ASCOM REST

API Reference

Abstract

This document describes the ASCOM REST API and required behavioural characteristics in detail .

Peter Simpson, Bob Denny

peter@peterandjill.co.uk, rdenny@dc3.com

Contents

1. ASCOM REST

This document is the technical reference for the ASCOM REST APIs and describes how to exploit the API. It also explains some of the fundamental behavioural principles that underly the APIs and their effective exploitation.

1.1 ASCOM REST API Documentation

The ASCOM RESTful APIs are documented using the Swagger toolset and are available through a URL on the ASCOM Standards web site. The ASCOM API is fully documented here:

`https://www.ascom-standards.org/api`

To start exploring go to the above API URL and click a grey Show/Hide link to expand one of the sets of methods and then click the blue GET or orange PUT methods for detailed information on that API call. Note that this documentation, along with the companion ASCOM Remote Server Management API is documented in the next section.

Anyone who is familiar with the ASCOM COM based APIs will immediately feel at home with the functionality available through ASCOM REST.

1.2 API Contract

This section describes how to construct REST calls to devices that expose ASCOM REST interfaces. It assumes a basic knowledge of HTTP, JSON and REST.

1.2.1 API Format

The standard API format (fixed text is black and variable elements are red) is:

```
http://Host/API/VVersionNumber/DeviceType/DeviceNumber/Command?Parameters
```

e.g.: `http://api.peakobservatory.com/API/V1/Telescope/0/AxisRates?Axis=0`
e.g.: `http://api.peakobservatory.com/API/V1/Telescope/0/CanSlew`

1.2.2 Case Tolerance

Processing of all elements of the API, except for parameters, should be done in a case insensitive manner to provide maximum flexibility.

Drivers should accept both of these examples as equivalent and process them identically:

```
http://api.peakobservatory.com/API/V1/Telescope/0/AxisRates?Axis=0
http://api.peakobservatory.com/api/v1/telescope/0/axisrates?axis=0
```

1.2.3 Client ID and Transaction Number

To support client applications using asynchronous transmission models, optional client ID and transaction numbers may be supplied with the request to identify the driver instance and specific transaction.

The driver instance number will appear in the ASCOM Remote Server logs as a debugging aid.

The transaction ID will be returned to the driver in the remote server JSON response together with any output from the driver.

1.2.4 Http Verbs

GET	Used for all information retrieval where the device state is not changed, e.g. most properties and a few functions such as Telescope. AxisRates(Axis).
PUT	is used for all other commands i.e. those which change the state of the device regardless of whether they are properties or methods e.g. Telescope.SideOfPier and Telescope.SlewToCoordinates().

1.2.5 HTTP Status Codes

The returned HTTP status code should reflect the server or driver's status as follows:

Code	Interpretation	
200	**OK**	The API request was successfully processed (even if the driver returned an exception)
400	**Bad request**	The API request could not be processed because at least one of the supplied version number, device type, device number, command or parameter values, are missing or invalid.
500	**Internal Server Error**	An error occurred in the Remote Driver Server itself, which prevented successful processing of the request.

1.2.6 Supplying Parameters

Many ASCOM methods require parameter values. All methods that use the **HTTP GET** verb should include parameters as name-value pairs in the query string.

All methods that use the **HTTP PUT** verb should include parameters using the "application/x-www-form-urlencoded" media type.

1.2.7 JSON Responses

The outcome of the command is returned as a JSON encoded class. The following information is returned for every transaction:

Item	Type	Contents
ClientTransactionID	Long	Transaction ID supplied by the client in its request
ServerTransactionID	Long	The server's transaction number. This increments by 1 on each call to the server.
Method	String	The name of the method called by the client
ErrorNumber	Int	If the driver throws an exception, its number appears here, otherwise the value 0 is returned. This will be of value to non .NET clients, in order to determine what has occurred since they have no use for a .NET exception structure.
ErrorMessage	String	If the driver throws an exception, its message appears here, otherwise an empty string is returned.
DriverException	Exception (.NET)	If the driver throws an exception, it is returned as a .NET exception structure encoded in JSON form.

In addition, the JSON response will include the output from the command (if any) in the "Value" parameter. This example is from the Telescope Simulator SupportedActions property:

```
GET /api/v1/Telescope/0/SupportedActions?Client=1&ClientTransaction=6
```

{"Value":["AssemblyVersionNumber","SlewToHA","AvailableTimeInThisPointingState","TimeUntilPointingStateCanChange"],"ClientTransactionID":6,"ServerTransactionID":6,"Method":"SupportedActions","ErrorNumber":0,"ErrorMessage":"","DriverException":null}

This example shows the response for:

```
GET /api/v1/Telescope/0/CanSlewAsync?Client=1&ClientTransaction=20
```

{"Value":true,"ClientTransactionID":20,"ServerTransactionID":168,"Method":"CanSlewAsync","ErrorNumber":0,"ErrorMessage":"","DriverException":null}

1.2.8 Driver Exception Handling

For COM based Windows drivers, any exceptions are captured by the ASOCM Remote Server and returned in JSON encoded format so that the Remote Client can recreate the original exception and throws it to the client application.

This approach is of little value to a non-Windows client, so an integer error number and error message string are also returned, which can be used by the client as needed, without having to use the complex .NET exception class structure.

The example below shows a returned exception for an attempt to set the site elevation to -301, which is an illegal value in the ASCOM specification.

```
PUT /api/v1/Telescope/0/SiteElevation
```

(parameters for the PUT verb are placed in the form body (not shown here) and do not appear after the URI as they do for the GET verb)

The response contains the error number and message:

{"ClientTransactionID":51,"ServerTransactionID":58,"Method":"SiteElevation",
"ErrorNumber":-2147220479,
"ErrorMessage":"SiteElevation set - '-301' is an invalid value. The valid range is: -300 to 10000."

As well as the exception itself:

{"ClientTransactionID":51,"ServerTransactionID":58,"Method":"SiteElevation","ErrorNumber":-2147220479,"ErrorMessage":"SiteElevation set - '-301' is an invalid value. The valid range is: -300 to 10000.","DriverException":{"ClassName":"System.Runtime.InteropServices.COMException","Message":"SiteElevation set - '-301' is an invalid value. The valid range is: -300 to 10000.","Data":null,"InnerException":null,"HelpURL":null,"StackTraceString":"   at System.Dynamic.ComRuntimeHelpers.CheckThrowException(Int32 hresult, ExcepInfo& excepInfo, UInt32 argErr, String message)\r\n   at CallSite.Target(Closure , CallSite , ComObject , Double)\r\n   at System.Dynamic.UpdateDelegates.UpdateAndExecute2[T0,T1,TRet](CallSite site, T0 arg0, T1 arg1)\r\n   at CallSite.Target(Closure , CallSite , Object , Double)\r\n   at System.Dynamic.UpdateDelegates.UpdateAndExecute2[T0,T1,TRet](CallSite site, T0 arg0, T1 arg1)\r\n   at ASCOM.Web.RemoteDeviceServer.ServerForm.WriteDouble(String method, HttpRequest request, HttpResponse response) in C:\\Users\\Peter\\Documents\\Visual Studio Projects\\ASCOM Web\\Remote Device Server\\ServerForm.cs:line 997","RemoteStackTraceString":null,"RemoteStackIndex":0,"ExceptionMethod":"8\nCheckThrowException\nSystem.Dynamic, Version=4.0.0.0, Culture=neutral, PublicKeyToken=b03f5f7f11d50a3a\nSystem.Dynamic.ComRuntimeHelpers\nVoid CheckThrowException(Int32, System.Dynamic.ExcepInfo ByRef, UInt32, System.String)","HResult":-

2147220479,"Source":"ASCOM.Simulator.Telescope","WatsonBuckets":null}} Errors and HTTP Status codes

2. ASCOM APIs - Essential Concepts

Today's world is clearly modular, cross-platform, and distributed. The core aspect of any modular system is its interfaces. If a system is built on top of poorly designed interfaces, it suffers throughout its life with limitations, instabilities, gremlins, and the like. Interface design *and negotiation* is an engineering art, the best practitioners are those that have suffered and learned.

2.1.1 Object Models - Properties and Methods

The ASCOM APIs are built on an object model which provides properties that represent some state of the device, and methods that can change the state of the device. For example, the current positional right ascension of a telescope mount is a property, and a command to slew the mount to a different position is a method. In ASCOM COM, properties are normally accessed by assignment statements in the native syntax of any of twenty languages on Windows. Methods are represented by native syntax function calls, some with parameters. There are exceptions. Some properties require parameters to signify some aspect of state, and thus may be represented by a function call which returns the property value (which need not be a scalar), for example, Telescope.AxisRates(Axis).

2.1.2 ASCOM API Characteristics

The following information applies to the existing COM-based ASCOM APIs as well as the REST-based APIs. The behaviours must be the same to provide transparent interoperation.

- **Routine Operations:** Interface design always involves some negotiations between the parties. Inevitably, a device maker may wish to have included in the interface some clever means to make their device stand out above those of his competitors. On the other hand, client programmers don't want to be writing code to manage an ever-expanding set of these clever functions. It defeats the purpose of the standardized API. The ASCOM API was therefore designed at its outset to cover *routine operations only*.

  For example, a mount really only needs "point to these coordinates" and "track the apparent motion of my object". The more accurately it does these things, the better. As a client program developer, I don't want to be concerned about PEC or encoder resolutions or servo currents.

- **Synchronous vs Asynchronous Methods:** One may think of a method call as one that returns only when the requested operation has completed, which is a synchronous call. But some types of operations can benefit by *starting* the operation and returning immediately. For example, the Rotator.Move() method may return immediately.

  If so, its return means only that the rotation was successfully *started*. Rotators are typically slow, and the system can benefit by overlapping mount and rotator movement, so both provide asynchronous calls. The status properties such as Rotator.IsMoving and Telescope.IsSlewing are used to monitor progress of asynchronous calls.

- **"Can" Properties:** Some ASCOM APIs have "can" properties, which tell the client whether or not a corresponding capability is available. For example, in the Telescope API, the CanSlewAltAz property tells the client whether this specific mount can successfully execute the SlewToAltAz() method. These "can" properties exist only for methods which can't be directly tested without changing the state of the device.

For example, a client *can* tell that the mount provides its positional azimuth by trying to read the Azimuth property; it will either get an answer or a "not implemented" error. However, a client *cannot* tell whether a mount can slew to alt/az coordinates without calling the method and possibly changing the mount's position. This is why a CanSlewAltAz property is provided for the SlewToAltAz() method.

2.1.3 Behavioural Rules

Heterogeneous distributed systems require both common standardized APIs and a set of behavioural rules that must be obeyed by all modules in the system. The *implementation* of a module is where these rules are effected, they do not appear in the abstract API definitions themselves. These behavioural rules are already implemented by ASCOM COM drivers.

ASCOM's modular rules are:

- **Do it right or report an error:** Fetching or changing a property, or calling a method, must always result in one of two outcomes: The request must complete successfully, or an error must be signalled, preferably with some (human readable) indication of why the request could not be satisfied. An example of violating this rule would be a method call to move a rotator to a given mechanical angle, but the rotator ends up at some other angle and no error is reported to the caller. Bug!
- **Retries prohibited:** No module must ever depend on another to provide timeouts or retry logic. If a device needs check-and-retry logic in its routine operation, that logic must be contained within the module itself. If there's a problem and your module's own retry logic can't resolve the issue report the error as required above.
- **Independence of operations:** To the extent possible with the device, each API operation should be independent of the others. For example, don't impose a specific call order such as needing to fetch the positional right ascension of a mount immediately before fetching the declination.
- **Timing Independence:** To the extent possible with the device, modules must not place timing constraints on properties and methods. Implement asynchronous calls wherever possible in order not to lock up clients unnecessarily.
- **Performance:** Clients should regulate their demand for driver status values in order not to jam up the driver or controller causing problems or freezes. Drivers should implement low processing cost caching, where appropriate, for status values as part of managing their overall performance and responsiveness.
- **No Status Inconsistencies:** Drivers should always report status accurately. Consider a dome shutter that a client needs to open. The client will use the ShutterStatus property after calling OpenShutter() to retrieve the shutter's status. If the driver reports ShutterOpen, even for an instant, before the shutter stats to open, the client will assume that the shutter is properly open and move on to its next task, even though the shutter is till opening.