Contents

HTTP Server

*** Sergey Linev GSI, Darmstadt ***

Chapter 1

HTTP server in ROOT

The idea of THttpServer is to provide remote http access to running ROOT application and enable HTML/JavaScript user interface. Any registered object can be requested and displayed in the web browser. There are many benefits of such approach:

- standard http interface to ROOT application
- no any temporary ROOT files to access data
- user interface running in all browsers

1.1 Starting the HTTP server

To start the http server, at any time, create an instance of the THttpServer class like:

```
auto serv = new THttpServer("http:8080");
```

This will start a civetweb-based http server on the port 8080. Then one should be able to open the address `http://localhost:8080` in any modern browser (Firefox, Chrome, Opera, Safari, IE11) and browse objects created in application. By default, the server can access files, canvases, and histograms via the gROOT pointer. All those objects can be displayed with JSROOT graphics.

There is a server snapshot of running macro tutorials/http/httpserver.C from ROOT tutorials.

One could specify several options when creating http server. They could be add as additional URL parameters to the constructor arguments like:

```
auto serv = new THttpServer("http:8080?loopback&thrds=2");
```

Following URL parameters are supported:

Name	Description
thrds=N	number of threads used by the civetweb (default is 10)
top=name	configure top name, visible in the web browser
auth_file=filename	authentication file name, created with htdigets utility
auth_domain=domain	authentication domain
loopback	bind specified port to loopback 127.0.0.1 address
debug	enable debug mode, server returns html page with request info
websocket_timeout=tm	set web sockets timeout in seconds (default 300)
websocket_disable	disable web sockets handling (default enabled)
cors=domain	define value for CORS header “Access-Control-Allow-Origin” in server response
log=filename	configure civetweb log file
max_age=value	configures “Cache-Control: max_age=value” http header for all file-related requests, default 3600

Name	Description
nocache	try to fully disable cache control for file requests
winsymlinks=no	do not resolve symbolic links on file system (Windows only), default true
dirlisting=no	enable/disable directory listing for browsing filesystem (default no)

If necessary, one could bind http server to specific IP address like:

```
new THttpServer("http:192.168.1.17:8080")
```

One also can provide extra arguments for THttpServer itself:

Name	Description
readonly, ro	use server in read-only mode (default)
readwrite, rw	use server in read-write mode
global	let scan global directories for canvases and files (default)
noglobal	disable scan of global directories
basic_sniffer	use basic `TRootSniffer` without support of hist, gpad, graph, tree classes

Example:

```
new THttpServer("http:8080;ro;noglobal")
```

1.2 Registering objects

At any time, one could register other objects with the command:

```
TGraph* gr = new TGraph(10);
gr->SetName("gr1");
serv->Register("graphs/subfolder", gr);
```

One should specify sub-folder name, where objects will be registered. If sub-folder name does not starts with slash `/`, than top-name folder `/Objects/` will be prepended. At any time one could unregister objects:

```
serv->Unregister(gr);
```

THttpServer does not take ownership over registered objects - they should be deleted by user.

If the objects content is changing in the application, one could enable monitoring flag in the browser - then objects view will be regularly updated.

1.3 Accessing file system

THttpServer provides partial access to the files from file system. First of all, JSROOT scripts and files can be accessed via "jsrootsys/" path like "http://localhost:8080/jsrootsys/modules/core.mjs". Files from ROOT install directory can be get via "rootsys/" path like "http://localhost:8080/rootsys/icons/about.xpm". Also files from current directory where ROOT is running can be accessed via "currentdir/" path like "http://localhost:8080/currentdir/file.txt".

If necessary, one can add custom path as well, using THttpServer::AddLocation method:

```
 serv->AddLocation("mydir/", "/home/user/specials");
```

Then files from that directory could be addressed via URL like "http://localhost:8080/mydir/myfile.root"

1.4 Command interface

THttpServer class provide simple interface to invoke command from web browser. One just register command like:

```
serv->RegisterCommand("/DoSomething", "SomeFunction()");
```

Element with name `DoSomething` will appear in the web browser and can be clicked. It will result in `gROOT->ProcessLineSync("SomeF` call.

One could configure argument(s) for the command. For that one should use `%arg1`, `%arg2` and so on identifiers. Like:

```
serv->RegisterCommand("/DoSomething", "SomeFunction(%arg1%,%arg2%)");
```

User will be requested to enter arguments values, when command element clicked in the browser. Example of the command which executes arbitrary string in application via ProcessLine looks like:

```
serv->RegisterCommand("/Process", "%arg1%");
```

When registering command, one could specify icon name which will be displayed with the command.

```
serv->RegisterCommand("/DoSomething", "SomeFunction()", "rootsys/icons/ed_execute.png");
```

In example usage of images from `$ROOTSYS/icons` directory is shown. One could prepend `button;` string to the icon name to let browser show command as extra button. In last case one could hide command element from elements list:

```
serv->Hide("/DoSomething");
```

One can find example of command interface usage in tutorials/http/httpcontrol.C macro.

1.5 Customize user interface

JSROOT is used to implement UI for the THttpServer. Default webpage shows list of registered objects on the left side and drawing area on the right side - see example. JSROOT allows to configure different parameters via URL - like monitoring interval or name of displayed items item=Files/job1.root/hpxpy&opt=colz&monitoring=1000.

Some of such parameters can be configured already on the server:

```
serv->SetItemField("/", "_monitoring", "1000");   // monitoring interval in ms
serv->SetItemField("/", "_drawitem", "Files/job1.root/hpxpy");  // item to draw
serv->SetItemField("/", "_drawopt", "colz");
```

In such case URL parameters are not required - specified item will be displayed automatically when web page is opened. One also can configure to display several items at once. For that one also can configure layout of the drawing area:

```
serv->SetItemField("/", "_layout", "grid2x2");    // layout for drawing area
serv->SetItemField("/", "_drawitem", "[Files/job1.root/hpxpy,Files/job1.root/hpx]");  // items
serv->SetItemField("/", "_drawopt", "[colz,hist]");  // options
```

One also can change appearance of hierarchy browser on the left side of the web page:

```
serv->SetItemField("/", "_browser", "off");    // allowed "fix" (default), "float", "no", "off"
serv->SetItemField("/", "_toptitle", "Custom title");    // title of web page, shown when browser off
```

If necessary, one also can automatically open ROOT file when web page is opened:

```
serv->SetItemField("/", "_loadfile", "currentdir/hsimple.root"); // name of ROOT file to load
```

1.6 Configuring user access

By default, the http server is open for anonymous access. One could restrict the access to the server for authenticated users only. First of all, one should create a password file, using the **htdigest** utility.

```
[shell] htdigest -c .htdigest domain_name user_name
```

It is recommended not to use special symbols in domain or user names. Several users can be add to the ".htdigest" file. When starting the server, the following arguments should be specified:

```
auto serv = new THttpServer("http:8080?auth_file=.htdigest&auth_domain=domain_name");
```

After that, the web browser will automatically request to input a name/password for the domain "domain_name"

Based on authorized accounts, one could restrict or enable access to some elements in the server objects hierarchy, using `THttpServer::Restrict()` method.

For instance, one could hide complete folder from ‘guest’ account:

```
serv->Restrict("/Folder",  "hidden=guest");
```

Or one could hide from all but ‘admin’ account:

```
serv->Restrict("/Folder",  "visible=admin");
```

Hidden folders or objects can not be accessed via http protocol.

By default server runs in readonly mode and do not allow methods execution via ‘exe.json’ or ‘exe.bin’ requests. To allow such action, one could either grant generic access for all or one could allow to execute only special method:

```
serv->Restrict("/Folder/histo1", "allow=all");
serv->Restrict("/Folder/histo1", "allow_method=GetTitle");
```

One could provide several options for the same item, separating them with ‘&’ sign:

```
serv->Restrict("/Folder/histo1", "allow_method=GetTitle&hide=guest");
```

Complete list of supported options could be found in TRootSniffer:Restrict() method documentation.

1.7 Using FastCGI interface

FastCGI is a protocol for interfacing interactive programs with a web server like `Apache`, `lighttpd`, `Microsoft ISS` and many others.

When starting THttpServer, one could specify:

```
serv = new THttpServer("fastcgi:9000");
```

In fact, the FastCGI interface can run in parallel to http server. One can just call:

```
serv = new THttpServer("http:8080");
serv->CreateEngine("fastcgi:9000");
```

One could specify a debug parameter to be able to adjust the FastCGI configuration on the web server:

```
serv->CreateEngine("fastcgi:9000?debug=1");
```

By default 10 threads are used to process FastCGI requests. This number can be changed with “thrds” url parameter:

```
serv->CreateEngine("fastcgi:9000?thrds=20");
```

If `thrds=0` parameter specified, the only thread will be use to received and process all requests.

All user access will be ruled by the main web server. Authorized account names could be used to configure access restriction in THttpServer.

1.7.1 Configure fastcgi with Apache2

Since Apache version 2.4 FastCGI is directly supported - there is no need to compile and install external modules any more. One only need to enable `mod_proxy` and `mod_proxy_fcgi` modules and add following line to **Apache2** configuration file:

```
ProxyPass "/root.app/" "fcgi://localhost:9000/" enablereuse=on
```

More information can be found in FastCGI proxy docu. After restarting apache server one should be able to open address: `http://apache_host_name/root.app/`. There are many ways to configure user authentication in Apache. Example of digest auth for FastCGI server:

```
<Location "/root.app/">
   AuthType Digest
   AuthName "root"
   AuthDigestDomain "/root.app/" "root"
   AuthDigestProvider file
   AuthUserFile "/srv/auth/auth.txt"
   Require valid-user
</Location>
```

1.7.2 Configure fastcgi with lighttpd

An example of configuration file for **lighttpd** server is:

```
server.modules += ( "mod_fastcgi" )
fastcgi.server = (
   "/root.app" =>
     (( "host" => "192.168.1.11",
        "port" => 9000,
        "check-local" => "disable",
        "docroot" => "/"
     ))
)
```

Be aware, that with *lighttpd* one should specify IP address of the host, where ROOT application is running. Address of the ROOT application will be following: `http://lighttpd_host_name/root.app/`. Example of authorization configuration for FastCGI connection:

```
auth.require = ( "/root.app" => (
                   "method" => "digest",
                   "realm" => "root",
                   "require" => "valid-user"
                ) )
```

1.8 Integration with existing applications

In many practical cases no change of existing code is required. Opened files (and all objects inside), existing canvas and histograms are automatically scanned by the server and will be available to the users. If necessary, any object can be registered directly to the server with a THttpServer::Register() call.

Central point of integration - when and how THttpServer get access to data from a running application. By default it is done during the `gSystem->ProcessEvents()` call - THttpServer uses a synchronous timer which is activated every 100 ms. Such approach works perfectly when running macros in an interactive ROOT session.

If an application runs in compiled code and does not contain `gSystem->ProcessEvents()` calls, two method are available.

1.8.1 Asynchronous timer

The first method is to configure an asynchronous timer for the server, like for example:

```
serv->SetTimer(100, kFALSE);
```

Then, the timer will be activated even without any gSystem->ProcessEvents() method call. The main advantage of such method is that the application code can be used without any modifications. But there is no control when access to the application data is performed. It could happen just in-between of **`TH1::Fill()`** calls and an histogram object may be incomplete. Therefore such method is not recommended.

1.8.2 Regular calls of THttpServer::ProcessRequests() method

The second method is preferable - one just inserts in the application regular calls of the THttpServer::ProcessRequests() method, like:

```
serv->ProcessRequests();
```

In such case, one can fully disable the timer of the server:

```
serv->SetTimer(0, kTRUE);
```

1.9 Data access from command shell

The big advantage of the http protocol is that it is not only supported in web browsers, but also in many other applications. One could use http requests to directly access ROOT objects and data members from any kind of scripts.

If one starts a server and register an object like for example:

```
auto serv = new THttpServer("http:8080");
TNamed* n1 = new TNamed("obj", "title");
serv->Register("subfolder", n1);
```

One could request a JSON representation of such object with the command:

```
[shell] wget http://localhost:8080/Objects/subfolder/obj/root.json
```

Then, its representation will look like:

```
{
   "_typename" : "TNamed",
   "fUniqueID" : 0,
   "fBits" : 0,
   "fName" : "obj",
   "fTitle" : "title"
}
```

The following requests can be performed:

Name	Description
`root.bin`	binary data produced by object streaming with `TBufferFile`
`root.json`	ROOT JSON representation for object and objects members
`file.root`	Creates TMemFile with the only object, from ROOT 6.32
`root.xml`	ROOT XML representation
`root.png`	PNG image (if object drawing implemented)
`root.gif`	GIF image
`root.jpeg`	JPEG image
`exe.json`	method execution in the object
`exe.bin`	method execution, return result in binary form
`cmd.json`	command execution
`item.json`	item (object) properties, specified on the server
`multi.json`	perform several requests at once
`multi.bin`	perform several requests at once, return result in binary form

All data will be automatically zipped if '.gz' extension is appended. Like:

```
[shell] wget http://localhost:8080/Objects/subfolder/obj/root.json.gz
```

If the access to the server is restricted with htdigest, it is recommended to use the **curl** program since only curl correctly implements such authentication method. The command will look like:

```
[shell] curl --user "accout:password" http://localhost:8080/Objects/subfolder/obj/root.json --digest -o root.
```

1.9.1 Objects data access in JSON format

Request `root.json` implemented with TBufferJSON class. TBufferJSON generates such object representation, which could be directly used in JSROOT for drawing. `root.json` request returns either complete object or just object member like:

```
    [shell] wget http://localhost:8080/Objects/subfolder/obj/fTitle/root.json
```

The result will be: `"title"`.

For the `root.json` request one could specify the 'compact' parameter, which allow to reduce the number of spaces and new lines without data lost. This parameter can have values from '0' (no compression) till '3' (no spaces and new lines at all). In addition, one can use simple compression algorithm for big arrays. If compact='10', zero values in the begin and at the end of the array will be excluded. If compact='20', similar values or large zero gaps in-between will be compressed. Such array compression support in JSROOT from version 4.8.2.

Usage of `root.json` request is about as efficient as binary `root.bin` request. Comparison of different request methods with TH2 histogram from hsimple.C shown in the table:

Request	Size
root.bin	7672 bytes
root.bin.gz	1582 bytes
root.json	8570 bytes
root.json?compact=3	6004 bytes
root.json?compact=23	5216 bytes
root.json.gz?compact=23	1855 bytes

One should remember that JSON representation always includes names of the data fields which are not present in the binary representation. Even then the size difference is negligible.

`root.json` used in JSROOT to request objects from THttpServer.

1.9.2 Generating images out of objects

For the ROOT classes which are implementing Draw method (like TH1 or TGraph) one could produce images with requests: `root.png`, `root.gif`, `root.jpeg`. For example:

```
[shell] wget "http://localhost:8080/Files/hsimple.root/hpx/root.png?w=500&h=500&opt=lego1" -O lego1.png
```

For all such requests following parameters could be specified:

- **h** - image height
- **w** - image width
- **opt** - draw options

1.9.3 Methods execution

By default THttpServer starts in monitoring (read-only) mode and therefore forbid any methods execution. One could specify read-write mode when server is started:

```
auto serv = new THttpServer("http:8080;rw");
```

Or one could disable read-only mode with the call:

```
serv->SetReadOnly(kFALSE);
```

Or one could allow access to the folder, object or specific object methods with:

```
serv->Restrict("/Histograms", "allow=admin"); // allow full access for user with 'admin' account
serv->Restrict("/Histograms/hist1", "allow=all"); // allow full access for all users
serv->Restrict("/Histograms/hist1", "allow_method=Rebin"); // allow only Rebin method
```

'exe.json' accepts following parameters:

- `method` - name of method to execute
- `prototype` - method prototype (see TClass::GetMethodWithPrototype for details)
- `compact` - compact parameter, used to compress return value
- `_ret_object_` - name of the object which should be returned as result of method execution (used together with remote TTree::Draw call)

Example of retrieving object title:

```
[shell] wget 'http://localhost:8080/Objects/subfolder/obj/exe.json?method=GetTitle' -O title.json
```

Example of TTree::Draw method execution:

```
[shell] wget 'http://localhost:8080/Files/job1.root/ntuple/exe.json?method=Draw&prototype="Option_t*"&opt="px
```

One also used `exe.bin` method - in this case results of method execution will be returned in binary format. In case when method returns temporary object, which should be delete at the end of command execution, one should specify `_destroy_result_` parameter in the URL string:

```
[shell] wget 'http://localhost:8080/Objects/subfolder/obj/exe.json?method=Clone&_destroy_result_' -O clone.js
```

If method required object as argument, it could be posted in binary, JSON or XML format as POST request. If binary form is used, one should specify following parameters:

```
[shell] wget 'http://localhost:8080/hist/exe.json?method=Add&h1=_post_object_&_post_class_=TH1I&c1=10' --post
```

Here is important to specify post object class, which is not stored in the binary buffer. When submitting argument as JSON produced with TBufferJSON::ToJSON method, class is not required:

```
[shell] wget 'http://localhost:8080/hist/exe.json?method=Add&h1=_post_object_json_&c1=10' --post-file=h.json
```

To get debug information about command execution, one could submit `exe.txt` request with same arguments.

1.9.4 Commands execution

If command registered to the server:

```
serv->RegisterCommand("/Folder/Start", "DoSomthing()");
```

It can be invoked with `cmd.json` request like:

```
[shell] wget http://localhost:8080/Folder/Start/cmd.json -O result.txt
```

If command fails, `false` will be returned, otherwise result of `gROOT->ProcessLineSync()` execution.

If command definition include arguments:

```
serv->RegisterCommand("/ResetCounter", "DoReset(%arg1%,%arg2%)");
```

One could specify them in the URL string:

```
[shell] wget http://localhost:8080/ResetCounter/cmd.json?arg1=7&arg2=12 -O result.txt
```

1.9.5 Performing multiple requests at once

To minimize traffic between sever and client, one could submit several requests at once. This is especially useful when big number of small objects should be requested simultaneously. For this purposes `multi.bin` or `multi.json` requests could be used. Both require string as POST data which format as:

```
subfolder/item1/root.json\n
subfolder/item2/root.json\n
subfolder/item1/exe.json?method=GetTitle\n
```

If such requests saved in 'req.txt' file, one could submit it with command:

```
[shell] wget http://localhost:8080/multi.json?number=3 --post-file=req.txt -O result.json
```

For `multi.json` request one could use only requests, returning JSON format (like `root.json` or `exe.json`). Result will be JSON array. For `multi.bin` any kind of requests can be used. It returns binary buffer with following content:

```
[size1 (little endian), 4 bytes] + [request1 result, size1 bytes]
[size2 (little endian), 4 bytes] + [request2 result, size2 bytes]
[size3 (little endian), 4 bytes] + [request3 result, size3 bytes]
```

While POST data in request used to transfer list of multiple requests, it is not possible to submit such kind of requests, which themselves require data from POST block.

To use `multi.json` request from the JavaScript, one should create special 'POST' HTTP request and properly parse it. JSROOT provides special method to do this:

```
import { httpRequest, draw } from './jsrootsys/modules/core.mjs';
let res = await httpRequest("your_server/multi.json?number=3", "multi",
                            "Files/job1.root/hpx/root.json\nFiles/job1.root/hpxpy/root.json\nFiles/job1.root
for (let n = 0; n < res.length; ++n) {
   console.log('Requested element of type', res[n]._typename);
   // draw('drawid', res[n], 'hist');
}
```

Here argument "multi" identifies, that server response should be parsed with `parseMulti()` function, which correctly interprets JSON code, produced by `multi.json` request. When sending such request to the server, one should provide list of objects names and not forget "?number=N" parameter in the request URL string.

1.10 Using unix sockets

Starting from ROOT version 6.28, one can start server with unix socket. Just do:

Just call:

```
[root] new THttpServer("socket:/tmp/root.socket")
```

Name of socket should be unique and not match any existing files.

Most easy way to access `THttpServer` running via unix socket is to configure ssh tunnel:

```
[shell] ssh -L 7654:/tmp/root.socket localhost
```

Once such tunnel is configured one can open following URL in web browser:

```
[shell] xdg-open http://localhost:7654
```

1.11 Websockets supports

Websockets support available starting from ROOT v6.12. Minimal example provided in $ROOTSYS/tutorials/http/ws.C macro.

To work with websockets, subclass of THttpWSHandler should be created and registered to THttpServer:

```
#include "THttpWSHandler.h"

class TUserHandler : public THttpWSHandler {
public:
   TUserHandler(const char *name, const char *title) : THttpWSHandler(name, title) {}

   // provide custom HTML page when open correspondent address
   TString GetDefaultPageContent() override { return "file:ws.htm"; }

   Bool_t ProcessWS(THttpCallArg *arg) override;
};
```

Central method is `TUserHandler::ProcessWS(THttpCallArg *arg)`, where four kinds of websockets events should be handled:

- WS_CONNECT - clients attempts to create websockets, return false when refusing connection
- WS_READY - connection is ready to use, **wsid** can be obtained with `arg->GetWSId()` calls
- WS_DATA - new portion of data received by webcosket
- WS_CLOSE - connection closed by the client, **wsid** is no longer valid

These kinds are coded as method name of THttpCallArg class and can be used like:

```
Bool_t TUserHandler::ProcessWS(THttpCallArg *arg)
{
   if (arg->IsMethod("WS_CONNECT")) {
      return kTRUE; // accept all connections
   }

   if (arg->IsMethod("WS_READY")) {
      SendCharStartWS(arg->GetWSId(), "Init"); // immediately send message to the web socket
      return kTRUE;
   }

   if (arg->IsMethod("WS_CLOSE")) {
      return kTRUE; // just confirm connection
   }

   if (arg->IsMethod("WS_DATA")) {
      TString str = arg->GetPostDataAsString();
      printf("Client msg: %s\n", str.Data());
      SendCharStarWS(arg->GetWSId(), "Confirm");
      return kTRUE;
   }

   return kFALSE; // ignore all other kind of requests
}
```

Instance of **TUserHandler** should be registered to the THttpServer like:

```
THttpServer *serv = new THttpServer("http:8080");
TUserHandler *handler = new TUserHandler("name1","title");
serv->Register(handler);
```

After that web socket connection can be established with the address `ws://host_name:8080/name1/root.websocket` Example client code can be found in `$ROOTSYS/tutorials/http/ws.htm` file. Custom HTML page for websocket handler is specified with `TUserHandler::GetDefaultPageContent()` method returning `"file:ws.htm"`.