Pixera API Documentation

Document Revision: 5

API Revision: 32

Available in Pixera beta builds after July 23rd 2019.

Contents

1. Introduction

The Pixera API provides access to Pixera functionality via a TCP/IP or UDP network port. The API is an active area of development and will continue to be extended so that it can be used to do almost anything that is possible via the main Pixera UI. The documentation of the API is also still very much in the early stages. This document aims to provide an introduction and an entry point from which to begin exploring the API in practice. It can not yet offer a complete description of all relevant functionality.

The API is not tied to a particular implementation. Within the Pixera code base, the main API code establishes a bridge between the internal Pixera objects and the outside world. Automatic code generation is then used to create interpreters for different concrete forms of access to the API. For example, both JSON-RPC and Javascript can be used to access the (logically) same functionality. As Pixera develops further, additional forms of access (e.g. scripting languages such as Python or Lua) may be added. But they will also offer the same functions, classes and methods as the current implementations do. This lends stability at a conceptual level and allows users to choose the form of access that is most convenient for them.

Despite the goal of having the different implementations available as comparably powerful options, it must be said that due to the way development has progressed JSON is currently the "first among equals" implementation. This means that there is certain functionality which is exposed via JSON that is not yet available in the other implementations. This is especially the case with regard to utility mechanisms that are just outside the scope of the API as something that reflects Pixera functionality. These exceptions will be discussed below.

The core of the API represents concepts that are part of Pixera. Beyond this core, the API mechanisms in a wider sense allow modifications of Pixera's state based on concepts defined externally. Currently, the only example of this is support for DMX512 input.

The core of the API is bidirectional in the sense that many of the available functions and methods return values. All of this exchange is expressed with the semantics of Pixera, though. If, in contrast, output is needed that is geared towards external devices, the API offers output channels that simply transport strings or binary data to whatever is attached to them.

2. Configuration in Pixera

The starting point for working with the API is the sub-tab labelled "API" in the settings. Five ports can be configured there. The first two are for access to Pixera state. There are five modes to choose from:

- JSON/TCP: process requests and send replies in the JSON-RPC format (https://www.jsonrpc.org/specification) via TCP/IP. Javascript code can be sent to Pixera as a character string that is a parameter to a special JSON message.

- JSON/TCP (dl): as above but with a delimiter-based approach to denoting packet borders instead of the standard size-based approach (see also Packaging API Invocations). The delimiter is: "0xPX".

- HTTP/TCP: Pixera can serve files commonly used on the Internet, especially: HTML, CSS, PNG. This allows Pixera to act as a simple web server. JSON-RPC messages can also be sent to the web server, allowing access to Pixera functionality from any web frontend technology. It is also possible to send JSON to Pixera without Pixera hosting the page.

- Binary/TCP: a binary representation of the API

- DMX/UDP: DMX512 packet reception over UDP (ArtNet). The data is interpreted according to the DMX patches set up in the main Pixera UI. ArtNet functionality will not be covered in this document. Rather, refer to the main Pixera documentation.

Note that multiple clients may connect to each port.

The next port is an optional heartbeat port. If a valid port number is entered then the information in the API sub tab regarding the available ports will be sent every second in the form of a JSON string (use a multicast IP to send to a multicast group). This allows clients to listen for the heartbeat and connect as soon as they receive the port and IP information.

The remaining two ports are for string or binary output. The output can currently only be triggered from cues in the timeline. To edit the output, use the appropriate field in the cue inspector. Strings can be entered directly, binary data should be entered in the form of two-character hexadecimal bytes (e.g. "FF 7B 0A" for the value 255 followed by 123 and 10).

Please note that Pixera must be restarted for changes in this sub-tab to take effect!

3. Current API Structure Documents

The core of the API consists of a set of functions, classes and methods that represent Pixera entities. There are three documents that describe these elements. They are generated automatically from the Pixera code base and reflect the structure of the current API revision. The revision number is in the name of the documents.

- pixera_api_plain_rev[num].txt: This shows the API in its most reduced form, giving only the names and data types that make up the definitions. This document provides the easiest way to get an overview of the currently available functionality.

- pixera_api_comments_rev[num].txt: Like the document above but with comments that provide additional context for individual functions and methods.

- pixera_api_examples.txt_rev[num]: Like the document above but with JSON syntax examples for every function and method invocation, including the structure of the expected reply. All the

examples are generated automatically. This means that some of the used parameter values may not make perfect sense. But the syntax is guaranteed to be correct. So the examples are an easy way to get started in that they can be copy/pasted and then customized with parameter values to quickly create valid JSON requests.

A quick glance at the "plain" description of the API should quickly show one of its most important features: it is object oriented. The basic approach is that, within a subset of the API (a namespace) there are functions that provide access to different Pixera objects (a timeline, for example). The objects have a class definition that describes what they can be asked to do. These actions can include changing their state or providing information on it. Part of an object's information can relate to child objects that it contains. These objects are accessed by asking the parent for a handle to one of them. When the handle is available then those methods defined in the child's class definition can be used with it.

A result of this approach is that all class methods defined in the API expect a handle value to be set as the first parameter. In the interests of concise description, the handle parameters are omitted in the reduced definitions in pixera_api_plain_rev[num].txt. But examples of specifying a handle can be found in all the JSON invocations that relate to a class instance in pixera_api_examples_rev[num].txt.

In the JSON implementation, a special flag can be set which places the method and handle used in an invocation into the response. This provides an alternative besides the message id for matching invocation and response. To set this flag, send the following JSON request to Pixera:

{"jsonrpc":"2.0", "id":49, "method":"Pixera.Utility.setShowContextInReplies",
"params":{"doShow":true}}

When this setting is active, JSON responses will contain a "context" sub-object with "method" and "handle" keys that have the same values as those in the last request sent to the connection.

4. Packaging API Invocations

The main underlying protocol for accessing Pixera is TCP/IP. Since this protocol is stream-oriented, it is not possible to send Pixera, for example, a JSON string without any further information. Rather, packets must be built so that Pixera knows where one packet ends and the next one begins.

This is done with a very simple header. The header starts with a tag consisting of the four characters "pxr1". It is then followed by four bytes containing an unsigned integer (least significant byte first) the value of which corresponds to the size of the packet payload (i.e. the size of the entire packet excluding the tag and the size integer itself). This is followed by the payload data. In the case of JSON, that would be the JSON message in UTF-8 format..

Consider the function Pixera.Timelines.getTimelineAtIndex. The example given in pixera_api_examples.txt is:

{"jsonrpc":"2.0", "id":20, "method":"Pixera.Timelines.getTimelineAtIndex", "params":{"index":1}}

The image below shows how a valid packet for that invocation looks in RAM. The first four bytes are "pxr1". This is followed by 5d, i.e. 93. That is the length of the JSON string (without spaces). The other three size bytes are null since the size of the string fits in the first (the least significant) byte.

```
70 78 72 31 5d 00 00 00 7b 22 69 64 22 3a 31 32  pxr1]...{"id":12
2c 22 6a 73 6f 6e 72 70 63 22 3a 22 32 2e 30 22  ,"jsonrpc":"2.0"
2c 22 6d 65 74 68 6f 64 22 3a 22 50 69 78 65 72  ,"method":"Pixer
61 2e 54 69 6d 65 6c 69 6e 65 73 2e 67 65 74 54  a.Timelines.getT
69 6d 65 6c 69 6e 65 41 74 49 6e 64 65 78 22 2c  imelineAtIndex",
22 70 61 72 61 6d 73 22 3a 7b 22 69 6e 64 65 78  "params":{"index
22 3a 30 7d 7d fd fd fd fd 00 00 00 00 00 00 00  ":0}}ýýýý.......
```

An alternative packet-border approach based on a delimiter is also available. This requires the mode of the API access port to be "JSON/TCP (dl)". The connection then expects each individual JSON message to have the four characters "0xPX" appended to it. A leading tag/size pair is not used in this mode. Returned results will also start directly with the JSON string but end with the delimter.

In the binary implementation of the API, the main part of the payload is described by pairs of tokens and values. In requests, the token/value pairs are used to describe the parameters of the function. In replies, they describe the return value or values.

The token is an unsigned byte that describes the type of the value that follows in the stream. The currently used tokens are:

TYPE_TOK_HANDLE = 1
TYPE_TOK_STRING = 2
TYPE_TOK_BOOL = 3
TYPE_TOK_INT = 4
TYPE_TOK_DOUBLE = 5
TYPE_TOK_FLOAT = 6
TYPE_TOK_UCHAR = 7
TYPE_TOK_UINT = 8
TYPE_TOK_HANDLE_ARRAY = 64
TYPE_TOK_STRING_ARRAY = 65
TYPE_TOK_BOOL_ARRAY = 66
TYPE_TOK_INT_ARRAY = 67
TYPE_TOK_DOUBLE_ARRAY = 68
TYPE_TOK_FLOAT_ARRAY = 69
TYPE_TOK_UCHAR_ARRAY = 70
TYPE_TOK_UINT_ARRAY = 71

The following conventions are also used:

- The payload begins with a sequence number. It is a 32 bit unsigned integer (least significant byte first). Pixera will place the same value in a reply to the message so that the client can relate requests and replies to one another. The sequence number is not preceded by a type token!

- The sequence number is followed by four bytes that can be used to further specify the context of the request. They are also placed into the reply. These bytes are not preceded by a type token!

- The order of parameter values in a packet is the same order that they are listed in the documentation. If the function returns a struct, the order of the return values is the same as the order in which the struct members are listed in the documentation.

- Handles are 64 bit unsigned integers. However, only the first 53 bits are used so that conversion to and from double is lossless.

- INT and UINT are assumed to be four bytes in size.

- All integers are transferred least significant byte first.

- A float is assumed to be four bytes in size, a double eight bytes.

- All floating point values are represented in conformance with IEEE 754.

- String data is preceded by a signed 32 bit integer giving the size of the string.

- Array data is preceded by a signed 32 bit integer giving the number of array elements.

- Each array element is a standard token/data pair.

When accessing the Pixera API via HTTP, the JSON messages must be sent to the server as part of a POST request. This is easiest if Pixera is hosting the page because in that case the server URL is implied. To use this mechanism, place a folder called "html_root" in the data subdirectory of a Pixera installation and place a HTML page there. Make sure the Pixera access port is set to "HTTP/TCP". The page can be accessed by entering the Pixera system's IP, the port and the path to the page (starting from html_root) into a browser. The URL should look something like this:

http://192.168.178.52:1400/api_test.html

Within the page, use a XMLHttpRequest to send JSON messages to Pixera. The following is an example Javascript function that sends the passed-in command (a JSON string) and then passes the result to the passed-in resultFunc.

```
function pixeraApi(cmd, resultFunc)
{
    var request = new XMLHttpRequest();
    request.open('POST', '', true);
    request.setRequestHeader("Content-type", "application/json");
    request.send(cmd);

    request.onload = function ()
```

```
    {
        resultFunc(request.status, request.responseText);
    };
}
```

Pixera need not host the HTML pages that access the API, though, since cross-domain access is supported. If Pixera is not the source of the pages then the Javascript for posting the requests needs to be amended to take the Pixera system's IP into account. For example:

```
function pixeraApi(ip, port, cmd, resultFunc)
{
    var request = new XMLHttpRequest();
    request.open('POST', 'http://' + ip + ':' + port, true);
    request.setRequestHeader("Content-type", "application/json");
    request.send(cmd);

    request.onload = function ()
    {
        resultFunc(request.status, request.responseText);
    };
}
```

The code above allows Pixera API access from any web page.

Another form of invoking the API is to use Javascript code not to transfer JSON strings but, rather, to manipulate Pixera objects directly. In the near future, Pixera will provide opportunities for users to enter Javascript that is programmed against the API and have it execute immediately. Currently, Javascript must be sent to Pixera as the parameter of a special JSON method. This would, for example, be a way for client programs to provide their users with the ability to write and execute macros that manipulate Pixera.

The special JSON-RPC function needed for this is called "Pixera.Utility.runJsScript". It has two parameters. The first is "jsCode". Its value is the Javascript code to be loaded by Pixera. The second parameter is "jsFunction". It names a function defined in the Javascript code that Pixera should execute.

The following script sets some position, rotation and scale values in the first layer of the first timeline. It then assigns the first resource in the folder "Media" to the layer. The line "tl.store()" saves all the changes to the timeline. When asking Pixera to execute this, the code would be the value of the "jsCode" parameter and "onTest" would be the value of the "jsFunction" parameter.

```
function onTest()
{
    var odb = Pixera.Utility.outputDebug;

    var tns = Pixera.Timelines;
    var tl = tns.getTimelineAtIndex(0);
    var layer = tl.getLayerAtIndex(0);
    var param = layer.getNodeWithName("Position").getParamWithName("x");
    param.setValue(0);
```

```
    layer.getNodeWithName("Position").setValues([6, 1, 3]);
    layer.getNodeWithName("Position").setValues([2, 4, 9.5]);
    layer.getNodeWithName("Scale").setValues([2.0, 1.0, 0.5]);

    setVals(tl, 0, "Rotation", [10, 30, 40]);


    var resFolder = Pixera.Resources
            .getResourceFolderWithNamePath("Media");

    var resources = resFolder.getResources();
    var ctRes = resources.length;
    if(ctRes > 0)
    {
        odb("assigning: " + resources[0].getName());
        layer.assignResource(resources[0].getId());
    }

    tl.store();
}
```

Another aspect of this example script is worth noting. Due to a limitation in the current implementation of the Javascript interpreter in Pixera, object instances can not be passed as parameters to Javascript functions. This is why getId() is used to assign the resource to the layer in the next to last statement in the script.

5. Handling Changes to the API

In general, AV Stumpfl will try to minimize changes to the API and will try to maintain compatibilty with older uses of the API if doing so does not place undue burdens on users of the current version. But the API is under active development and will evolve. And even in those case where old functionality does not change the question arises of how to deal with newly introduced functionality when clients can not be updated in parallel. Or, vice versa, how a newer client can deal with an older version of Pixera.

The API currently provides two approaches for handling this. The first is to use the function Pixera.Utility.getApiRevision(). This function will return a new value on every update of the API and will thus allow exact matching of clients to the API supported by a particular Pixera installation.

The downside of this approach is that *any* changes to the API, even those in areas irrelevant to a particular client application, may then be seen as constituting an incompatiblity. A more fine-grained approach is available in the form of the function Pixera.Utility.getHasFunction. It returns true if the passed-in function name is known to the active Pixera instance.

Using this approach a client application could first check if all the functions it needs are available and proceed if and only if that is the case. This could allow the application to function as expected in all situations where the functionality is in fact available. At the same time, users of the application would not have to worry about API changes that do not concern the features they are interested in.

6. Accessing State Updates (Monitoring)

The API provides a means for clients to keep up to date on state changes in Pixera. The current approach is based on a polling model so it harmonizes well with the request/reply structure of communication via JSON-RPC or with a web server.

The monitoring functionality is currently only available via the JSON implementation!

To request the current state, send the method invocation (with no parameters) "Pixera.Utility.pollMonitoring".

The data is maintained separately for each connection to Pixera and is stored until the client requests it. It is worth noting that this constitutes an important difference to a "pure" polling model. Using only polling in the narrow sense state changes that are not in effect at the time of the polling could be missed by the client. In the monitoring approach taken here, though, a discrete change like adding a cue is stored as an entry for the connection and this information remains available until whichever time the client decides to request it. This allows the client to optimize the polling interval for the frequency at which it wants to process the data without having to worry that it will miss any data.

This approach also allows the data to be stored and transferred more efficiently. Assume, for example, that two cues were changed since the monitoring state was last requested. This can then be represented with a string denoting the action ("cueChanged") along with a list of cue handles that were affected. It is not necessary to transfer the "cueChanged" string for every cue.

Some of the information available via monitoring represents values that change rapidly over time, e.g. the current position of a timeline. For these changes to conceptually continuous variables, only the most recent value is retained and that value is guaranteed to be present in the next monitoring data set that is transferred to the client. A s a result, a client monitoring all timeline positions at a rate lower than any of the timelines' FPS would only have precisely the data transferred that is relevant for the update frequency it is using.

The data is flushed from memory after it was returned to the caller on a connection. This means that only information that was generated since the last polling invocation is returned and the caller does not have to be able to identify and ignore stale data.

The returned data is organized by subjects. The range of subjects will be expanded as Pixera development continues. If a client is not interested in a subject, it can send the following request (e.g. for the subject "timelinePositions"):

{"jsonrpc":"2.0", "id":31, "method":"Pixera.Utility.unsubscribeMonitoringSubject", "params":{"subject":"timelinePositions"}}

This will remove the subject from the updates for that particular connection.

The function "Pixera.Utility.subscribeMonitoringSubject" can be used to (re-)subscribe to a subject.

Each monitoring response has an array named "result" with one entry for each subject. Each subject has a name and an array called "entries" with information on individual occurrences.

The currently available subjects are:

- "clipAdded": The entries array contains a single JSON array called "handles". It contains the handles of all clips added since the last time the monitoring results were requested for the connection.

- "clipChanged": As above but for all clips that were changed.

- "clipRemoved": As above but for all clips that were removed.

- "cueAdded": The entries array contains a single JSON array called "handles". It contains the handles of all cues added since the last time the monitoring results were requested for the connection.

- "cueChanged": As above but for all cues that were changed.

- "cueRemoved": As above but for all cues that were removed.

- "cueApplied": As above but for all cues that were applied, either by way of a running timeline reaching them or by other means (e.g. an API invocation).

- "timelineAdded": The entries array contains a single JSON array called "handles". It contains the handles of all timelines added since the last time the monitoring results were requested for the connection.

- "timelineRemoved": As above but for all timelines that were removed.

- "timelineTransport": The entries array contains "handle"/"value" pairs. The "value" holds the transport mode (as in the parameter to Pixera.Timelines.Timeline.setTransportMode) of the timeline with the "handle".

- "timelinePositions": The entries array contains "handle"/"value" pairs. The "value" gives the time (in frames) of the timeline with the "handle".

- "timelineCountdowns": The entries array contains "handle"/"value"/"flag" tuples. The "value" gives the time (in frames) of the current countdown of the timeline with the "handle". If the countdown is a result of the timeline moving towards the next cue the "flag" value will be 1. If the countdown is due to the timeline pausing while the wait duration of a cue is counted down then the "flag" value will be 2.

7. Monitoring Event Orientation

The state access mechanism discussed above gathers together all state changes for each connection and then transfers them all at once when the client requests it (via "Pixera.Utility.pollMonitoring").

Alternatively, monitoring can be used in an "event-oriented" mode in which monitoring entries are sent to the client connection as soon as they occur. To change the mode, send a message of the following form:

{"jsonrpc":"2.0", "id":32, "method":"Pixera.Utility.setMonitoringEventMode",
"params":{"mode":"all"}}

The "mode" parameter can take three values:

- "none" (default): Entries are never sent directly. Rather, all data is transferred as a reply to the pollMonitoring request.

- "all": All entries are immediately sent. With this setting, pollMonitoring need never be called. (If it is called, the result will never contain any entries.)

- "onlyDiscrete": In this mode, those state changes which do not involve continuously changing values are sent immediately while state changes that consist of rapidly updating a value are only delivered as a response to pollMonitoring.

Clients can recognize incoming monitoring events by the fact that they have a "type" attribute with the value "monEvent". For example, an event communicating that a cue has been added could look like this:

{"jsonrpc":"2.0", "id":-1, "type":"monEvent", "name":"cueAdded",
"entries":[{"handles":[6511533668781846]}]}

8. Experimental Functionality: Custom UIs in Pixera

An area of ongoing development concerns using the API mechanisms to build and control UIs within Pixera. This functionality can currently only be tested in special versions of Pixera which will be made available as individual collaborative efforts require. They are described here briefly anyway because the goal is to make them broadly available in the medium term.

The UIs are defined using an XML document. The tags in the document list the controls that are to be presented to the user. In addition, the document can contain Javascript that is executed depending on how the controls are interacted with. There is an additional area for scripting functionality shared by all controls.

The XML file can be placed in an encrypted bundle so that the creator of the file can grant access only to certain users (identified via their dongles).

The example below shows what could be an inspector for an effect. It asks Pixera to load a combo box and a button. When the combo box is first loaded it is filled with preset values. When the user selects an entry the value of the effect's parameter is set to a value determined by the preset. When the button is pressed that value is stored to the timeline as a key.

```
<?xml version="1.0" encoding="utf-8"?>
<controls label="Zephyr">
<helperScript>
  var odb = Pixera.Utility.outputDebug;
</helperScript>
<control type="comboBox" label="Presets">
  <script handler="onUpdate">
    var presetComboBox = Pixera.Ui.getComboBoxWithId(controlId);
    presetComboBox.clear();
    presetComboBox.addItem("Better than you Blue", 1);
    presetComboBox.addItem("St. Giles Blue", 2);
    presetComboBox.addItem("Pitch Blue", 3);
    presetComboBox.setSelectedId(1);
  </script>
  <script handler="onEvent">
    var presetComboBox = Pixera.Ui.getComboBoxWithId(controlId);
    var selId = presetComboBox.getSelectedId();
    var fx = Pixera.Timelines.getNodeFromId(contentId);
    if(fx === undefined)
    {
      return;
    }
    var param = fx.getParamWithName("Qolor");
    if(param === undefined)
    {
      return;
    }
    param.setValue(200 * selId);
  </script>
</control>
 <control type="textButton" label="" text="Store">
    <script handler="onEvent">
      var fx = Pixera.Timelines.getNodeFromId(contentId);
      if(fx === undefined)
      {
        return;
      }
      fx.getTimeline().store();
    </script>
  </control>
</controls>
```