end of the URL.

You can think of the URL as being made up of the following components:

Connection information

This consists of the protocol (`http:`), the name of the host in which SpecTcl is running and the port on which its REST server is oeprating.

Application name

The tclhttpd server is extensible and, although this will not be described, you could add additional services to it. The application name identifies a bundle of services that are logically related (in this case the SpecTcl REST services).

The application name is the first element of the URL's path and, for the REST plugin is always `spectcl`

Service group

The services offered by the REST server are grouped in logically related operations. For example, there is a `spectrum` group which offers operations and queries on spectra.

The service group is the path element that immediately follows the application name. The remaining sections of this chapter describe each service group.

Operation

Each service group offers a set of operations. For example the `spectrum` service group offers an operation called `list`. This is encoded in the path element of the URL immediately following the service group.

Query parameters

If you think of each operation as a function in a programming language, you can think of the query parameters as the arguments to that function In a URL, query parameters are introduced with a `?` character and consist of a set of `parameter=value` strings that are separated by the `&` character.

Suppose for example, you want to list all spectra whose names begin with `raw`. Assuming that the SpecTcl REST server is running on port 8080 in the host charlie.nscl.msu.edu, a GET operation on the following URL will return that information:

http://charlie.nscl.msu.edu:8080/spectcl/spectrum/list?pattern=raw*

Information returned from the REST request is formatted as a JavaScript Object Notation (JSON) object. All of these objects have an attribute named `status`. If the value of this attribute is `OK`, the request succeeded. If not the operation failed and the `status` value is a short reason for the failure.

Almost all returned objects also include a `detail` attribute. On failures, this can contain more detailed information about the failure. For success, this is where data associated with the request is returned. The contents of the `detail` attribute will be described with the description of each REST request.

3.2. Parameter requests

The `parameter` service group provides services related to SpecTcl parameters and tree parameters. The base URL for this service group is like:

`http://`*`host.name:port-num`*`/spectcl/parameter`

The remainder of this section describes the operations provided by this service goup.

3.2.1. list

Lists the SpecTcl raw parameters and their properties. If the parameter is also defined as a tree parameter the tree parameter properties are supplied.

The `filter` query parameter accepts a glob pattern. If it is not provided it defaults to `*` which matches all parametrs.

On success, the `detail` attribute of the returned JSON object is an array of objects, one for each parameter whose name matches the `filter`

Each object in the array has the following attributes:

`name`

The parameter name

`id`

The parameter id.

`bins` (tree parameters only)

Number of suggested bins an axis of this parameter should have.

`low` (tree parameters only)

Recommended low limit for an axis of this parameter.

`hi` (tree parameters only)

Recommended high limit for an axis of thsi parameter.

`units` (tree parameters only)

Units of measure this parameter is in.

3.2.2. edit

Modifies the properites of a tree parameter. This operation has the following query parameters:

`name`

Name of the tree parameter. The value of this parameter defaults to empty which won't usually match a tree parameter. The value of `name` must be the name of a currently defined tree parameter.

If the name is not a tree parameter a `status` of `not found` is returned with a `detail` that is the value of the `name` parameter.

`bins`

If supplied the tree parameter's bins property will be modified to the value of this parameter. If not supplied, that attribute will not be changed.

`low`

If supplied the tree parameter's low limit property will be modified to the value of ths parameter. If not supplied, that property will not be modified.

`high`

If supplied, the tree parameter's high limit property will be modified to the value of this parameter. If not supplied, that property will remain unchanged.

`units`

If supplied, the tree parameter's units property will be modified to the value of this parameter. If not supplied, that property will remain unchanged.

On success the return has a `status` of `Ok` and an empty `detail`. Several errors are possible and detected: `not found` indicates the tree parameter was not found. `command failed` indicates that a **treeparameter** failed.

3.2.3. promote

This operation promotes a simple, raw SpecTcl prameter to a tree parameter. The difference between a parameter and a tree parameter from the point of view of the REST interface is that tree parameters have additional properties that can assist you in choosing good axis limits an binning when creating spectra.

The following query parameters are recognized by the `promote` operation. Note that most of them are mandator and an error is returned if mandatory parameters are not supplied.

`name` (mandatory)

Name of the parameter. This parameter must already exist but must also not be a tree parameter.

`low` (mandatory)

Sets the low limit property of the parameter.

`high` (mandatory)

Sets the high limit property of the parameter

`bins` (mandatory)

Sets the bins property of the parameter.

`units` (optional)

Sets the units of measure of the parameter.

On success, the `detail` attribute of the returned object is empty. Several error returns are possible (`status` not `Ok`):

`missing parameter`. If one or more of the mandatory parameters is not present. In that case the `detail` field is the name of one of the missing parameters.

`not found`. If the `name` parameter value does not correspond to an existing parameter. The `detail` is the name of the parameter.

already treeparameter. If the parameter named is already a tree parameter. The `detail` is the name of the parameter.

command failed. If the **treeparameter -create** command failed. The `detail` field is the error message emitted by the failing command.

3.3. spectrum requests

The `spectrum` service group provides operations that revolve around spectra in SpecTcl. This service group allows you to request information about spectra, create and delete spectra.

The base URL for this service group is

`http://`*`host.name:port-num`*`/spectcl/spectrum`

3.3.1. list

Produces information about the spectra whose names match a pattern with glob wildcards characters. The `filter` optional parameter provides the value of this pattern. If not supplied it defaults to `*` which matches all spectra.

The `detail` attribute of the returned object is an array of objects, one object for each matching spectrum. Each of these objects has the following fields that, taken together, describe the spectrum:

`name`

The spectrum name.

`type`

The SpecTcl spectrum type code.

`params`

The SpecTcl spectrum parameter definitions. This is provided as an array of strings.

`axes`

This is an array of objects that describe the SpecTcl axes. Each object has the attributes `low` for the axis low limit, `high` for the axis high limit and `bins` for the number of bins on that axis.

`chantype`

The SpecTcl channel type code (e.g. `long`).

3.3.2. delete

Deletes a single spectrum. The `name` parameter provides the name of the spectrum to delete.

On success the `detail` field is empty. The non success returns include:

missing parameter. The `name` parameter was not supplied.

not found. There was no spectrum with the name provided.

command failed. The **spectrum -delete** command failed. `detail` contains the error message returned by the command.

3.3.3. create

Creates a new spectrum. The spectrumto be created is defined by the query parameters, most of which are mandatory:

`name` (mandatory)

The name of the new spectrum. This must not be the name of an existing spectrum.

`type` (mandatory)

The spectrum type code e.g. `1` for a 1-d spectrum.

`parameters` (mandatory)

The parameters expressed as a space separated list.

`axes` (mandatory)

The axis specifications. This is a space separated list of SpecTcl axis specifications e.g.: `{0 1023 100} {0 1023 200}`.

`chantype` (optional)

The spectrum channel type, defaults to `long`.

On success, the `detail` field of the returned objexct is empty. Several types of failures are directly tested for:

`missing parameter`

A mandatory parameter was not supplied.

command failed

The **spectrum -create** command failed. `detail` is the error message from that command.

3.3.4. clear

Clears the counts in a set of spectra. The `pattern` parameter supplies a glob patternt hat the cleared spectra match. `pattern` defaults to `*` which clears all spectra

This operation has no failure returns. The worst thing it can do is to clear nothing (no matching patter). The `detail` attribute of the returned objetct is also empty

3.3.5. contents

Returns the contents of the a spectrum. The contents of a spectrum are sufficient to reconstruct the spectrum channels and overflow statistics. The only parameter is the mandatory `name` parameter which is the name of the spectrum to return.

There are some significant differences in what can be returned. These differences depend on the underlyng spectrum type (1 or 2d) and the version of SpecTcl the REST server is running on (4.0 or earlier).

Prior to SpecTcl 4.0, spectra did not maintain over/undeflow statistics. The REST interface for those version of SpecTcl produced an array of channel objects for the channels with non-zero counts as the value of the `detail` attribute:

Channel objects have the following attributes:

`x`

: The X channel number.

`y` (2-d spectra only)

: The Y channel number.

`v`

: The number of counts at that channel.

Beginning with SpecTcl version 4.0, the `detail` attribute became an object with the attributes `channels` and `statistics`. The `channels` attribute contains the array of channel objects and the `statistics` attribute contains an object that describes the over/underflow statistics for the spectrum. Its attributes are:

`xunderflow`

: The number of times an X parameter value was below the X axis low limit.

`yunderflow` (2-d only)

: The numer of times a Y parameter value was below the Y axis low limit.

`xoverflow`

: The number of times an x parameter was above the X axis high limit.

`yoverflow` (2-d only)

The number of times a Y parameter value was above the Y axis high limit.

Two optimizations are performed. Channel objects are only present for non-zerof channels. 2d spectra may be emitted with the `Content-encoding deflate`. In that case the special header `Uncompressed-Length` provides the number of bytes required to hold the uncompressed JSON after defation.

3.4. gate requests

This service group provides operations on Gates. The base URI of this service group is:

http://*host.name:port-num*/spectcl/gate

3.5. list

List the definitions of gates whose names match a pattern with glob wildcards. The `pattern` parameter provides the pattern. If `pattern` is not specified, it defaults to `*` which matches all gate names.

The `detail` field of the reply is an array of objects. Each objects describes a gate who's name matches the `pattern`. The actual shape of the objects can vary quite a bit depending on the gate type.

All gate types have the following attributes in their objects:

`name`

Name of the gate.

`type`

Gate type code.

The remaining attributes depend on the value of the `type` attribute.

3.5.1. Compound gates (+ * -)

Compound gates are gates that are formed from other gates. The gates a compound gate is formed from are called *component gates*. The + gate is true when any of the components is true. + means `Or`. The * get by contrast is an `And` gate; it reuires all of its component gates to be true. Finally the -, or `Not` gate has only a single component gate and is true when that gate is false (logical complement).

The remaining attribute of a compound gate is called `gates` and consists of an array of gate name strings.

3.5.2. Slice (s)

Slice gates check a single parameter against a low and high limit and are true for events when the parameter is inside the limits. The `parameters` attribute is an array that contains the name of the single parameter. The plural and array are used so that we can have some uniformity for gates that depend on parameters.

The `low` and `high` attributes are the low and high limits of the slice.

Below is the JSON that might be returned for a slice gate named `cut`:

```
{
    "status" : "OK",
    "detail" : [{
        "name"       : "cut",
        "type"       : "s",
        "parameters" : ["event.raw.00"],
        "low"        : 29.710001,
        "high"       : 52.480003
    }]
}
```

Note that even though only one gate has matched `detail` is an array. Similarl note that `parameters` is an array.

3.5.3. Gamma slice (gs)

A gamma slice gate is much like a slice gate. Gamma slice gates, however have several parameters and are true whenever one of those parameters is in the gate. When used as gates they are very much like an or of a bunch of identical slice gates, one on each parameter. Their value, however, is when used as folds in a gamma spectrum.

The format of this gate is identical to that of a slice gate, however there really is more than one parameter in the `parameters` array for example:

```
{
    "status" : "OK",
    "detail" : [{
        "name"       : "gs",
        "type"       : "gs",
        "parameters" : ["event.raw.00","event.raw.01","event.raw.02","event.raw.03"],
        "low"        : 194.369995,
        "high"       : 368.279999
    }]
}
```

3.5.4. Simple 2-d gates (`b c`)

Simple 2-d gates are *band* and *contour* gates. These gates are defined on two parameters and have an array of points that define an area in the two dimensional space defined by those parameters.

The `parameters` field contains the name of the x followed by the name of the y parameter. The the `points` attribute contains an array of objects that contain `x` and `y` attributes:

```
{
    "status" : "OK",
    "detail" : [{
        "name"       : "c",
        "type"       : "c",
        "parameters" : ["event.raw.00","event.raw.01"],
        "points"     : [{
            "x" : 182.821289,
            "y" : 280.725586
        },{
            "x" : 512.499023,
            "y" : 180.823242
        },{
            "x" : 675.339844,
            "y" : 340.666992
        },{
            "x" : 665.349609,
            "y" : 514.497070
        }]
    }]
}
```

The name of this gate is `c` and it is a contour (`type` = "c"). If this were a band, the `type` would be "b". The only difference between a band an a contour is what the points mean. In a contour, the points define

a closed figure and the gate is true when the x/y parameters are both present and inside the figure. For a band, the gate is true when both x/y parameters are present and the point is below the figure defined by the points.

Contours have an implied last point that is the same as the first point. That is the last point in the contour connects to the first point to close the contour.

3.5.5. Gamma bands and contours (`gb gc`

These gates are like bands and contours however they have more than two parameters. All combinations are tried against the gate and if any are true the gate is true. Once more these are more valuable when used to define a fold.

The JSON is pretty much the same as for bands and contours, however there will can be more than two `parameters`:

```
{
    "status" : "OK",
    "detail" : [{
        "name"       : "gammacont",
        "type"       : "gc",
        "parameters" : ["event.raw.00","event.raw.01","event.raw.02","event.raw.03","event.
        "points"     : [{
            "x" : 122.879997,
            "y" : 409.600006
        },{
            "x" : 440.320007,
            "y" : 204.800003
        },{
            "x" : 860.159973,
            "y" : 450.559998
        },{
            "x" : 696.320007,
            "y" : 665.599976
        }]
    }]
}
```

3.5.6. Bit mask gates (`em am nm`)

These gates are defined on a single parameter that is assumed to contain integer. The `em` (equal mask) is true when the parameter is equal to the mask. The `am` (and mask) is true when the parameter, bit-wised anded with the mask is equal to th emask (all the bits set in the mask are set in the parameter). The `nm` is true if the parameter bit-wise anded with the bit-wise complemnt of the mask is equal to the mask.

In addition to the `parameters` attribute the description contains `value` which is the value of the mask:

```
  {
    "status" : "OK",
    "detail" : [{
        "name"       : "mask",
        "type"       : "em",
        "parameters" : ["event.raw.00"],
        "value"      : 0X1234
    }]
}
```

3.6. delete

Deletes a gate named by the `name` parameter. The `name` parameter is required and must name an existing gate.

The `detail` attribute of the returned object is empty. Note that in SpecTcl, gates are never actualyl deleted, the are turned into gates that are always `false` instead. This allows you to know and reason about the behavior of gates that depend on the deleted gates.

3.7. edit

This operation creates or re-defines (edits) an existing gate. The `name` parameter specifies the name of the gate. If the gate does not exist it will be created. Gates can not only be edited to change their points. Any aspect of a gate can be modified (except the name of the gate).

The `type` parameter determines the new type of the gate. The remaining parameters depend on the value of this type.

3.7.1. Compound gates (`*` `+` `-`)

These gates only require a `gate` parameter whose value is a Tcl list of component gates.

3.7.2. Constant gates (`T F`)

These are gates that are either always TRUE or always FALSE. They are normally used as placeholders. For example SpecTcl itself replaces a deleted gate with a FALSE gate in order to ensure that any dependent gates will continue to function in a predictable manner.

These gates do not require any additional parameters.

3.7.3. Bands and Contours (`b c`)

These gates need both an `xparameter` and a `yparameter` value to describe the X and Y parameters to check against the definition. In addtion, these gates need several instances of `xcoord` parameters (x position of gate points) and `ycoord` parameters (y position of gate points).

If there are differing number of x and y coordinates (you really should avoid that), only those points for which the x/y coordinates were both specified will be used. A band must have a minimum of two points while a contour requires at least three points. For contours, there is an additional implied line segment joining the last and first points to close the figure.

3.7.4. Bit mask gates (`em am nm`)

Bit mask gates require a `parameter` and a `value` parameter. The `parameter` indicates which event parameter will be checked againts the `value` which is a bitmask.

3.7.5. Slice gates (`s`)

Slice gates require three query parameters:

`parameter`
: Provides the name of the parameter checked against the slice.

`low`
: Low limit of the slice.

`high`
: High limit of the slice.

Note that if `low` is larger than `high`, SpecTcl will reverse these parameters to create a sensible gate.

3.7.6. Gamma 2d gates (`gb gc`

These neewd the follwoing additional parametrs:

`parametrer`

Must appear t least twice and can appear several times. Each appearance adds a parameter to the list of parameters the gate will be checked against.

Gamma gates are true for each pair of parameters that are in the gate. They are more useful as folds than as actual gates.

`xcoord`

Provides an X coordinate for a gate point. These are orderd and will be matched up with `ycoord` values (see below). A band requires at least two of these and a contour at least three.

`ycoord`

Provides the y coordinate for a gate point. These are ordered and will be matched up with corresponding `xcoord` values. If there is an excess of either `xcoord` or `ycoord` values, the excess is discarded.

A band requires at least two points while a contour requires at least three.

3.7.7. Gamma slice (`gs`)

As for a slice gate a `high` and `low` parameter are required to set the limits for the gate. For Gammma slice gates, however you can have more than one `parameter` query option. Each `parameter` adds another parameter to the list of parameters checked against the gate.

Gamma slice gates are true if any of the parameters are within the slice. As such, they are more useful as folds where they can cut down the set of parameters that can actually increment spectra.