# 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

**1. Introduction**..........1

**2. Incorporating the REST plugin**..........2

**3. REST requests supported.** ..........4

3.1. General Request format ..........4
3.2. Parameter requests ..........5
3.2.1. list ..........5
3.2.2. edit ..........6
3.2.3. promote..........7
3.2.4. create..........7
3.2.5. listnew..........8
3.2.6. check..........8
3.2.7. uncheck..........8
3.2.8. version ..........9
3.3. spectrum requests..........9
3.3.1. list ..........9
3.3.2. delete..........10
3.3.3. create..........10
3.3.4. clear ..........11
3.3.5. contents..........11
3.4. gate requests..........12
3.5. list..........12
3.5.1. Compound gates (`+ * -`)..........13
3.5.2. Slice (`s`) ..........13
3.5.3. Gamma slice (`gs`) ..........14
3.5.4. Simple 2-d gates (`b c`) ..........14
3.5.5. Gamma bands and contours (`gb gc` ..........15
3.5.6. Bit mask gates (`em am nm`)..........16
3.6. delete ..........16
3.7. edit..........17
3.7.1. Compound gates (`* + -`)..........17
3.7.2. Constant gates (`T F`)..........17
3.7.3. Bands and Contours (`b c`)..........17
3.7.4. Bit mask gates (`em am nm`)..........17
3.7.5. Slice gates (`s`) ..........18
3.7.6. Gamma 2d gates (`gb gc` ..........18
3.7.7. Gamma slice (`gs`) ..........19
3.8. Gate Applications...........19
3.8.1. Applying a gate to a spectrum. ..........19
3.8.2. Listing gate applications..........20
3.9. Attaching data sources (New in 5.5) ..........21
3.10. Binding Spectra to Display Memory (new in 5.5) ..........22
3.10.1. /sbind/all ..........22
3.10.2. /sbind/sbind..........22
3.10.3. /sbin/list ..........23
3.11. Accessing the fit command (new in 5.5)..........23

3.12. REST interface for the fold command (new in 5.5) ........ 25
3.13. Accessing the channel command (new in 5.5) ........ 26
3.14. Projecting spectra (new in 5.5) ........ 27
3.15. Spectrum Underflow and Overflow Statistics (new in 5.5) ........ 28
3.16. Access to the treeevariable command (new in 5.5) ........ 29
3.17. Accessing the filter command (New in 5.5 ........ 30
3.18. Accessing the integrate command (new in 5.5) ........ 32
3.19. Accessing the SpecTcl parameter command (new in 5.4) ........ 34
3.20. Accessing the pseudo command (new in 5.5) ........ 36
3.21. Access to the sread command (new in 5.5) ........ 37
3.22. Access the ringformat command (new in 5.5) ........ 38
3.23. Accessing the unbind command (new in 5.5) ........ 38
3.24. Accessing the ungate command (new in 5.5) ........ 39
3.25. Accessing the swrite command (new on 5.5) ........ 39
3.26. Controlling data anlaysis (new in 5.5) ........ 40
3.27. Accessing the roottree command (new in 5.5) ........ 40
3.28. Accessing the pman comman (new in 5.5) ........ 42
3.29. Accessing the evbunpack command (new in 5.5) ........ 45
3.30. Excuting arbitrary commands in SpecTcl (new in 5.5) ........ 47
3.31. Traces (New in 5.5). ........ 48

# List of Examples

2-1. Incorporating and starting the REST server ........................................................................................2

# 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.

For SpecTcl 5.3 and later, starting the REST interface is much simpler than described in the remainder of this document. This is because The REST In SpecTclInit.tcl simply set the `HTTPDPort` variable in `SpecTclInit.tcl` to the desired server port. Note that:

- Port numbers below 1024 are privileged ports and cannot be used by programs that are not running as `root`
- If the desired port is in use, SpecTcl will search for a nearby unused port.
- When the REST interface starts, the actual port number selected will be output to the terminal on which SpecTcl was started.

The remainder of this section only applies to versionf of SpecTcl earlier than 5.3.

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 supported.

## 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.

### 3.2.4. create

Provides direct access to the SpecTcl **treeparameter -create** command. This command can create a new parameter if there is no existing parameter. The REST interface returns nothing. It accepts the following query parameters:

`name` (mandatory)

    Name of the new treeparameter. If there isn't an underlying parameter with this name it will be created.

`low` (mandatory)

    Suggested spectrum low limit for this parameter

`high` (mandatory)

    Suggested spectrum high limit for this parameter.

`bins` (mandatory)

    Suggested spectrum binning for this parameter given for its recommended low and high values.

`units` (optional defaults to "")

    Units of measure of the parameter

### 3.2.5. listnew

Lists tree parameter that have been created by **treeparameter -create** note these can be created by local commands (from scripts e.g.) of from REST requests that can come from any client

On successful completion, the `detail` attribute will be an array of strings that are the created tree parameters.

### 3.2.6. check

Returns the state of a tree parameter check flag. The required query parameter `name` is the name of the tree parameter to operate on.

The check flag indicates if a tree parameter has been created or modified since SpecTcl started or since the flag was cleared. It can be used to reduce the amount of information that needs to be written to saved settings files in the event there are a large number of tree parameter definitions.

### 3.2.7. uncheck

Clears the check flag of the treeparameter whose name is the `name` query parameter.

### 3.2.8. version

The `detail` attribute of the response to this request is the treeparameter implementation version. This is a string, not a number.

## 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`).

`gate`

The gate applied to the spectrum. Note that all spectra have gates applied. They start out with the special `-T-` gate applied which is a true gate.

### 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.

With SpecTcl 5.5, the optional query parameter `compress` (which defaults to in order to perserve compatibility), can control if compression is done. This was done to cater to an error in the tcl http package that throws a `data error` in the even the data are binary.

## 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.

## 3.8. Gate Applications.

The `/spectcl/apply` set of URLs provide access to the SpecTcl **apply** command. This set of URLs can

- Apply a gate to a spectrum. Applying a gate to a spectrum only allows that spectrum to be incremented for events that make that gate `true`
- List the gate applied to a spectrum. Note that to make the SpecTcl logic regular, all Spectra actually have exactly one gate applied to them. Always.

  When Spectra are initially created, the gate `-TRUE-` is applied which is a True gate. True gates are true regardless of the event contents.

### 3.8.1. Applying a gate to a spectrum.

To apply a gate to a spectrum the URI of the form:

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

Should be used. This URI has two required query parameters:

gate

  The name of the gate to apply to the spectrum.

spectrum

The name of the spectrum to which the gate is applied.

The response from the server, on success, is

```
{
    "status" : "OK",
    "detail" : ""
}
```

Any other value for the status attribute is an error return message and the detail attribute furthere clarifies the error.

### 3.8.2. Listing gate applications

To list the gates applied to a spectrum a URI of the form.

**http://*host.name:port-num*/spectcl/apply/list**

Should be used. This URI accepts a single optional query parameter `pattern` whose value is a glob pattern string that filters the output by spectrum names that match that pattern. If this parameter is omitted it defaults to `*` which lists the gates applied to all spectra.

The result is a JSON object with attributes `status` and `detail`. The `status` attribute value will always be a the string `OK` as this URI produces no errors.

The `detail` attribute value is an array of JSON objects. Each element of the array describes a single gate application for a matching spectrum. The attributes of these objects are:

`spectrum`

The value of this attribute is the name of a spectrum.

`gate`

The value of this attribute is the name of the gate applied to this spectrum. Note again, that every spectrum has a gate applied to it, however some spectra may have a `true` gate applied to themto effectively un gate them.

Sample return value:

```
{
    "status" : "OK",
    "detail" : [{
        "name" : "some.spectrum",
        "gate" : "some.gate"
    },
    {
        "name" : "ungated.spectrum",
        "gate" : "-TRUE-"
    }
    ]
}
```

## 3.9. Attaching data sources (New in 5.5)

The SpecTcl **attach** command can be accessed using the `/spectcl/attach` URL domain.

**http://host:port/spectcl/apply/apply?type=srctype&source=src[&size=bsize&format=fmt]**

Attaches a new data source to SpecTcl. Query parameters are:

`type`

The type of data source. Currently this is one of `file` to read data from a file or `pipe` to read data from a program on the other end of a pipe.

`source`

The source string expected by the **attach** command for the specified source type. For `file` data sources, this is just the path to the file to read. Note that since SpecTcl's directory is not known it's recommended the full path be provided. For `pipe` data sources this is the full command string.

`size`

Optional query parameter that, if supplied, sets the blocking factor for reads from the data source. This defaults to `8192`

`format`

Optional query parameter that, if supplied, sets the data format. Acceptable values are:

`ring`

The data are ring items, see, however the `/spectcl/ringformat` domain to select the version of ring items. This is the default value `nscl`

Data are fixed length 8192 byte buffers from NSCLDAQ before version 10.0

`jumbo`

Data are fixed length buffers longer than 128K bytes from NSCLDAQ prior to version 10.0. This was used for some MoNA experiments that took trace data and therefore needed longer buffers. The difference is an extension to the 16 bit used blocksize (in 16 bit words) that supports these sizes.

`filter`

The data are XDR filter data. Note that SpecTcl must have been configured to analyze filter data.

Note that attaching does not imply starting data analysis. That requires using the `/spectcl/start` request.

**http://host:port/spectcl/apply/list**

The `detail` part of the JSON returned is the of the form `SourceType: string` where `SourceType:` is either `File` or `Pipe` and the string is the data source string, that is the filename for `File:` and the command on the other end of the `Pipe:`.

## 3.10. Binding Spectra to Display Memory (new in 5.5)

The `/sbind` domain provides access to the **sbind** SpecTcl command. This there are three subdomains:

### 3.10.1. /sbind/all

URLs of the form:

**http://host:port:/sbind/all**

Bind all spectra to display memory. Nothing of value is returned in the `detail` field of the reply data.

### 3.10.2. /sbind/sbind

URL's of the following form:

**http://host:port/sbind/spectrum?spectrum=name1[&spectrum=name2...]**

Bind all spectra named in all instances of the `spectrum` query parameter to display memory. Nothing of use is returned in the `detail` part of the JSON reply data.

### 3.10.3. /sbin/list

URL's of the form:

**http://host:port/sbind/list[?pattern=glob]**

Return information about the bindings of spectra that match the `pattern` query parameter interpreted as a glob spectrum name match string. If omitted the pattern matched defaults to `*` which matches all spectra.

The `detail` attribute of the JSON returned data is an array of objects with the following attributes:

`spectrumid`
: The id of the bound spectrum.

`name`
: The name of the bound spectrum.

`binding`
: The shared memory binding id.

Note that spectra which are not bound but match the pattern will not appear in the array.

## 3.11. Accessing the fit command (new in 5.5)

The `/spectcl/fit` domain provides access to the SpecTcl **fit** command. **fit** provides you with the ability to fit spectra to arbitrary functions. The built-in `linear` and `gaussian` fit types are provided and the programming manual describes how to extend the set of fits that are provided by this command.

The URL below:

**http://host:port/spectcl/fit/create?name=fitname&spectrum=specname&low=lochan&high=hichan&type=fittype**

Provides the capability to create new fits (**fit create**). The query parameters are as follows:

`name`

Name to be associated with the fit. This will be used when referring to the fit in the future.

`spectrum`

The spectrum on which the fit is to be computed.

`low, high`

The *channel* coordinates over which the fit is to be computed. At present, the fit is done in channel coordinates.

`type`

The type of the fit to perform. The built in fit types are `linear` and `gaussian` which have pretty obvious meanings. Applications may extend this set of functions.

The URL

**http://host:port/spectcl/fit/update[?pattern=glob]**

Provides the ability to update the set of fits whose names match the glob pattern in the query parameter `pattern`. As spectra accumulate, fit data will be outdated. This allows the fit information to be recomputed to match current data.

If the pattern is not provided it defaults to `*`

The URL

**http://host:port/spectcl/fit/delete?name=fitname**

provides the ability to delete the fit object named after the contents of the query parameter `name`. Once deleted, the fit cannot be recovered.

URLs of the form

**http://host:port/spectcl/fit/list?pattern=glob**

Allows you to get information about the fits whose names match the `pattern` query parameter. The pattern is treated as a glob pattern and, if not supplied, defaults to `*`

The result is a `detail` attribute that is an array. Each element of the array is an object that describes a matching fit. The atributes of these objects are:

`name`

Name of the fit.

`spectrum`

Name of the spectrum the fit is computed on.

`type`

Type of the fit being computed.

`low, high`

Fit limits in spectrum channel coordinates.

`parameters`

An object that contains the fit parameters. The attributes of this object will depend on the actual fit type. Likely there will always be a `chisquare` attribute that describes the goodness of fit.

## 3.12. REST interface for the fold command (new in 5.5)

The fold command has three sub opterations that:

1. Fold a list of gamma spectra on a gamma gate.
2. Remove a fold from a gamma spectrum .

3. List the folded spectra (that match a glob pattern).

The URL below applies folds to spectra:

**http://host:port/spectcl/fold/apply?gate=gammagate&spectrum=specname1[&spectrum=specname2...]**

The `gate` query parameter specifies a gate which must be a gamma gate. One or more `spectrum` query parameters specify the spectra to be folded with that gate.

the `spectcl/fold/apply` URLs do not return anything useful other than the status.

The URL below lists folded spectra that match the `pattern` query parameter (treated as a glob pattern). If the `pattern` query parameter is not supplied, it defaults to `*`

**http://host:port/spectcl/fold/list[?pattern=globpattern]**

It is important to note that the pattern matches *spectra* not gates. The `detail` part of the return data is an array of JSON objects. Each object has the following attributes:

`spectrum`

Name of a folded spectrum.

`gate`

Name of the gate used to fold the spectrum.

Spectra without folds will not be listed in the array.

The following URL is used to unfold a spectrum:

**http://host:port/spectcl/fold/remove?spectrum=spectrum-name**

The `spectrum` query parameter specifies the name of the spectrum from which to remove the fold. Nothing useful other than the `status` attribute of the reply is returned on success.

## 3.13. Accessing the channel command (new in 5.5)

Spectrum channel values can be inspected or set using this interface Note that the spectrum contents interface may be a more performant method to get bulk spectrum information.

URLS of the type below can set the value of a spectrum channel:

**http://host:port/spectcl/channel/set?spectrum=name&xchannel=x&value=value[&ychannel=y]**

The `spectrum` query parameter specifies the name of the spectrum to operate on. `xchannel` specifies the x channel number. If the spectrum selected is a 2-d spectrum, `ychannel` is needed to specify the y channel. Regardless, `value` specifies the value to stuff into that spectrum channel.

Note that the *channel* coordinates are specified, not real coordinates.

**http://host:port/spectcl/channel/get?spectrum=name&xchannel=x[&ychannel=y]**

URLs of the form above, return the value of a channel of a spectrum in the `detail` attribute of the reply object. The query values `spectrum` and `xchannel` specify the spectrum name and x channel to fetch. If the spectrum is 2d, an additional `ychannel` query parameter is required to specify the y channel.

Note again, that the `xchannel` and `ychannel` coordinates are in spectrum coordinate units not real coordinates.

## 3.14. Projecting spectra (new in 5.5)

The REST interface supports spectrum projection:

**http://host:port/spectcl/project?snapshot=1|2&source=specname&newname=newspecname&direction=x|y[&contou**

URLS of this form project an existing spectrum onto one of the axes creating an new spectrum. The query parameters are:

`snapshot`

This is a boolean value. If nonzero, the projected spectrum will not increment, it's a snapshot of the projection in time. If `0`, as new data arrive, the projected spectrum will increment just like any other.

`source`

Specifies the name of the spectrum to project.

`newname`

Specifies the name of the new projected spectrum.

`direction`

Either `x` or `y` identifying the axis onto which the spectrum is projected.

`contour`

Optional. If specified, this is a contour that must have been displayable on the `source` spectrum. The projected spectrum is initially populated only with counts that are within that contour. Furthermore, if the projected spectrum is not a snapshot spectrum, it is gated on that contour (initially) so that the projection remains faithful as new data arrive.

Note that there is nothing to stop a user from applying a different gate later however, in that case, clearly the spectrum will not increment in a manner that is faithful to the original projection.

## 3.15. Spectrum Underflow and Overflow Statistics (new in 5.5)

URLs of the form:

**http://host:port/specstats[?pattern=globname-pattern]**

Return the overflow and underflow statistics for the spectra whose names match the optional `pattern` query parameter. If that pattern is not supplied it defaults to `*` returning information about all spectra.

The `detail` attribute of the returned JSON object is an array of objects that have the attributes:

`name`

Name of the spectrum being described.

`underflows`

Array of underflow counters. If a 1-d spectrum this is a single element. For 2-d spectra this is a two element array containing x and y axis undeflowas in that order.

`overflows`

Same as for `undeflows` but the array contain overflow counter(s).

## 3.16. Access to the treeevariable command (new in 5.5)

URLs of the following form list the tree variables:

**http://host:port/spectcl/treeparameter/list**

The `detail` attribute of the returned JSON object is an array of objects. Each object describes a tree variable and has the following attributes:

`name`

Name of the treevariable.

`value`

Current value of the treevariable.

`units`

Units of measure of the tree variable. Note that if this is an empty string the variable should be assumed to be unitless.

Tree variable values may be changed as well as their units:

**http://host:port/spectcl/treeparameter/set?name=varname&value=newvalue&units=newunits**

Note that the units are not optional just as they are not optional for the underlying SpecTcl **treevariable -set** command.

Treevariables have a flag that indicates if they have been modified in the life of the SpecTcl run. This flag is normally used to limit the amount of information that must be saved in files that capture the SpecTcl analysis state. This flag can be interrogated:

**http://host:port/spectcl/treevariable/check?name=varname**

The `detail` attribute of the returned JSON object is `0` if the variable has not been changed and `1` if it has.

It is possible for user interfaces to directly set the changed flag above. This is done with URLS of the following form:

**http://host:port/spectcl/treevariable/setchanged?name=varname**

Finally, there are cases where it's important to fire Tcl traces associated with tree variables. This can be done as follows:

**http://host:port/spectcl/treevariable/firetraces[?pattern=glob-pattern]**

For these URL's, *`glob-pattern`* is a GLOB pattern used to filter which traces are fired. Only variables whose names match that pattern will have their traces fired. Note that if not supplied the pattern defaults to `*` which matches all variable names.

## 3.17. Accessing the filter command (New in 5.5

Filters can be created using URLs of the form:

**http://host:port/spectcl/filter/new?name=filtername&gate=gatename&parameter=p1[&parameter=p2...]**

In this URL, the query parameters are:

`name`

The name of the filter to create.

`gate`

Specifies the name of a gate that will determine which events are output by the filter.

`parameter`

Each occurance of this query parameter specifies the name of a parameter that will be written to the filter output. Note that neither the server nor SpecTcl check for name uniqueness, it is therefore possible, although pathalogical, to output the same parameter in an event more than once.

To delete a filter use URLs of the form:

**http://host:port/spectcl/filter/delete?name=filtername**

The query parameter `name` specifies the name of the filter to delete. If the filter is enabled it is first disabled which results in a properly terinated file.

Filters can be enabled using URLs of the following form:

**http://host:port/spectcl/filter/enable?name=filtername**

Where the query parameter `name` specifies the name of the filter to enable.

Similarly:

**http://host:port/spectcl/filter/disable?name=filtername**

Disables the filter specified by the `name` query parameter.

URLs of the form

**http://host:port/spectcl/name?name=filtername&file=path**

Specify the output file for the filter. `name` specifies the filter name and `path` the filter output file path. Note that SpecTcl is writing the filter file and therefore the `path` is interpreted as a path in the context of the server not the client. Furthermore, the `path` value must specify a file that can be written by SpecTcl (not the client).

Urls of the form

**http://host:port/spectcl/list**

Returns, for the `detail` part of the JSON returned object an array of objects. Each object will describe a single filter and contains the following attributes:

`name`

Name of the filter.

`gate`

Name of the gate applied to the filter.

`file`

Name of the output file to which the filter writes.

`parameters`

An array of parameter names written to the output file for each event that passes the gate.

`enabled`

If the filter is enabled, this attribute has the value `enabled` otherwise `disabled`.

`format`

Contains the format string e.g. `xdr`.

The format of the filter output can be extended. URLs of the form:

**http://host:port/spectcl/filter/format?name=filtername&format=format**

Specify the format of a filter. `name` specifies the filter name and `format` the format type string.

## 3.18. Accessing the integrate command (new in 5.5)

There are essentially three forms of the integrate command and these are reflected in three URL/query combinations.

1. It is possible to integrate the interior of a gate on any spectrum on which it is displayable:

**http://host:port/spectcl/integrate?spectrum=spectrum-name&gate=gate-name**

In this form, the query parameter `spectrum` is the name of the spectrum on which to do the integration and `gate`'s value is the name of the gate within which to integrate.

2. It is possible to integrate a 1-d spectrum within some explicit area of interest provided in spectrum world coordinates. This is done with a URL of the form:

   **http://host:port/spectcl/integrate?spectrum=spectrum-name&low=low-limit&high=high-limit**

   As before the `spectrum` parameter is the name of the spectrum to integrate. `low` and `high` represent the low and high limits, in world coordinates, of the region of interest to integrate.

3. It is also possible to integrate a spectrum within an arbitrary polygonal region of interest. The interior of the polygon is defined using the same odd crossing algorithm used to determine the interior of SpecTcl contours; for any point, if a line drawn in any direction crosses an odd number of edges of the polygon, the point is in the gate interior.

   The format of the URL to integrate a 2-d spectrum is:

   **http://host:port/spectcl/integrate?spectrum=spectrum-name&xcoord=x1...&ycoord=y1...**

   The coordinates of the polygon are given by the `xcoord` and `ycoord` values. These are given in world coordinates and there must be at least three `xcoord`, `ycoord` pairs. The X/Y coordinates are created from ordered pairs of occurences of the `xcoord` and `ycoord` query parameter.

Regardless of how the integration is specified, the `detail` field of a successful completion is an object with the following fields:

`centroid`

: The world coordinates of the integration's centroid. For a 1-d Spectrum, this is a single value. For a 2-d spectrum, this is a 2 element array containing, in order, the x and y coordinates of the centroid.

`counts`

: Contains the number of counts inside the region of interest.

`fwhm`

The full width at half maximum of the peak (assumed to be gaussian peak shape). For a 1-d spectrum, this is just a single number. For a 2-d spectrum, this will be a 2 element array that contains the x and y extents of the FWHM in order.

## 3.19. Accessing the SpecTcl parameter command (new in 5.4)

The `/spectcl/parameter` domain of URLS actuall gives access (mostly) to the SpecTcl **treeparameter** command. The set of URLs described in this section provide access to the raw **parameter** command. Note that while this command is not much used, it is part of the SpecTcl command set and therefore a comprehensive REST interface must support it.

URL's of the form

**http://host:port/spectcl/rawparameter/new?name=pname&number=id[&resolution=bits&low=l&high=hi&units=u**

Provide support for creating a new parameter definition. A parameter definition makes a correspondence between a name and a slot in the `CEvent` pseudo array that the SpecTcl event processing pipeline fills in. Additional metadata can be provided for documentation purposes. As such there are required parameters and optional parameters and some optional parameters are coupled, in the sense that they require other optional parameters to be present as well.

The query parameters mean:

`name` (required)

Name of the parameter. This name must be unique.

`number` (required)

The parameter id. This value must also be unique. It specifies the slot in the `CEvent` object in which that parameter will be placed by the event processing pipeline.

`units` (optional)

Units of measure of the parameter (metadata). This value is not interpreted by SpecTcl but is associated with the parameter definition.

`resolution` (optional)

Provides the number of bits of resolution the parameter has. This is most useful for paramters that come from the hardware. SpecTcl does not interpret this value.

`low` (optional but...)

Assumed low limit of the parameter values. This also requires `resolution`, `high` and `units` be present. This value is not interpreted by SpecTcl

`high` (optional but...

THe assumed high limit of parameter values. This also requries `resolution`, `low` and `units` be present. This value is not interpreted by SpecTcl.

Once created, a parameter may be deleted using URLs of the following form:

**http://host:port/spectcl/rawparameter/delete?name=pname**

**http://host:port/spectcl/rawparameter/delete?id=pid**

As you can see there are two alternative forms. The first form uses the `name` to specify the name of the parameter to delete while the second form, uses the `id` to specify the parameter id (`CEvent` slot) of the parameter to delete.

Finally, the parameters and their attrubutes can be listed using one of the two URL forms below:

**http://host:port/spectcl/rawparameter/list?pattern=glob-pattern**

**http://host:port/spectcl/rawparameter/list?id=pid**

The first form lists the parameter sthat match the `glob-pattern` provided to the `pattern` query parameter while the second form provides only the parameter with the parameter id given by the `id` query parameter.

The `detail` attribute of the returned JSON contains an array of objects. Each object describes one parameter. Some of the attributes of these objects are present in all elements but the presence or absence of others depends on how the parameter was defined. The attributes are:

`name` (always present)

Name of the parameter being described.

`id` (always present)

Id (`CEvent` slot) of the parameter.

`resolution` (optional)

Only present if the resolution of the parameter was specified, this is that resolution.

`low` (optional)

Only present if the parameter low limit was specified, this is that low limit.

`high` (optional)

Ony present if the parameter high limit was specified, this is that high limit.

`units` (optional)

Only present if the parameter units of measure were specified. The units of measure string.

## 3.20. Accessing the pseudo command (new in 5.5)

The **pseudo** command in SpecTcl provides the ability to create a new parameter from existing parameters. The new parameter is computed using a script that is passed to the **pseudo** command when the parameter is generated. See the documentation of the **pseudo** command in the SpecTcl command reference.

Three URLs provide access to all aspects of the **pseudo** command:

**http://host:port/spectcl/psuedo/create?pseudo=pname&body=script&parameter=param1[&parameter=param2...]**

The new pseudo parameter is given the name in the `pseudo` query parameter. This parameter must have already been defined. `parameter` defines the parameters the pseudo depends on. The `body` query parameter defines the Tcl script that will compute the pseudo parameter. Again see the SpecTcl command reference for **pseudo** for information about the parameters that are passed to this script.

The properties of any pseudo parameters can be gotten via URLs of the form:

**http://host:port/spectcl/pseudo/list[pattern=glob-pattern]**

The return value `detail` attribute is a JSON array where each element is a JSON object that describes one pseudo parameter. Note that if the optional `pattern` query parameter is provided it is a string with glob wild card characters, and only those pseudos whose names match that pattern will be listed in th array.

Each object in the array has the following attributes:

`name`

Name of the pseudo parameter.

`parameters`

Array of strings that are the names of the parameters the pseudo depends on.

`computation`

The script that defines the computation that takes the parameters and from the creates the pseudo.

Finally pseudo parameters can be deleted using this URL

**http://host:port/spectcl/pseudo/delete?name=pseudo-name**

The `name` query parameter describes the name of the pseudo to delete. Note that the underlying parameter remains defined.

## 3.21. Access to the sread command (new in 5.5)

The **sread** command supports reading spectra from file. This is done using the following URL form

**http://host:port/spectcl/sread?filename=fname[&format=fmt&snapshot=flag&replace=flag&bind=flag]**

With the exception of `filename`, the query parameters are all optional:

`filename`

Name of the file to be read. note that the filename will be interpreted by SpecTcl not the client.

`format`

Format of the file. This defaults to `ascii` which is the best format.

`snapshot`

By default this is 1. If true, the resulting spectrum will be a snapshot spectrum. That is it will not be hooked to the increment logic of SpecTcl. If false, (0), future events can increment the resulting spectrum.

`replace`

By default this is 0. If true (1), the spectrum read in will replace any spectrum that already exists with the name of the spectrum in the file. Otherwise, if necesary, a new unique spectrum name will be assigned to the new spectrum.

`bind`

By default this is 1, which binds the spectrum into the displayer shared memory region. If 0, the spectrum will not be bound to the displayer.

## 3.22. Access the ringformat command (new in 5.5)

The ringformat command can be accessed via URLs of the form

**http://host:port/spectcl/ringformat?major=majvsn[&minor=minor]**

The `major` query parameter sets the major version, while the `minor` sets the minor version. Note that the `minor` version defaults to 0 which, since the format of ring items is not allowed to change within a major version of NSCLDAQ is normally correct.

## 3.23. Accessing the unbind command (new in 5.5)

Three URLs allow you to access the capabilities of the SpecTcl **unbind** command:

**http://host:port/spectcl/unbind/byname?name=name1[&name=name2...]**

Unbinds the spectra named by the repeated occurences of the `name` query parameter.

**http://host:port/unbind/byid?id=id1[&id=id2...]**

Unbinds the spectra whose ids are designated by repeated occurences of the `id` query parameter.

**http://host:port/unbind/all**

Unbinds all spectra from the display memory.

## 3.24. Accessing the ungate command (new in 5.5)

The SpecTcl **ungate** command removes any gate condition from a spectrum. It can be accessed via the following URL

**http://host:port/spectcl/ungate?name=name1[&name=name2...]**

Where the spectra to be ungated are specified by repeated occurences of the `name` query parameter.

## 3.25. Accessing the swrite command (new on 5.5)

The **swrite** REST interface can be used to write spectrum files on request from clients.

**http://host:port/spectcl/swrite?file=filename?spectrum=spec1[&spectrum=spec2...&format=fmtname]**

The query parameters are as follows:

`file` (mandatory)

Path to the file to write. Remember this file is written by SpecTcl and therefore the file path must not only be as SpecTcl would see it but a file writable by SpecTcl.

`spectrum` (mandatory, can be multiple)

Each occurence of this query parameter names a spectrum to be written to file. At least one spectrum must be specified.

`format` (optional)

Specifies the output fie format. Note that by default, this is `ascii`.

## 3.26. Controlling data anlaysis (new in 5.5)

Data analysis from the data source can be started or stopped using the following two urls:

**http://host:port/spectcl/analyze/start**

**http://host:port/spectcl/analyze/stop**

Note that starting analysis when analysis is running or stopping analysis when analysis is stopped will result in an error return from either of these URLs.

## 3.27. Accessing the roottree command (new in 5.5)

Note that **roottree** is only present in a SpecTcl instances that have loaded the `rootinterface` package loaded. See the SpecTcl command reference for this command

The URLs's for this part of the REST interface allow you to create Root output trees. These are objects that create CERN ROOT output files with trees that have some subsset of parameters of some subset of events.

Note that writing root trees significantly impacts the performance of SpecTcl owing to the poor performance of ROOT file output. Use with caution.

To create a new root tree you can use the following URL

**http://host:port/spectcl/roottree/create?tree=treename&parameter=pattern1[&gate=gatename&parameter=pattern**

The query parameters are as follows:

`tree`

Name of the new root tree object. This must be unique.

`parameter` (multiple)

Each of these parameters specifies a pattern that can have glob wildcard characters. The parameters that will be booked into the tree is set of parameters whose names match any of the patterns.

At least one pattern is required.

`gate` (optional)

If provided, the value of this parameter is the name of an existing gate that will determine which events are placed in the tree. Only those events that satisfy the gate will be bookeed into the tree.

If this parameter is not provided, the built-in `-TRUE-` gate is used which books all events.

To delete a tree that's already been created:

**http://host:port/spectcl/roottree/delete?tree=tree-name**

Where `tree` is the name of the tree to delete.

To learn which trees have been created and their properties:

**http://host:port/spectcl/roottree/list[pattern=name-pattern]**

Provides a return value who's `detail` lists all of the root tree objects whith names that match the optional `pattern` query parameter value. This pattern can contain glob pattern wild card characters. The `detail` attribute will be an array of objects. Each object describes one root tree and will have the following attributes

`tree`

Name of the tree to create.

`parameters`

Array of parameter name patterns that define the parameers to book into the tree.

`gate`

Name of the gate that must be satisfied to add an event to the tree. Note that if there is not one the pre-defined name `-TRUE-` will appear here.

## 3.28. Accessing the pman comman (new in 5.5)

The SpecTcl **pman** command allows users of SpecTcl to dynamically control the data analysis pipeline that turns raw data into parameters that can be histogrammed. The SpecTcl programming reference and SpecTcl programming guide describe how to set up event processors to make use of this capability.

The base command is a command ensemble with quite a few sub-commands and therefore, there are quite a few URL forms dedicated to this capability.

URLs of the following form:

**http://host:port/spectcl/pman/create?name=*`pipeline-name`***

Create a new empty analysis pipeline. The name of the new pipeline is the value of the `name` query parameter. Before being useful, the pipeline must be filled with event processors. To be used, the pipeline must also be made the current pipeline. These capabilities are exposed as well.

URLs of the following form:

**http://host:port/spectcl/pman/ls**

The `detail` attribute of the returned object is an array of strings that provide the names of all data analysis pipelines.

URLS as shown below:

**http://host:port/spectcl/pman/current**

Return as the `detail` attribute of the returned object, describes the current processor. Note that when SpecTcl starts, a pipeline named `default` is created and event processors created/registered in the traditional manner are inserted into it. The initial event processor pipeline, unless programmatically specified otherwise is the `default` pipeline.

`detail` is an object with the attributes `name` that is the name of the pipeline and `processors` which is an array of the event processors in that pipeline in the order in which they will be executed.

URLs of the following form:

**http://host:port/spectcl/pman/lsall[?pattern=*`name-pattern`*]**

Lists the pipelines and their event processors. The optional `pattern` query parameter allows you to provide a pattern that can contain glob wildcard characters. If provided, only pipeline whose name match the pattern are listed. If `pattern` is not provided it defaults to `*` which matches all pipeline names.

The `detail` attribute of the returned object is an array of objects. Each object describes a single pipeline and has the attributes:

`name`

Name of the pipeline being described.

`processors`

Array of strings of event processor names that are in the pipeline. These are indexed in the order in which they execute if the pipeline runs. See below for how to ad event processors to pipelines.

URLs of the following format:

**http://host:port/spectcl/pman/lsevp[?pattern=*`name-pattern`*]**

Allows the client to learn about the event processors that have been registered for use with the pipeline management system. Event processors can be registered in two ways:

1. Statically at run time by compiled code linked into the user's tailored SpecTcl
2. Dynamically at run time by loading a shared object.

If the `pattern` query parameter is supplied, its value is a *`name-pattern`* that restricts the set of proccessors interrogated. The *`name-pattern`* can contain Glob wild card characters. If not provided, `pattern` defaults to `*` which matches all event processor names.

The `detail` attribute of the returned object contains an array of strings. Each string is the name of an event processor that matches the `pattern` value.

URLs of the form:

**http://host:port/spectcl/pman/use?name=*`pipeline-name`***

Changes the current event processing pipeline used by SpecTcl to process events. The value of the `name` mandatory query parameter is the name of an existing event processing pipeline to make current.

URLs of the following form:

**http://host:port/spectcl/pman/add?pipeline=*`pipe-name`*&processor=*`evp-name`***

Add an existing event processor to the end of an existing processing pipeline. The mandatory `pipeline` query parameter specifies the name of the event processing pipeline. The mandatory `processor` query parameter specifies the name of the event processor to append to that pipeline.

URLs of the following form:

**http://host:port/spectcl/pman/rm?pipeline=*`pipe-name`*&processor=*`evp-name`***

Remove an event processor from a processing pipeline. The mandatory `pipeline` query parameter specifies the name of the pipeline and the mandatory `processor` query parameter specifies the name of the event processor to remove.

URLs of the following form:

**http://host:port/spectcl/pman/clear?pipeline=*`pipe-name`***

Removes all event processors from the pipeline specified by the mandatory `pipeline` query parameter. The pipeline continues to exist.

Often it's useful to create a new event processing pipeline by adding new event processors from an existing pipeline. Urls of the form:

**http://host:port/spectcl/pman/clone?source=*`old-pipename`*&new=*`new-pipename`***

Create a copy of an existing event processing pipeline. The mandatory `source` query parameter is the name of an existing pipeline to copy. The `new` mandatory query parameter is the name of the new pipline that will be created by this operation. The new pipeline will have all of the pipeline elements the original one had in the same order.

## 3.29. Accessing the evbunpack command (new in 5.5)

Inreasingly experimental setups are made up of independent detector subsystems. The NSCLDAQ event builder is then used to stitch together coincident event fragments into full experiment events.

It is convenient for the developer of each detector subsystem to indpendently build a software ecosystem to support their subsystem. It is equally convenient for the experiment integrator to be able to leverage parts of that ecosystem. Specifically, each detector subsystem may have a well developed set of event processors for SpecTcl, and it would be very good to be able to drop those into a SpecTcl built to analyze the event built data.

The **evpunpack** command leverages event processing pipline support and internal SpecTcl classes to help do just that. Specifically, you can create event processors programmatically from registered event processing pipelines that are capable of unpacking data from the event builder.

URLs of the following form,

**http://host:port/spectcl/evpunpack/create?name=evpname&frequencey=mhz&basename=treeparams;**

Create a new unpacker for event builder data. The following are all mandatory query parameters:

`name`

Name of the new event unpacker. This will be registered with the pipeline manager and, therefore, can participate in pipelines created by that command/REST interface.

`frequency`

The frequency of the event builder timestamp in MHz. This is used to provide an elapsed run time that can be used in creating diagnostic spectra from the parameters the processor automatically creates.

`basename`

The base name of the set of parameters the processor makes. For each event, the event processor fills in several diagnostic parameters in addition to any the pipelines attached to it produce:

`basename.event_no`

The event number.

`basename.run_time`

Elapsed run time based on the timestamp and frequency, in seconds.

`basename.sources`

Number of fragments in an event.

`basename.unrecognized_source`

Number of sources in the event for which no event processing pipeline was registered.

`basename.n_present`

Flag to indicate source n was present.

The intent of these parameters is to provide support for several standard event builder diagnostic spectra.

URLs of the following form:

**http://host:port/spectcl/evbunpack/add?name=evbname&source=sourceid&pipe=pipe-name**

Registers an event processing pipeline (either programmatically created or created using the pman command/REST interface) to process the event fragments from a specific event source. The data presented to the event processor is the same as if it were seeing data from a system that only took data from that subsystem.

The mandatory query parameter `name` is the name of the event processor being created. the mandatory query parameter `source` is the source id of the data that we are teaching it to process and `pipe` specifies the name of an event processing pipeline that will be invoked to process the data from fragments from the source.

URLs of the following form:

**http://host:port/spectcl/evbunpack/list[pattern=*name-pattern*]**

List the names of event processors that have been created via this command. This is returned in the `detail` attribute of the returned object as an array of strings.

## 3.30. Excuting arbitrary commands in SpecTcl (new in 5.5)

This is intended to support access to user defined SpecTcl commands. The REST interface, however arbitrary commands and scripts can be run in the interpreter using this interface:

**http://host:port/spectcl/script/command=*tcl-command***

This runs the Tcl script that is the value of the query parameter `command` at the global level of the

The result of the command is returned in the `detail` of the returned object. If the command raised an error, the `status` attribute of the returned object will be `ERROR` and the `detail` field of the returned data will have the error message raised as its value.

SpecTcl interpreter. *`tcl-command`* need not be a simple script. For example, if tcl-command's value is:

```
 {
    set result [list]
    foreach s [parameter -list] {
        lappend result [lindex $s 0]
    }
    set result
}
```

The `detail` field of a request with this as the `command` value will contain a Tcl list (not JSON array) containing the names of all parameters.

A very important point is that the server does no interpretation of the script's returned values. This is fine for Tcl clients but may be difficult to deal with properly for clients in other languages or in programs that cannot embed a Tcl interpreter to parse the results.

## 3.31. Traces (New in 5.5).

Traces are a mechanism to allow SpecTcl scripts to be informed of changes to the parameter, spectrum and gate dictionaries. As SpecTcl's analysis configuration is dynamic there are use cases, which will not be described here, for actively informing a script of changes in these objects.

Tracing is problematic for REST interfaces. Specifically, the REST models is that the client makes a request, and the server fulfils it. Fully implementation of tracing would require the server to initiate an operation in the client, rather than the other way around.

What *can* be done and what the SpecTcl REST server does do, is establish its own traces for all things that are tracable in the SpecTcl command extensions. Actual traces can be buffered for clients which then can poll for them and use the results of those polls client-side.

All of this requires that:

1. A client expresses an interest in traces and the server provides a token that associates that interest with its internal data structures (trace buffers).
2. Periodically, the client can poll the server to get the traces that fired since the last poll.s
3. The client expresses that it is no longer interested in traces causing the server to remove any trace buffers and other data associated with that client.

It is possible that a client exits without shutting down its interest in traces. Therefore, when interest is established, the client must specify a retention period for data in the trace buffer. As new traces are fired, any old data are removed from the trace buffer associated with a client. This prevents the storage required by zombie clients from growing without bounds.

URL's of the form:

**http://host:port/spectcl/trace/establish?retention=*`seconds`***

Inform the REST server that the client will be interested in trace data. The *`retention`* query parameters is the minimum number of seconds the server should retain trace data on behalf of that specific client. On success, the `detail` attribute of the response is an integer token that should be used in future calls involving the trace subsystem.

Once the application no longer requires trace information, or as it is cleaning up for exit, a request to URLs of the form:

**http://host:port/spectcl/trace/done?token=`token-value`**

Query parameter *`token`* should have, as its value, the token returned from the `spectcl/trace/establish` operation that expressed interest in traces.

To poll a trace, use URLs of the form:

**http://host:port/spectcl/trace/fetch/token=`token-value`**

The `detail` attribute of the returned JSON will will be an object. Each attribute represents a trace type and each of those attributes will, in turn have a value that is an array.

`parameter`

The value of this attribute is an array. Each element of the array describes a parameter trace. Each element is the Tcl list that contains the operation that fired the trace (`add` or `delete`). followed by the trace parameters that SpecTcl provides **parameter -trace** handlers (the nane of the parameter that was added or deleted).

`gate`

The value of this attribute is an array. Each array element describes a gate trace. The array elements are a Tcl list where the first element of the list is the gate trace operation `add`, `delete` or `change` that describes the operation that fired the trace and the remainder are the parameters that are passed to the local trace handler.

`spectrum`

The value of this attribute is an array. Each element of the array describes a spectrum trace. The elemnts are Tcl lists. The first element of that list is the operation, `add` or `delete` that fired the trace while the remainder are trace parameter(s) passed to the trace handler by SpecTcl

`binding`

The value of this attribute is an array. Each element of the array describes a change in shared memory binding status and is, itself, a three element array. The elements of the array are, in order ether the text `add` or `remove` indicating if the remaining elements describe a spectrum being added or removed from shared memory, the name of the affected spectrum and the Xamine id, or binding slot, that was allocated to the spectrum or from which the spectrum was removed.