SHOW Documentation

Release 0.8.6

Joseph Durel

May 19, 2019

Contents:

SHOW is designed to be an idiomatic library for standalone webserver applications written for modern C++. SHOW is simple in the same way the standard library is simple — it doesn't make any design decisions for the programmer, instead giving them a set of primitives for building an HTTP web application.

Both HTTP/1.0 and HTTP/1.1 are supported. SHOW assumes a modern approach to application hosting, and is intended to be run behind a full reverse proxy such as NGINX. As such, SHOW will not support HTTP/2 or TLS (HTTPS). Instead, you should write your applications to serve local HTTP/1.0 and HTTP/1.1 requests.

SHOW is released under the zlib license. C++11 support and a POSIX operating system (or POSIX compatibility layer) are required.

CHAPTER 1

Tutorial

This shows the basic usage of SHOW; see the examples for a more thorough introduction.

1.1 Including & Compiling

The preferred method of including SHOW is via the CMake package. Once installed somewhere CMake can find it, import and use SHOW in your *CMakeLists.txt* with:

```
FIND_PACKAGE( SHOW REQUIRED COMPONENTS show )
ADD_EXECUTABLE( my_server my_server.cpp )
TARGET_LINK_LIBRARIES( my_server PRIVATE SHOW::show )
```

You should also switch your compiler to C++11 mode with:

```
SET( CMAKE_CXX_STANDARD 11 )
SET( CMAKE_CXX_STANDARD_REQUIRED ON )
SET( CMAKE_CXX_EXTENSIONS OFF )
```

For GCC and Clang, you can either link *show.hpp* to one of your standard include search paths, or use the `-I` flag to tell the compiler where too find the header:

```
clang++ -I "SHOW/src/" ...
```

SHOW is entirely contained in a single header file, you have to do then is include SHOW using `#include <show.hpp>`. With either compiler you'll also need to specify C++11 support with `-std=c++11`.

1.2 Creating a Server

To start serving requests, first create a *`server`* object:

```
show::server my_server{
    "0.0.0.0",  // IP address on which to serve
    9090,       // Port on which to serve
};
```

That's it, you've made a server that sits there forever until it gets a connection, then hangs. Not terribly useful, but that's easy to fix.

1.3 Handling a Connection

For each call of `my_server.serve()` a single *connection* object will be returned or a *connection_timeout* thrown. You may want to use something like this:

```
while( true )
    try
    {
        show::connection connection{ my_server.serve() };
        // handle request(s) here
    }
    catch( const show::connection_timeout& ct )
    {
        std::cout
            << "timed out waiting for a connection, looping..."
            << std::endl
        ;
        continue;
    }
```

The server listen timeout can be a positive number, 0, or -1. If it is -1, the server will continue listening until interrupted by a signal; if 0, *server::serve()* will throw a *connection_timeout* immediately unless connections are available.

The connection is now independent from the server. You can adjust the connection's timeout independently using *connection::timeout()*. You can also pass it off to a worker thread for processing so your server can continue accepting other connections; this is usually how you'd implement a real web application.

1.4 Reading Requests

request objects have a number of `const` fields containing the HTTP request's metadata; you can see descriptions of them all in the docs for the class.

Note that these fields do not include the request content, if any. This is because HTTP allows the request content to be streamed to the server. In other words, the server can interpret the headers then wait for the client to send data over a period of time. For this purpose, *request* inherits from `std::streambuf`, implementing the read/get functionality. You can use the raw `std::streambuf` methods to read the incoming data, or create a `std::istream` from the request object for `std::cin`-like behavior.

For example, if your server is expecting the client to *POST* a single integer, you can use:

```
show::request request{ test_server.serve() };

std::istream request_content_stream{ &request };

```

(continues on next page)

(continued from previous page)

```
int my_integer;
request_content_stream >> my_integer;
```

Please note that the above is not terribly safe; production code should include various checks to guard against buggy or malignant clients.

Also note that individual request operations may timeout, so the entire serve code should look like this:

```
while( true )
    try
    {
        show::connection connection{ my_server.serve() };
        try
        {
            show::request request{ connection };
            std::istream request_content_stream{ &request };
            int my_integer;
            request_content_stream >> my_integer;
            std::cout << "client sent " << my_integer << "\n";
        }
        catch( const show::client_disconnected& ct )
        {
            std::cout << "got a request, but client disconnected!" << std::endl;
        }
        catch( const show::connection_timeout& ct )
        {
            std::cout << "got a request, but client timed out!" << std::endl;
        }
    }
    catch( const show::connection_timeout& ct )
    {
        std::cout << "timed out waiting for a connection, looping..." << std::endl;
        continue;
    }
```

If this feels complicated, it is. Network programming like this reveals the worst parts of distributed programming, as there's a lot that can go wrong between the client and the server.

Another thing to keep in mind is that HTTP/1.1 — and HTTP/1.0 with an extension — allow multiple requests to be pipelined on the same TCP connection. SHOW can't know with certainty where on the connection one request ends and another starts — it's just the nature of pipelined HTTP. Sure, the *Content-Length* header could be used, and chunked transfer encoding has well-established semantics, but if the client uses neither it is up to your application to figure out the end of the request's content. In general, you should reject requests whose length you can't readily figure out, but SHOW leaves that decision up to the programmer. But you should never try to create a *request* from a *connection* before you've finished reading the content from a previous request.

See also:

- `std::streambuf` on cppreference.com
- `std::istream` on cppreference.com
- `std::cin` on cppreference.com

1.5 Sending Responses

Sending responses is slightly more involved than reading basic requests. Say you want to send a "Hello World" message for any incoming request. First, start with a string containing the response message:

```
std::string response_content{ "Hello World" };
```

Next, create a headers object to hold the content type and length headers (note that header values must be strings):

```
show::headers_t headers{
    { "Content-Type", { "text/plain" } },
    { "Content-Length", {
        std::to_string( response_content.size() )
    } }
};
```

Since it's a `std::map`, you can also add headers to a `headers_t` like this:

```
headers[ "Content-Type" ].push_back( "text/plain" );
```

Then, set the HTTP status code for the response to the generic *200 OK*:

```
show::response_code code{
    200,
    "OK"
};
```

Creating a response object requires the headers and response code to have been decided already, as they are marshalled (serialized) and buffered for sending as soon as the object is created. A response object also needs to know which request it is in response to. While there's nothing preventing you from creating multiple responses to a single request this way, most of the time that will break your application.

Create a response like this:

```
show::response response{
    connection,
    show::http_protocol::HTTP_1_0,
    code,
    headers
};
```

Finally, send the response content. Here, a `std::ostream` is used, as *`response`* inherits from and implements the write/put functionality of `std::streambuf`:

```
std::ostream response_stream{ &response };
response_stream << response_content;
```

See also:

- `std::map` on cppreference.com
- `std::ostream` on cppreference.com
- `std::streambuf` on cppreference.com

CHAPTER 2

Types

2.1 Main Types

The public interfaces to the main SHOW classes are documented on the following pages:

2.1.1 Server

class server

The server class serves as the basis for writing an HTTP application with SHOW. Creating a server object allows the application to handle HTTP requests on a single IP/port combination.

server(**const** std::string &*address*, unsigned int *port*, int *timeout* = -1)

Constructs a new server to serve on the given IP address and port. The IP address will typically be `localhost`/`0.0.0.0`/`::`. The port should be some random higher-level port chosen for the application.

The timeout is the maximum number of seconds *`serve()`* will wait for an incoming connection before throwing *`connection_timeout`*. A value of 0 means that *`serve()`* will return immediately if there are no connections waiting to be served; -1 means *`serve()`* will wait forever (until the program is interrupted).

~server()

Destructor for a server; any existing connections made from this server will continue to function

connection **serve**()

Either returns the next connection waiting to be served or throws *`connection_timeout`*.

const std::string &**address**() **const**

Get the address this server is servering on

unsigned int **port**() **const**

Get the port this server is servering on

int **timeout**() **const**

Get the current timeout of this server

int **timeout** (int)
Set the timeout of this server to a number of seconds, 0, or -1

2.1.2 Connection

class connection
Objects of this type represent a connection between a single client and a server. A connection object can be used to generate *request* objects; one in the case of HTTP/1.0 or multiple in the case of HTTP/1.1.

The connection class has no public constructor (besides the move constructor), and can only be created by calling *server::serve()*.

connection (*connection*&&)
Explicit move constructor as one can't be generated for this class

~connection ()
Destructor for a connection, which closes it; any requests or responses created on this connection can no longer be read from or written to

const std::string &**client_address** () **const**
The IP address of the connected client

unsigned int **client_port** () **const**
The port of the connected client

const std::string &**server_address** () **const**
The address of the server handling the connection

unsigned int **server_port** () **const**
The port of the server handling the connection

int **timeout** () **const**
Get the current timeout of this connection, initially inherited from the server the connection is created from

int **timeout** (int)
Set the timeout of this connection independently of the server; the argument is a number of seconds, 0, or -1

See also:

- *server::timeout()*

2.1.3 Request

class request : **public** std::streambuf
Represents a single request sent by a client. Inherits from `std::streambuf`, so it can be used as-is or with a `std::istream`.

See also:

- `std::streambuf` on cppreference.com
- `std::istream` on cppreference.com

enum content_length_flag
A utility type for *unknown_content_length()* with the values:

Value	Evaluates to
`NO`	`false`
`YES`	`true`
`MAYBE`	`true`

const std::string &**client_address**() **const**
The IP address of the client that sent the request

const unsigned int **client_port**() **const**
The port of the client that sent the request

bool **eof**() **const**
Returns whether or not the request, acting as a `std::streambuf`, has reached the end of the request contents. Always returns `false` if the content length is unknown.

See also:

- *unknown_content_length()*

request(*connection*&)
Constructs a new request on a connection. Blocks until a connection is sent, the connection timeout is reached, or the client disconnects. May also throw *request_parse_error* if the data sent by the client cannot be understood as an HTTP request.

See also:

- *connection_timeout*
- *client_disconnected*

request(*request*&&)
Explicit move constructor as one can't be generated for this class

void **flush**()
Flushes the request contents from the buffer, putting it in a state where the next request can be extracted. It is only safe to call this function if *unknown_content_length()* evaluates to `false`.

http_protocol **protocol**() **const**
The HTTP protocol used by the request. If `NONE`, it's usually safe to assume HTTP/1.0. If `UNKNOWN`, typically either a *400 Bad Request* should be returned, just assume HTTP/1.0 to be permissive, or try to interpret something from *protocol_string()*.

const std::string &**protocol_string**() **const**
The raw protocol string sent in the request, useful if *protocol()* is `UNKNOWN`

const std::string &**method**() **const**
The request method as a capitalized ASCII string. While the HTTP protocol technically does not restrict the available methods, typically this will be one of the following:

GET	Common methods
POST	
PUT	
DELETE	
OPTIONS	Useful for APIs
PATCH	Relatively uncommon methods
TRACE	
HEAD	
CONNECT	

See also:

- List of common HTTP methods on Wikipedia for descriptions of the methods

const std::vector<std::string> &**path**() **const**

The request path separated into its elements, each of which has been URL- or percent-decoded. For example:

```
/foo/bar/hello+world/%E3%81%93%E3%82%93%E3%81%AB%E3%81%A1%E3%81%AF
```

becomes:

```
{
    "foo",
    "bar"
    "hello world",
    ""
}
```

const *query_args_type* &**query_args**() **const**

The request query arguments. SHOW is very permissive in how it parses query arguments:

Query string	Interpreted as
`?foo=1&bar=2`	`{ { "foo", { "1" } }, { "bar", { "2" } } }`
`?foo=bar=baz`	`{ { "foo", { "baz" } }, { "bar", { "baz" } } }`
`?foo=&bar=baz`	`{ { "foo", { "" } }, { "bar", { "baz" } } }`
`?foo&bar=1&bar=2`	`{ { "foo", { "" } }, { "bar", { "1", "2" } } }`

const *headers_type* &**headers**() **const**

The request headers

See also:

- List of common HTTP headers on Wikipedia

content_length_flag **unknown_content_length**() **const**

Whether the content length of the request could be interpreted

This member may be a bit confusing because it is "*un*-known" rather than "know". It's convenient for *content_length_flag* to evaluate to a boolean value, but there are two possible reasons the content length would be unknown. Either

1. the request did not send a *Content-Length* header, or
2. the value supplied is not an integer or multiple *Content-Length* headers were sent.

In many languages (including C++), 0 is `false` and any other value is `true`; so the boolean value needs to be `false` for a known content length and `true` for anything else.

unsigned long long **`content_length`**`()` **`const`**
: The number of bytes in the request content; only holds a meaningful value if *`unknown_content_length()`* is YES/`true`

2.1.4 Response

`class response` : **`public`** std::streambuf
: Represents a single response to a request. Inherits from `std::streambuf`, so it can be used as-is or with a `std::ostream`.

SHOW does not prevent mutliple response from being created or sent for a single request. Most of the time this is something that would break the application; however, under certain conditions in HTTP/1.1 multiple *100*-type responses can be sent before a final *200*+ response.

See also:

- `std::streambuf` on cppreference.com
- `std::ostream` on cppreference.com

`response`(*connection*&, *http_protocol*, **`const`** *response_code*&, **`const`** headers_t&)
: Constructs a new response to the client who made a connection. The protocols, response code, and headers are immediately buffered and cannot be changed after the response is created, so they have to be passed to the constructor.

`~response``()`
: Destructor for a response object; ensures the response is flushed

`virtual` void **`flush`**`()`
: Ensure the content currently written to the request is sent to the client

2.2 Support Types

`enum http_protocol`
: Symbolizes the HTTP protocols understood by SHOW. The enum members are:

`HTTP_1_0`	HTTP/1.0
`HTTP_1_1`	HTTP/1.1
`NONE`	The request did not specify a protocol version
`UNKOWN`	The protocol specified by the request wasn't recognized

There is no `HTTP_2` as SHOW is not intended to handle HTTP/2 requests. These are much better handled by a reverse proxy such as NGINX, which will convert them into HTTP/1.0 or HTTP/1.1 requests for SHOW.

`class response_code`
: A simple utility `struct` that encapsulates the numerical code and description for an HTTP status code. An object of this type can easily be statically initialized like so:

```
show::response_code rc = { 404, "Not Found" };
```

See the list of HTTP status codes on Wikipedia for an easy reference for the standard code & description values.

The two fields are defined as:

unsigned short **code**

std::string **description**

class query_args_type

An alias for `std::map< std::string, std::vector< std::string > >`, and can be statically initialized like one:

```
show::query_args_type args{
    { "tag", { "foo", "bar" } },
    { "page", { "3" } }
};
```

This creates a variable `args` which represents the query string `?tag=foo&tag=bar&page=3`.

See also:

- `std::map` on cppreference.com
- `std::vector` on cppreference.com

class headers_type

An alias for `std::map< std::string, std::vector< std::string >, show::_less_ignore_case_ASCII >`, where `show::_less_ignore_case_ASCII` is a case-insensitive compare for `std::map`.

While HTTP header names are typically given in `Dashed-Title-Case`, they are technically case-insensitive. Additionally, in general a given header name may appear more than once in a request or response. This type satisfies both these constraints.

Headers can be statically initialized:

```
show::headers_type headers{
    { "Content-Type", { "text/plain" } },
    { "Set-Cookie", {
        "cookie1=foobar",
        "cookie2=SGVsbG8gV29ybGQh"
    } }
};
```

See also:

- `std::map` on cppreference.com
- `std::vector` on cppreference.com

2.3 Throwables

Not all of these strictly represent an error state when throw; some signal common situations that should be treated very much in the same way as exceptions. SHOW's throwables are broken into two categories — connection interruptions and exceptions.

2.3.1 Connection interruptions

class connection_interrupted
A common base class for both types of connection interruptions. Note that this does not inherit from `std::exception`.

class connection_timeout : **public** *connection_interrupted*
An object of this type will be thrown in two general situations:

- A server object timed out waiting for a new connection
- A connection, request, or response timed out reading from or sending to a client

In the first situation, generally the application will simply loop and start waiting again. In the second case, the application may want to close the connection or continue waiting with either the same timoute or some kind of falloff. Either way the action will be application-specific.

class client_disconnected : **public** *connection_interrupted*
This is thrown when SHOW detects that a client has broken connection with the server and no further communication can occur.

2.3.2 Exceptions

See also:

- `std::runtime_error` on cppreference.com

class socket_error : **public** std::runtime_error
An unrecoverable, low-level error occurred inside SHOW. If thrown while handling a connection, the connection will no longer be valid but the server should be fine. If thrown while creating or working with a server, the server object itself is in an unrecoverable state and can no longer serve.

The nature of this error when thrown by a server typically implies trying again will not work. If the application is designed to serve on a single IP/port, you will most likely want to exit the program with an error.

class request_parse_error : **public** std::runtime_error
Thrown when creating a request object from a connection and SHOW encounters something it can't manage to interpret into a *`request`*.

As parsing the offending request almost certainly failed midway, garbage data will likely in the connection's buffer. Currently, the only safe way to handle this exception is to close the connection.

class response_marshall_error : **public** std::runtime_error
Thrown by *`response`*'s constructor when the response arguments cannot be marshalled into a valid HTTP response:

- One of the header names is an empty string
- One of the header names contains a character other than A-Z, a-z, 0-9, or -
- Any header value is an empty string

class url_decode_error : **public** std::runtime_error
Thrown by *`url_decode()`* when the input is not a valid URL- or percent-encoded string.

Note: *`url_encode()`* shouldn't throw an exception, as any string can be converted to percent-encoding.

CHAPTER 3

Functions

std::string **`url_encode`**(**`const`** std::string &*o*, bool *use_plus_space* = true)
: URL-encode a string `o`, escaping all reserved, special, or non-ASCII characters with percent-encoding.

  If `use_plus_space` is `true`, spaces will be replaced with + rather than `%20`.

std::string **`url_decode`**(**`const`** std::string&)
: Decode a URL- or percent-encoded string. Throws *`url_decode_error`* if the input string is not validly encoded.

CHAPTER 4

Constants

All constants are `const`-qualified.

4.1 Version

The `version` sub-namespace contains information about the current SHOW version. It has the following members:

std::string **`name`**
: The proper name of SHOW as it should appear referenced in headers, log messages, etc.

int **`major`**
: The major SHOW version (`X.0.0`)

int **`minor`**
: The minor SHOW version (`0.X.0`)

int **`revision`**
: The SHOW version revision (`0.0.X`)

std::string **`string`**
: A string representing the major, minor, and revision version numbers

CHAPTER 5

Separate Utilities

These are some useful utilities included with SHOW, but in their own header files so they're optional.

5.1 Base-64 Encoding

These are utilities for handling base64-encoded strings, very commonly used for transporting binary data in web applications. They are included in *show/base64.hpp*.

string **base64_encode**(**const** std::string &*o*, **const** char **chars* = *base64_chars_standard*)
Base64-encode a string `o` using the character set `chars`, which must point to a `char` array of length 64.

See also:

- *base64_chars_standard*
- *base64_chars_urlsafe*

std::string **base64_decode**(**const** std::string &*o*, **const** char **chars* = *base64_chars_standard*, show::base64_flags *flags* = 0x00)
Decode a base64-encoded string `o` using the character set `chars`, which must point to a `char` array of length 64. Throws a *base64_decode_error* if the input is not encoded against `chars` or has incorrect padding.

Incorrect padding can be ignored by passing `show::base64_ignore_padding` as the `flags` argument.

See also:

- *base64_chars_standard*
- *base64_chars_urlsafe*

class base64_decode_error : **public** std::runtime_error
Thrown by *base64_decode()* when the input is not valid base64.

Note: *base64_encode()* shouldn't throw an exception, as any string can be converted to base-64.

char ***base64_chars_standard**
The standard set of base64 characters for use with *base64_encode()* and *base64_decode()*

char ***base64_chars_urlsafe**
The URL_safe set of base64 characters for use with *base64_encode()* and *base64_decode()*, making the following replacements:

- + → -
- / → _

5.2 Multipart Content Support

Multipart content is used to send a number of data segments each with their own separate headers. As such, text and binary data can be mixed in the same message.

SHOW provides the following utilities for parsing multipart requests in *show/multipart.hpp*. Typically, the `Content-Type` header for these types of requests will look something like:

```
Content-Type: multipart/form-data; boundary=AaB03x
```

The boundary string must be extracted from the header to pass to *multipart*'s constructor. A simple example with no error handling:

```
const auto& header_value = request.headers()[ "Content-Type" ][ 0 ];
auto content_supertype = header_value.substr( 0, header_value.find( "/" ) )
if( content_supertype == "multipart" )
{
    show::multipart parser{
        request,
        header_value.substr( header_value.find( "boundary=" ) + 9 )
    };

    // Iterate over multipart data ...
}
else
    // Process data as single message ...
```

class multipart
class description

template<class **String**>
multipart (std::streambuf &*buffer*, *String* &&*boundary*)
Constructs a new multipart content parser.

The supplied buffer will typically be a *request* object, but because multipart content can contain other multipart content recursively it can also be a *show::multipart::segment*. The `boundary` variable is a perfectly-forwarded boundary string for the multipart data.

Throws `std::invalid_argument` if the boundary is an empty string.

See also:

- `std::invalid_argument` on cppreference.com

multipart::*iterator* **begin**()
Returns an iterator pointing to the first segment in the multipart content. Calling this more than once on the same *multipart* throws a std::logic_error.

See also:

- std::logic_error on cppreference.com

multipart::*iterator* **end**()
Returns an iterator representing the end of the multipart content.

const std::string &**boundary**()
The boundary string the *multipart* is using to split the content

const std::streambuf &**buffer**()
The buffer the *multipart* is reading from

class *multipart*::**iterator**
Iterator type for iterating over multipart data segments. Implements most of input iterator functionality, except that its value_type (*multipart::segment*) cannot be copied.

class *multipart*::**segment** : **public** std::streambuf
Represents a segment of data in the multipart content being iterated over. Cannot be copied.

const *headers_type* &**headers**()
The headers for this individual segment of data; does not include the request's headers.

class multipart_parse_error : **public** *request_parse_error*
Thrown when creating a *multipart*, iterating over parts, or reading from a *multipart::segment* whenever the content violates the multipart format.

CHAPTER 6

Indices and tables

- genindex
- search

Index

S