HttpServer

Included in the C++ classes is a simple HTTP server. This can be used to serve up files from the local file system contained on your ESP32. In addition, a programmer defined callback mechanism is provided so that you can write an application which will receive notifications when a client request arrives and give you the opportunity to send a programmatic response back. This is targeted at handling inbound REST calls. The `HttpServer` also supports the WebSocket protocol to create a persistent connection between your client (browser) and the ESP32.

At the simplest level, you create an instance of an `HttpServer` class and ask it to start:

```
HttpServer httpServer();
httpServer.start(80);
```

The call to `start()` takes the port number that you wish to listen upon. The `start()` command is non-blocking meaning that the HTTP server will startup and listen for incoming requests in the background.

If you wish to define callback functions, you can invoke the `addPathHandler()` function on the `HttpServer` object instance. The syntax for this command is:

```
addPathHandler(std::string method, std::string path, wsHandlerFunction)
```

The `method` parameter is the HTTP method your are processing. This will commonly be GET or POST. The path parameter is the incoming path contained in the request which will trigger the callback. Note that the callback will only be triggered when **both** the `method` and `path` match.

The callback function has the following signatire:

```
void wsHandlerFunction(HttpRequest* pRequest, HttpResponse* pResponse)
```

Here is an example of a callback function:

```
static void helloWorldHandler(HttpRequest *pRequest, HttpResponse *pResponse) {
   ESP_LOGD(LOG_TAG, "In hello world handler");
   pResponse->setStatus(HttpResponse::HTTP_STATUS_OK, "OK");
   pResponse->addHeader(HttpRequest::HTTP_HEADER_CONTENT_TYPE, "text/plain");
   pResponse->sendData("Hello back");
   pResponse->close_cpp();
}
```

and here we define an HTTP Server to process the request:

```
HttpServer* pHttpServer = new HttpServer();
pHttpServer->addPathHandler(
   HttpRequest::HTTP_METHOD_GET,
   "/helloWorld",
   helloWorldHandler);
pHttpServer->start(80);
```

We register the handler at the path "`/helloWorld`". As such, a browser request targeted at:

```
http://<ESP32_IP>/helloWorld
```

will trigger the processing.

The `HttpRequest` object describes the nature of the request and the `HttpResponse` object allows you to provide a response.

The methods on `HttpRequest` are:

- std::string getBody()
- std::string getHeader(std::string name)
- std::map<std::string, std::string> getHeaders()
- std::string getMethod()
- std::string getPath()
- std::map<std::string, std::string> getQuery()
- Socket getSocket()
- std::string getVersion()
- WebSocket* getWebSocket()
- bool isWebsocket()
- std::vector<std::string> pathSplit()

and the methods on HttpResponse are:

- void addHeader(std::string name, std::string value)
- void close_cpp()
- std::string getHeader(std::string name)
- std::map<std::string, std::string> getHeaders()
- void sendData(std::string data)
- void setStatus(int status, std::string message)

WebSocket

Within an HTTP callback handler, you can query the incoming request whether or not it pertains to a new WebSocket creation. To do this, you use the `HttpServer#isWebSocket()` call. If this returns true, then you have a WebSocket. When you receive a WebSocket, you will *not* be passed an `HttpResponse` object because we are now no longer processing a simple request/response transaction. Instead, we are working with a persistent connection between the `HttpServer` and the client.

If you detect that your handler has been invoked because of a new WebSocket connection, then you can invoke the `HttpServer#getWebSocket()` function which will return you a reference to a WebSocket object. The WebSocket object has the following methods:

- `void close()` – Close the web socket.
- `Socket getSocket()` - Retrieve the underlying TCP/IP socket.
- `void send(std::string data)` – Send data to the partner.
- `void setHandler(WebSocketHandler handler)` – Set a handler for callback events.

Since an incoming WebSocket data transmission can occur at any time, you register a handler/callback to be informed when new data has arrived. You register this handler with the `setHandler()` function. This takes as input an instance of a class of type `WebSocketHandler` which is a class with virtual members. You can provide implementations for:

- `void onClose()` – Called when the WebSocket is closed.
- `void onError(std::string error)` – Called when an error with the WebSocket is detected.
- `void onMessage(WebSocketInputStreambuf *streambuf)` – Called when a new message is retrieved.

The `onMessage()` callback needs a little explanation. In the C++ standard library we have the concept of streams of data. This can be the source or sink of data where we don't have to consume or generate all the data in one single unit. For our `onMessage()` callback, we might not want to receive all the data in store it into RAM. For example, if we are receiving the content of a megabyte file from the partner, we would be in real trouble as the ESP32 only has 512K of RAM. To solve this puzzle, we are given an object that implements the `std::streambuf` interface architected by the C++ standard library. From this object we can create a `std::istream` input stream and start pulling the data as we need. We can also use an interesting form where we can create an output stream and pass a `std::streambuf` instance to that output stream. The result is that the output stream will work with the streambuf to consume its presented content and transmit that to the destination of the stream. For example:

```
void onMessage(WebSocketInputStreambuf *webSocketStreambuf) {
   // Create an output stream
   oStream << webSocketStreambuf;
}
```

This recipe does **not** read the data passed from the web socket into RAM before pushing it into to the output stream. Instead, it will read a part and push a part and repeat until all the stream data from the web socket has been consumed.