

ConsoleServer

A simple web server oriented towards OSC capable digital consoles

Applies to WING, X32, M32, XAir series

©2024 – Patrick-Gilles Maillot

Ver. 1.3

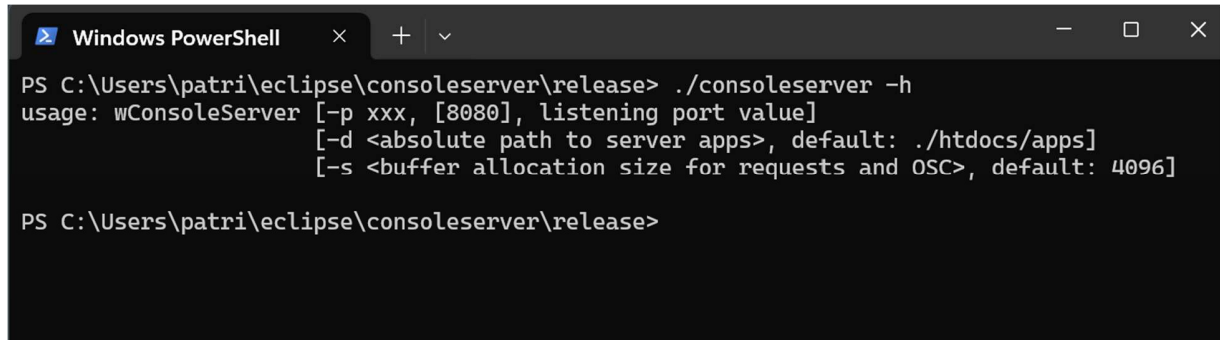
Table of Contents

Introduction	3
Structure of ConsoleServer directories.....	4
Launching a web browser application	5
Appendix: index.html (an example)	6
Appendix: Default Application list.....	8
X32 Actions	8
wSetX32IpChNumChName	8
wSetX32IpBusNumBusName	8
wGetX32IpChNameFromNum	8
wGetX32IpBusNameFromNum	8
WING actions	8
wServerSetMain1.....	8
wServerSendNames	8
wSetWingIpChNumChName	9
wSetWingIpBusNumBusName.....	9
wGetWingIpChNameFromNum	9
wGetWingIpBusNameFromNum.....	9
Generic OSC actions	9
ServerPutOSC	10
ServerPostOSC.....	10

Introduction

ConsoleServer is a simple http server that serves HTTP protocol compliant web requests such as **GET**, **PUT** and **POST** methods issued from web browsers.

The server can be launched from a terminal window and provides a few options as shown below



```
Windows PowerShell
PS C:\Users\patri\eclipse\consoleserver\release> ./consoleserver -h
usage: wConsoleServer [-p xxx, [8080], listening port value]
                    [-d <absolute path to server apps>, default: ./htdocs/apps]
                    [-s <buffer allocation size for requests and OSC>, default: 4096]

PS C:\Users\patri\eclipse\consoleserver\release>
```

The default communication port is **8080** and can be changed at launch time, as the directory where the server will find applications to launch following a **POST** or **PUT** methods. **PUT** is used for providing data to be sent to the console without expecting a reply from the console. **POST** is used in similar way, but a reply is expected from the console.

User Interface

The user interface is typically a web page running in a web browser. The user will connect to the server using its `IP:port` pair and this will trigger a **GET** request to provide the web page contents (by default `index.html`). This page can display information (as would do any web page) based on the service(s) the console owner wants to provide through using the web server.

Such services may prompt the user to provide data for the console, typically using **PUT** or **POST** request methods providing data the server should send to the console. In some cases, the console will reply with data that will be returned to the user. The elements involved in the case of **ConsoleServer** are:

1. **URL/Endpoint:** The URL (Uniform Resource Locator) or endpoint specifies the server location where the POST request is sent. It identifies the specific resource or action that the server should handle. It acts as a unique address for accessing an API endpoint.
2. **Request Method:** The request method indicates the type of action being performed on the server. In the case of a **PUT** request, it signifies that data is being submitted to the server to create or update a resource. **POST** works in a similar way but a reply is expected from the console and the server will return that reply in text form to the requesting web browser.
3. **Request Body:** The request body carries the actual data being sent in the **PUT** or **POST** request. It contains the payload or content that needs to be processed by the server. **ConsoleServer** uses text/plain type contents to provide the information to the console.
4. **Response:** The response is the server's reply to the client's request. It contains the result of the requested action, including status information, data from the server (if applicable), and additional metadata. The response allows the client to understand the outcome of the request and handle it accordingly.

One particularity of **ConsoleServer** is the API endpoints are directly named after the URL/Endpoints and can be programmed and used independently from the server code itself. **ConsoleServer** also provides built-in generic OSC PUT or POST which can be used by advanced users and provide a faster response time, to the expense of having to code/decode OSC within the user web browsing space.




ConsoleServer users can add as many services/endpoints/API entries as they want, without having to change any of the actual server code. The server will use the URL data to create a file path to the requested service which will in fact launch an external program found at the file path built by the server, and taking the body of the **PUT** or **POST** request method as arguments. Because applications used in **POST** methods are independent from the server, a file is used to return data from the console to the server. This file is a text file named `wConsoleServerDataFile.txt`, located in the directory where applications run (by default `htdocs/apps`).

Structure of ConsoleServer directories

ConsoleServer comes with the following directory tree:

<input type="checkbox"/> Name	Type
 htdocs	File folder
 ConsoleServer.exe	Application

htdocs directory:

<input type="checkbox"/> Name	Type
 apps	File folder
 static	File folder
 index.html	Microsoft Edge HTML Document

static directory:

<input type="checkbox"/> Name	Type
 images	File folder
 script.js	JavaScript File
 styles.css	Cascading Style Sheet Document

apps directory:

<input type="checkbox"/> Name	Type
 wServerSendNames.exe	Application

Launching a web browser application

By default running a web browser on the IP:port data corresponding to **ConsoleServer** will parse the index.html file. This file can be modified by the **ConsoleServer** owner at wish. The current contents given in appendix basically displays a banner and asks the web browser user (client) to provide names that will be sent to the API listed as URL in the index.html lines.

As the user enters data and hits the <enter> key, data is placed in a JSON structure and sent to the server along with URL name wServerSendNames. This specific API entry is also the name of a program found in the server apps directory.

When the server launches the wServerSendNames application, it passes the body of the POST request method to the application. wServerSendNames has then all the necessary information to complete its specific task, which is to decode the JSON payload in the body of the request and send the names it finds respective to their channel strips numbers. In essence, the wServerSendNames service enables a remote filling of **WING'** scribble names, as provided by session musicians, without them having physical access to, or any specific knowledge of the console.

Although the default functionality of **ConsoleServer** is indeed limited, one understands quickly the extension possibilities that are offered by such service for studios or console owners who need to restrict access to specific areas of their console or on the contrary, offer a wider freedom for session user to customize their experience. This can be achieved independently from the **ConsoleServer** code itself, by adding (or using the default) applications to the server application directory.

Appendix: index.html (an example)

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>Populating WING scribbles with names (c)2024 P.G. Maillot</title>
  <style>
    ul {
      border: 3px solid #ccc;
      width: 600px;
      list-style: none;
      margin-left: 10px;
      padding: 10px;
    }
    li {
      margin-left: 50px;
      margin-bottom: 8px;
    }
    li input {
      padding: 8px;
      border: 1px solid #ccc;
    }
  </style>
</head>

<body>
<center>
<h1><span style="font-size:64px">
<strong><div style="font-family:Arial,Helvetica,sans-serif"><div style="background-
color:#f1c40f">
Studio PGM
</div></div></strong>
</span></h1>
<br>

</center>
<br><hr/><br>
<div style="font-family:Arial,Helvetica,sans-serif">
<h2>Enter ch. strip Names</h2></div>
  <ul>
    <li>Ch 01: &nbsp;<input type="text" maxlength="16" size="30" id="name1" onkeydown="if
(event.key === 'Enter') sData()"></li>
    <li>Ch 02: &nbsp;<input type="text" maxlength="16" size="30" id="name2" onkeydown="if
(event.key === 'Enter') sData()"></li>
    <li>Ch 03: &nbsp;<input type="text" maxlength="16" size="30" id="name3" onkeydown="if
(event.key === 'Enter') sData()"></li>
    <li>Ch 04: &nbsp;<input type="text" maxlength="16" size="30" id="name4" onkeydown="if
(event.key === 'Enter') sData()"></li>
  </ul>

  <script>
    function sData() {
      const names = {
        "01": document.getElementById("name1").value,
        "02": document.getElementById("name2").value,
        "03": document.getElementById("name3").value,
        "04": document.getElementById("name4").value,
      };
      // Convert object to JSON string
      const jsonData = JSON.stringify(names);
      // Replace this with your actual submission logic, e.g., using fetch
      fetch("wServerSendNames", {method: 'PUT', body: jsonData})
        .then(response => console.log("Response:", response))
        .catch(error => console.error("Error:", error));
    }
  </script>
</body>
</html>
```

Will produce the following where names have been entered:

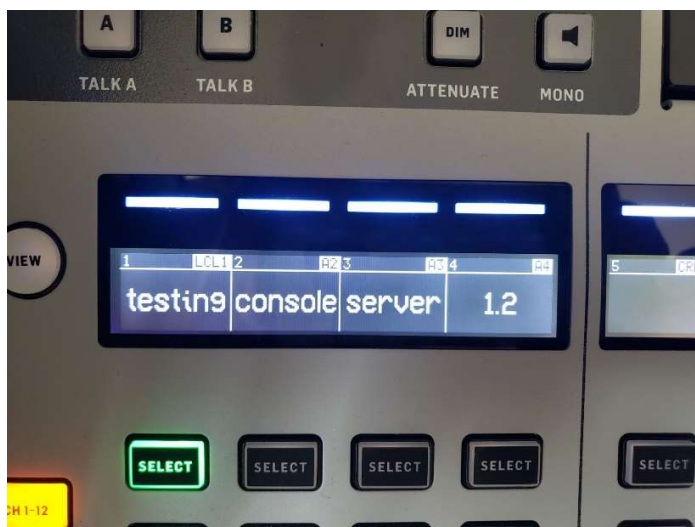
Studio PGM



Enter ch. strip Names

Ch 01:	<input type="text" value="testing"/>
Ch 02:	<input type="text" value="console"/>
Ch 03:	<input type="text" value="server"/>
Ch 04:	<input type="text" value="1.2"/>

And result on a WING console in the following:



Appendix: Default Application list

The paragraphs below list applications that are proposed by default with the **ConsoleServer** .zip package. For each application, a quick description of the app is proposed along with its interfacing with the server, and if applicable the returned data format.

X32 Actions

wSetX32IpChNumChName

Set X32 Channel strip name, providing X32 IP, channel number and channel name in a commas-separated string within a **PUT** method. Replies from the server include OK, Bad_Request, Not_Implemented, Internal_Server_Error.

wSetX32IpBusNumBusName

Set X32 Bus strip name, providing X32 IP, Bus number and Bus name in a commas-separated string within a **PUT** method. Replies from the server include OK, Bad_Request, Not_Implemented, Internal_Server_Error.

wGetX32IpChNameFromNum

Get X32 Channel strip name within a timeout of 500ms, providing X32 IP, and Channel number in a commas-separated string within a **POST** method. Replies from the server include OK, Bad_Request, Not_Implemented, Internal_Server_Error, Request_Timeout. On “OK”, the server response will contain the requested Channel name in full text form.

wGetX32IpBusNameFromNum

Get X32 Bus strip name within a timeout of 500ms, providing X32 IP, and Bus number in a commas-separated string within a **POST** method. Replies from the server include OK, Bad_Request, Not_Implemented, Internal_Server_Error, Request_Timeout. On “OK”, the server response will contain the requested Bus name in full text form.

WING actions

wServerSetMain1

A quick test for setting WING’s Main1 fader, typically used within a **PUT** method. The data send to the server is a string containing a representation of a floating point value [-144.0,...,10.0]. Replies from the server include OK, Bad_Request, Not_Implemented, Internal_Server_Error.

wServerSendNames

Send a list of channel strip names for WING’ scribbles, typically used within a **PUT** method. The data send to the server is a list of commas-separated strings containing the names for each WING channels. Replies from the server include OK, Bad_Request, Not_Implemented, Internal_Server_Error.

wSetWingIpChNumChName

Set WING Channel strip name, providing WING IP, channel number and channel name in a commas-separated string within a **PUT** method. Replies from the server include OK, Bad_Request, Not_Implemented, Internal_Server_Error.

wSetWingIpBusNumBusName

Set WING Channel strip name, providing WING IP, channel number and channel name in a commas-separated string within a **PUT** method. Replies from the server include OK, Bad_Request, Not_Implemented, Internal_Server_Error.

wGetWingIpChNameFromNum

Get WING Channel strip name within a timeout of 500ms, providing WING IP, and Channel number in a commas-separated string within a **POST** method. Replies from the server include OK, Bad_Request, Not_Implemented, Internal_Server_Error, Request_Timeout. On "OK", the server response will contain the requested Channel name in full text form.

wGetWingIpBusNameFromNum

Get WING Bus strip name within a timeout of 500ms, providing WING IP, and Bus number in a commas-separated string within a **POST** method. Replies from the server include OK, Bad_Request, Not_Implemented, Internal_Server_Error, Request_Timeout. On "OK", the server response will contain the requested Bus name in full text form.

Generic OSC actions

ConsoleServer supports OSC requests to be sent with no specific knowledge of the console at IP and Port values provided by the client. This offers a fully generic OSC support for any type of console or other device that can be addressed within the server. These requests are not calling external applications and can therefore provide minimal latency in the communication with the device. The OSC notation used follows the principles used in other OSC command line interface type tools such as X32_Command¹, or wosc². Buffers for generic OSC actions are limited to 4kBytes.

```
<command> [<format> [<data> [<data> [...]]]], where for example:  
command: /?, /, /ch/1/fdr, ...  
format: ', ' ',i' ',f' ',s' or a combination: ',siss' ',ffiss' ...  
data: a list of int, float or string types separated by a space char...
```

```
format can also be: ',b' used on node commands, see below:  
the ',b' format can used to request a WING node definition, or to send Wing  
node data such as /ch/1 ,b <data in hex>;  
for example, setting color of channel 1 to index 5 (refer to native commands):  
/ch/1 ,b d75a46d1bcd400000005dd
```

¹ See X32_Command at https://sites.google.com/site/patrickmaillot/x32#h.p_xEXVAqEZmgHK

² See wosc at <https://sites.google.com/site/patrickmaillot/wing#h.7ql1rnoc3z60>

Please refer to the abovementioned applications and the consoles respective OSC or native commands documentation³ for a list of OSC possibilities. **ConsoleServer** being an http-protocol connection application, it will not provide console data that have not first being requested by the server. For example, meters data or reading interactive user changes on faders or buttons are not part of the scope of **ConsoleServer**.

ServerPutOSC

Send an OSC command to connected console at given IP and Port within a **PUT** method. No reply is expected from the console. Data sent to the console include IP, Port, and OSC command in text format, in a commas-separated string. Replies from the server include `Not_Implemented`, `Internal OK`, `Bad_Request`, `_Server_Error`.

For example, setting X32 at IP 192.168.1.129 bus 1 name to value `newName`:

```
192.168.1.129,10023,/bus/01/config/name ,s newName
```

For example, setting WING at IP 192.168.1.155 channels 1 and 4 names to values `New1` and `New2` respectively:

```
192.168.1.155,2223,/ ,s ch.1.name=New1,.2.name=New2
```

ServerPostOSC

Send an OSC command to connected console at given IP and Port within a **POST** method, and return reply from console. Data sent to the console include IP, Port, and OSC command in text format, in a commas-separated string. Replies from the server include `OK`, `Bad_Request`, `Not_Implemented`, `Internal_Server_Error`, `Request_Timeout`. On “OK”, the server response will contain the reply to the OSC request in full text form.

For example, inquiring X32 at IP 192.168.1.129 for bus 1 name:

```
192.168.1.129,10023,/bus/01/config/name
```

The console reply is returned by the server to the calling browser as a single string of characters. As OSC is a binary protocol that contains many `\0` chars, these characters are replaced with the `~` character. A typical reply will look like:

```
/bus/01/config/name~,s~~bus001~~
```

For example, requesting the node data for ch1 EQ on Wing at IP 192.168.1.155:

```
192.168.1.155,2223,/ch/1/eq ,s *
```

³ See X32 Unofficial Remote Protocol at https://drive.google.com/file/d/1Yt_S1mpPt3CAzeg3Dnpe_lqctQ-1GLtZ/view?usp=drive_link, or Wing Remote Protocols at <https://drive.google.com/file/d/1-iptgd2Uxw4qPEbmegG2Sqccf8AbRRfk/view>

The console reply can [depending on actual EQ settings] be:

```
/ch/1/eq~~~,s~~on=1,mdl=STD,mix=100,lg=-9.9,lf=22.8,lq=1.00,leq=SHV,lg=-  
7.4,lf=119.1,lq=1.00,2g=+1.2,2f=305.8,2q=1.00,3g=+4.9,3f=1k38,3q=1.00,4g=0.0,4f  
=3k99,4q=1.00,hg=0.0,hf=12k00,hq=1.00,heq=SHV,~~
```