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 requests. The `HttpServer` also supports the WebSocket protocol to create a persistent connection between your client (browser) and the ESP32. Through the WebSocket you can send data in either direction.

Note: These classes use C++ Exception handling which means that you must enable that capability through "`make menuconfig`". This can be found at:

```
Compiler options -> Enable C++ exceptions
```

To use the HTTP Server, 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 for in-coming requests. The `start()` command is non-blocking meaning that the HTTP server will startup and then listen for incoming requests in the background.

If you wish to define callback functions that are invoked when a request arrives, you can call the `addPathHandler()` function on the `HttpServer` object instance. The syntax for this command is:

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

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 signature:

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

Here is an example of a callback function:

```
static void helloWorldHandler(HttpRequest* pRequest, HttpResponse* pResponse) {
   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()` – Get the body of the request message for PUT and POST.
- `std::string getHeader(std::string name)` – Get the value of a named header.
- `std::map<std::string, std::string> getHeaders()` – Get a map of all the headers and their values.
- `std::string getMethod()` – Get the method that passed in the request.
- `std::string getPath()` – Get the path of the request.
- `std::map<std::string, std::string> getQuery()` – Get the query string parts of the request.
- `Socket getSocket()` – Get the underlying TCP/IP socket.
- `std::string getVersion()` – Get the HTTP version passed in the request.
- `WebSocket* getWebSocket()` – Get the WebSocket object (assuming the request was the creation of a WebSocket).
- `bool isWebsocket()` – Return true if the request caused the creation of a new web socket.
- `std::map<std::string, std::string> parseForm()` - Parse the body data to return a map of the form name/value pairs.
- `std::vector<std::string> pathSplit()` – Split the path into its constituent parts.
- `std::string urlDecode(std::string str)` – Decode a URL string or body.

and the methods on `HttpResponse` are:

- `void addHeader(std::string name, std::string value)` – Add a header to the response.
- `void close()` – Close/complete the response.

- `std::string getHeader(std::string name)` – Get the value of a header we previously added.
- `std::map<std::string, std::string> getHeaders()` – Get all the headers that we previously added.
- `void sendData(std::string data)` – Send data to the partner.
- `void setStatus(int status, std::string message)` – Set the status value (eg. 200) for the response.

WebSocket

Within an HTTP callback handler, you can query the incoming request and 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 are informed that you have a WebSocket, the web socket protocol exchanges have already been performed and there is no need for you to do any other work in order to exchange data.

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.

Default implementations are provided for the functions that you choose not to code.

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 and store it into RAM. For example, if we are receiving the content of a megabyte sized 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.

WebSocket File Transfer

Imagine that you wish to upload a file to your ESP32 file system. While you can use HTTP POST to push a file, there is perhaps a better solution. We have create a class that can receive incoming files through a WebSocket. When you have received a new client initiated WebSocket at your HTTP server, you can associate the WebSocket with an instance of a class called WebSocketFileTransfer. This will then own the web socket and process incoming files and save their results to the ESP32 file system.

For example:

```
WebSocketFileTransfer webSocketFileTransfer("/spiflash");
void uploadOpenHandler(HttpRequest* pRequest, HttpResponse *pResponse) {
   if (pRequest->isWebSocket()) {
       webSocketFileTransfer.start(pRequest->getWebSocket());
   }
}

HttpServer *pHttpServer = new HttpServer();
pHttpServer->addPathHandler("GET", "/upload", uploadOpenHandler);
```

When we construct a WebSocketFileTransfer instance, we tell it the root of the file system that files shall be written relative to.

Now let us consider the format of the incoming data. **Two** or more messages will be received over the WebSocket. The first message is a JSON string that describes the file about to be received. The second message is the data content of the file itself. This can be multiple messages containing parts of the file or one message containing the whole file.

The JSON format is:

```
{
   "name":   <fileName to be created>,
   "length": <length of the file in bytes> // This is optional.
}
```

An example of usage of this function would be to transmit a ZIP file from a PC to the ESP32. For example, we could write a Node.js application that reads a ZIP, unpacks the zip and for each file contained within, sends it over a WebSocket to the ESP32. When we run this, the result will be a copy of the content off the ZIP on the ESP32 file system.