

PlatformIO is a Ukrainian Project ❤️ [Please help us stop the war.](#) Ad

# mathieucarbou/ESPAsyncWebServer

by **Mathieu Carbou**

LIBRARY

TOP 100

3.6.0 • **Public** • Published 2 months ago •

[http](#) [async](#) [websocket](#) [webserver](#)

Asynchronous HTTP and WebSocket Server Library for ESP32, ESP8266 and RP2040. Supports: WebSocket, SSE, Authentication, Arduino Json 7, File Upload, Static File serving, URL Rewrite, URL Redirect, etc.

 [Readme](#)

 [Installation](#)

 [Compatibility](#)

 [Examples](#) 7

 [Insights](#)

## ESPAsyncWebServer

release **v3.6.0**  PlatformIO **v3.6.0**

License **LGPL 3.0** Contributor Covenant **2.1**

 Build **passing** last commit **about 2 months ago**  Gitpod **Ready-to-Code**

Asynchronous HTTP and WebSocket Server Library for ESP32, ESP8266 and RP2040 Supports: WebSocket, SSE, Authentication, Arduino Json 7, File Upload, Static File serving, URL Rewrite, URL

Redirect, etc.

This fork is based on [yubox-node-org/ESPAsyncWebServer](https://github.com/yubox-node-org/ESPAsyncWebServer) and includes all the concurrency fixes.

- [Changes in this fork](#)
- [Dependencies](#)
- [Performance](#)
- [Important recommendations for build options](#)
- [AsyncWebSocketMessageBuffer and makeBuffer\(\)](#)
- [How to replace a response](#)
- [How to use Middleware](#)
- [How to use authentication with AsyncAuthenticationMiddleware](#)
- [Migration to Middleware to improve performance and memory usage](#)
- [Original Documentation](#)

## Changes in this fork

---

- (bug) A lot of bug fixes
- (ci) Better CI with a complete matrix of Arduino versions and boards
- (ci) Deployed in PlatformIO registry and Arduino IDE library manager
- (feat) **Arduino 3 / ESP-IDF 5** compatibility
- (feat) **ArduinoJson 7** compatibility
- (feat) **ESP32 / ESP8266 / RP2040** support
- (feat) **MessagePack** support
- (feat) **Middleware** support with pre-built middlewares for authentication, authorization, rate limiting, logging, cors, etc.
- (feat) **Request attributes** to store data on the request object
- (feat) **Response header** control and override
- (feat) **Response override**: support the ability to replace a previously sent response by another one
- (feat) **Resumable download** support using HEAD and bytes range
- (feat) **StreamConcat** example to show how to stream multiple files in one response
- (feat) Removed ESPIDF Editor (this is not the role of a web server)

library to do that - get the source files from the original repos if required)

- (perf) `AsyncTCPSock` support: AsyncTCP can be ignored and AsyncTCPSock used instead
- (perf) `char*` overloads to avoid using `String`
- (perf) `DEFAULT_MAX_WS_CLIENTS` to change the number of allowed WebSocket clients and use `cleanupClients()` to help cleanup resources about dead clients
- (perf) `setCloseClientOnQueueFull(bool)` which can be set on a client to either close the connection or discard messages but not close the connection when the queue is full
- (perf) `SSE_MAX_QUEUED_MESSAGES` to control the maximum number of messages that can be queued for a SSE client
- (perf) `WS_MAX_QUEUED_MESSAGES`: control the maximum number of messages that can be queued for a WebSocket client
- (perf) in-flight buffer control and queue congestion avoidance to help to improve parallel connections handling, high volume data transfers and mitigate poor implemented slow user-code callbacks delays on connections handling
- (perf) Code size improvements
- (perf) Lot of code cleanup and optimizations
- (perf) Performance improvements in terms of memory, speed and size

---

## WARNING: Important notes about future version 4.x

---

This ESPAsyncWebServer fork is now at version 3.x, where we try to keep the API compatibility with original project as much as possible.

Next version 4.x will:

1. Drop support for ESP8266, which goes EOL in a few years. All ESP8266 boards can be replaced by equivalent ESP32 boards.
2. Drop support for Arduino 2.x and ESP-IDF 4.x. The library will be compatible with Arduino 3.x and ESP-IDF 5.x.
3. Drop support for ArduinoJson 5.x and 6.x. The library will be compatible with ArduinoJson 7.x.

So if you need one of these feature, you will have to stick with 3.x or another fork.

## Dependencies

---

[!WARNING] The library name was changed from `ESP Async WebServer` to `ESPAsyncWebServer` as per the Arduino Lint recommendations, but its name had to stay `ESP Async WebServer` in Arduino Registry.

PlatformIO / pioarduino:

```
lib_compat_mode = strict
lib_ldf_mode = chain
lib_deps = mathieucarbou/ESPAsyncWebServer @ 3.6.0
```

Dependencies:

- **ESP32 with AsyncTCP:** `mathieucarbou/AsyncTCP @ 3.3.2` Arduino IDE: <https://github.com/mathieucarbou/AsyncTCP#v3.3.2>
- **ESP32 with AsyncTCPSock:** <https://github.com/mathieucarbou/AsyncTCPSock/archive/refs/tags/v1.0.3-dev.zip>
- **ESP8266:** `esphome/ESPAsyncTCP-esphome @ 2.0.0` Arduino IDE: <https://github.com/mathieucarbou/esphome-ESPAsyncTCP#v2.0.0>
- **RP2040:** `khoih-prog/AsyncTCP_RP2040W @ 1.2.0` Arduino IDE: [https://github.com/khoih-prog/AsyncTCP\\_RP2040W#v1.2.0](https://github.com/khoih-prog/AsyncTCP_RP2040W#v1.2.0)

### AsyncTCPSock

AsyncTCPSock can be used instead of AsyncTCP by excluding AsyncTCP from the library dependencies and adding AsyncTCPSock instead:

```
lib_compat_mode = strict
lib_ldf_mode = chain
lib_deps =
    ; mathieucarbou/AsyncTCP @ 3.3.2
    https://github.com/mathieucarbou/AsyncTCPSock/archive/refs/tags,
```

```
mathieucarbou/ESPAsyncWebServer @ 3.6.0
lib_ignore =
  AsyncTCP
mathieucarbou/AsyncTCP
```

## Performance

---

Performance of [mathieucarbou/ESPAsyncWebServer @ 3.6.0](#):

```
> brew install autocannon
> autocannon -c 10 -w 10 -d 20 http://192.168.4.1
```

With [mathieucarbou/AsyncTCP @ 3.3.2](#)

```
> autocannon -c 10 -w 10 -d 20 http://192.168.4.1
Running 20s test @ http://192.168.4.1
10 connections
10 workers
```

Stat	2.5%	50%	97.5%	99%	Avg	Stdev	Max
Latency	157 ms	5179 ms	10729 ms	11675 ms	5215.89 ms	3187.05 ms	12232 ms

  

Stat	1%	2.5%	50%	97.5%	Avg	Stdev	Min
Req/Sec	3	3	11	25	12.2	4.41	3
Bytes/Sec	13.1 kB	13.1 kB	48.2 kB	110 kB	53.4 kB	19.3 kB	13.1 kB

```
Req/Bytes counts sampled once per second.
# of samples: 200

488 requests in 20.02s, 1.07 MB read
```

With <https://github.com/mathieucarbou/AsyncTCPSock/archive/refs/tags/v1.0.3-dev.zip>:

```
[> autocannon -c 10 -w 10 -d 60 http://192.168.4.1
Running 60s test @ http://192.168.4.1
10 connections
10 workers
```

/

Stat	2.5%	50%	97.5%	99%	Avg	Stdev	Max
Latency	885 ms	16133 ms	29599 ms	30855 ms	14841.45 ms	8919.46 ms	31943 ms

Stat	1%	2.5%	50%	97.5%	Avg	Stdev	Min
Req/Sec	19	19	27	42	30.19	7.75	19
Bytes/Sec	83.2 kB	83.2 kB	118 kB	184 kB	132 kB	33.9 kB	83.2 kB

Req/Bytes counts sampled once per second.  
# of samples: 600

4k requests in 60.07s, 7.93 MB read

## SSE performance

In the example, there is an endpoint `/events` with some comments showing how these metrics are calculated.

Test is running for 20 seconds with 10 connections.

```
// With AsyncTCP, with 10 workers: no message discarded from the qu
//
```

```
// Total: 2038 events, 509.50 events / second
// Total: 2120 events, 530.00 events / second
// Total: 2119 events, 529.75 events / second
// Total: 2038 events, 509.50 events / second
// Total: 2037 events, 509.25 events / second
// Total: 2119 events, 529.75 events / second
// Total: 2119 events, 529.75 events / second
// Total: 2120 events, 530.00 events / second
// Total: 2038 events, 509.50 events / second
// Total: 2038 events, 509.50 events / second
//
```

```
// With AsyncTCPSock, with 10 workers: no message discarded from th
//
```

```
// Total: 2038 events, 509.50 events / second
// Total: 2120 events, 530.00 events / second
// Total: 2119 events, 529.75 events / second
// Total: 2038 events, 509.50 events / second
// Total: 2037 events, 509.25 events / second
// Total: 2119 events, 529.75 events / second
// Total: 2119 events, 529.75 events / second
// Total: 2120 events, 530.00 events / second
// Total: 2038 events, 509.50 events / second
```

// Total: 2038 events, 509.50 events / second

# Important recommendations for build options

---

Most of the crashes are caused by improper use or configuration of the AsyncTCP library used for the project. Here are some recommendations to avoid them and build-time flags you can change.

**CONFIG\_ASYNC\_TCP\_MAX\_ACK\_TIME** - defines a timeout for TCP connection to be considered alive when waiting for data. In some bad network conditions you might consider increasing it.

**CONFIG\_ASYNC\_TCP\_QUEUE\_SIZE** - defines the length of the queue for events related to connections handling. Both the server and AsyncTCP library in this fork were optimized to control the queue automatically. Do NOT try blindly increasing the queue size, it does not help you in a way you might think it is. If you receive debug messages about queue throttling, try to optimize your server callbacks code to execute as fast as possible. Read #165 thread, it might give you some hints.

**CONFIG\_ASYNC\_TCP\_RUNNING\_CORE** - CPU core thread affinity that runs the queue events handling and executes server callbacks. Default is ANY core, so it means that for dualcore SoCs both cores could handle server activities. If your server's code is too heavy and unoptimized or you see that sometimes server might affect other network activities, you might consider to bind it to the same core that runs Arduino code (1) to minimize affect on radio part. Otherwise you can leave the default to let RTOS decide where to run the thread based on priority

**CONFIG\_ASYNC\_TCP\_STACK\_SIZE** - stack size for the thread that runs sever events and callbacks. Default is 16k that is a way too much waste for well-defined short async code or simple static file handling. You might want to cosider reducing it to 4-8k to same RAM usage. If you do not know what this is or not sure about your callback code demands - leave it as default, should be enough even for very hungry

callbacks in most cases.

[!NOTE] This relates to ESP32 only, ESP8266 uses different ESPAsyncTCP lib that does not has this build options

I personally use the following configuration in my projects:

```
-D CONFIG_ASYNC_TCP_MAX_ACK_TIME=5000 // (keep default)
-D CONFIG_ASYNC_TCP_PRIORITY=10 // (keep default)
-D CONFIG_ASYNC_TCP_QUEUE_SIZE=64 // (keep default)
-D CONFIG_ASYNC_TCP_RUNNING_CORE=1 // force async_tcp task
-D CONFIG_ASYNC_TCP_STACK_SIZE=4096 // reduce the stack size
```

## AsyncWebSocketMessageBuffer and makeBuffer()

---

The fork from [yubox-node-org](#) introduces some breaking API changes compared to the original library, especially regarding the use of `std::shared_ptr<std::vector<uint8_t>>` for WebSocket.

This fork is compatible with the original library from [me-no-dev](#) regarding WebSocket, and wraps the optimizations done by [yubox-node-org](#) in the `AsyncWebSocketMessageBuffer` class. So you have the choice of which API to use.

Here are examples for serializing a json document in a websocket message buffer:

```
void send(JsonDocument& doc) {
    const size_t len = measureJson(doc);

    // original API from me-no-dev
    AsyncWebSocketMessageBuffer* buffer = _ws->makeBuffer(len);
    assert(buffer); // up to you to keep or remove this
    serializeJson(doc, buffer->get(), len);
    _ws->textAll(buffer);
}
```

```
void send(JsonDocument& doc) {
```



```

const size_t len = measureJson(doc);

// this fork (originally from yubox-node-org), uses another API
auto buffer = std::make_shared<std::vector<uint8_t>>>(len);
assert(buffer); // up to you to keep or remove this
serializeJson(doc, buffer->data(), len);
_ws->textAll(std::move(buffer));
}

```

I recommend to use the official API [AsyncWebSocketMessageBuffer](#) to retain further compatibility.

## How to replace a response

---

```

// It is possible to replace a response.
// The previous one will be deleted.
// Response sending happens when the handler returns.
server.on("/replace", HTTP_GET, [] (AsyncWebServerRequest* request) {
    request->send(200, "text/plain", "Hello, world");
    // ouch! finally we want to send a different response
    request->send(400, "text/plain", "validation error");
});

```

This will send error 400 instead of 200.

## How to use Middleware

---

Middleware is a way to intercept requests to perform some operations on them, like authentication, authorization, logging, etc and also act on the response headers.

Middleware can either be attached to individual handlers, attached at the server level (thus applied to all handlers), or both. They will be executed in the order they are attached, and they can stop the request processing by sending a response themselves.

You can have a look at the [SimpleServer.ino](#) example for some use cases.

For example, such middleware would handle authentication and set

some attributes on the request to make them available for the next middleware and for the handler which will process the request.

```
AsyncMiddlewareFunction complexAuth([](AsyncWebServerRequest* req)
    if (!request->authenticate("user", "password")) {
        return request->requestAuthentication();
    }

    request->setAttribute("user", "Mathieu");
    request->setAttribute("role", "staff");

    next(); // continue processing

    // you can act on the response object
    request->getResponse()->addHeader("X-Rate-Limit", "200");
});
```

Here are the list of available middlewares:

- **AsyncMiddlewareFunction**: can convert a lambda function (**ArMiddlewareCallback**) to a middleware
- **AsyncAuthenticationMiddleware**: to handle basic/digest authentication globally or per handler
- **AsyncAuthorizationMiddleware**: to handle authorization globally or per handler
- **AsyncCorsMiddleware**: to handle CORS preflight request globally or per handler
- **AsyncHeaderFilterMiddleware**: to filter out headers from the request
- **AsyncHeaderFreeMiddleware**: to only keep some headers from the request, and remove the others
- **LoggerMiddleware**: to log requests globally or per handler with the same pattern as curl. Will also record request processing time
- **AsyncRateLimitMiddleware**: to limit the number of requests on a windows of time globally or per handler

## How to use authentication with AsyncAuthenticationMiddleware

---

Do not use the `setUsername()` and `setPassword()` methods on the handlers anymore. They are deprecated. These methods were causing a copy of the username and password for each handler, which is not efficient.

Now, you can use the `AsyncAuthenticationMiddleware` to handle authentication globally or per handler.

```
AsyncAuthenticationMiddleware authMiddleware;

// [...]

authMiddleware.setAuthType(AsyncAuthType::AUTH_DIGEST);
authMiddleware.setRealm("My app name");
authMiddleware.setUsername("admin");
authMiddleware.setPassword("admin");
authMiddleware.setAuthFailureMessage("Authentication failed");
authMiddleware.generateHash(); // optimization to avoid generating

// [...]

server.addMiddleware(&authMiddleware); // globally add authentication

// [...]

myHandler.addMiddleware(&authMiddleware); // add authentication to handler
```

## Migration to Middleware to improve performance and memory usage

---

- `AsyncEventSource.authorizeConnect(...)` => do not use this method anymore: add a common `AsyncAuthorizationMiddleware` to the handler or server, and make sure to add it AFTER the `AsyncAuthenticationMiddleware` if you use authentication.
- `AsyncWebHandler.setAuthentication(...)` => do not use this method anymore: add a common `AsyncAuthenticationMiddleware` to the handler or server
- `ArUploadHandlerFunction` and `ArBodyHandlerFunction` => these callbacks receiving body data and upload and not calling anymore the authentication code for performance reasons.

These callbacks can be called multiple times during request parsing, so this is up to the user to now call the `AsyncAuthenticationMiddleware.allowed(request)` if needed and ideally when the method is called for the first time. These callbacks are also not triggering the whole middleware chain since they are not part of the request processing workflow (they are not the final handler).

## Maintainers

---

This fork of ESPAsyncWebServer and dependend libs are maintained as an opensource project at best effort level.

- [Mathieu Carbou](#)
- [Emil Muratov](#)

Thanks to all who contributed by providing PRs, testing and reporting issues.

## Original Documentation

---

- [Why should you care](#)
- [Important things to remember](#)
- [Principles of operation](#)
  - [The Async Web server](#)
  - [Request Life Cycle](#)
  - [Rewrites and how do they work](#)
  - [Handlers and how do they work](#)
  - [Responses and how do they work](#)
  - [Template processing](#)
- [Libraries and projects that use AsyncWebServer](#)
- [Request Variables](#)
  - [Common Variables](#)
  - [Headers](#)
  - [GET, POST and FILE parameters](#)
  - [FILE Upload handling](#)
  - [Body data handling](#)
  - [JSON body handling with ArduinoJson](#)
- [Responses](#)

- Redirect to another URL
- Basic response with HTTP Code
- Basic response with HTTP Code and extra headers
- Basic response with string content
- Basic response with string content and extra headers
- Send large webpage from PROGMEM
- Send large webpage from PROGMEM and extra headers
- Send large webpage from PROGMEM containing templates
- Send large webpage from PROGMEM containing templates and extra headers
- Send binary content from PROGMEM
- Respond with content coming from a Stream
- Respond with content coming from a Stream and extra headers
- Respond with content coming from a Stream containing templates
- Respond with content coming from a Stream containing templates and extra headers
- Respond with content coming from a File
- Respond with content coming from a File and extra headers
- Respond with content coming from a File containing templates
- Respond with content using a callback
- Respond with content using a callback and extra headers
- Respond with file content using a callback and extra headers
- Respond with content using a callback containing templates
- Respond with content using a callback containing templates and extra headers
- Chunked Response
- Chunked Response containing templates
- Print to response
- ArduinoJson Basic Response
- ArduinoJson Advanced Response
- Serving static files
  - Serving specific file by name
  - Serving files in directory
  - Serving static files with authentication
  - Specifying Cache-Control header

- Specifying Date-Modified header
  - Specifying Template Processor callback
  - Serving static files by custom handling
- Param Rewrite With Matching
- Using filters
  - Serve different site files in AP mode
  - Rewrite to different index on AP
  - Serving different hosts
  - Determine interface inside callbacks
- Bad Responses
  - Respond with content using a callback without content length to HTTP/1.0 clients
- Async WebSocket Plugin
  - Async WebSocket Event
  - Methods for sending data to a socket client
  - Direct access to web socket message buffer
  - Limiting the number of web socket clients
- Async Event Source Plugin
  - Setup Event Source on the server
  - Setup Event Source in the browser
- Scanning for available WiFi Networks
- Remove handlers and rewrites
- Setting up the server
  - Setup global and class functions as request handlers
  - Methods for controlling websocket connections
  - Adding Default Headers
  - Path variable

## Why should you care

---

- Using asynchronous network means that you can handle more than one connection at the same time
- You are called once the request is ready and parsed
- When you send the response, you are immediately ready to handle other connections while the server is taking care of sending the response in the background
- Speed is OMG
- Easy to use API, HTTP Basic and Digest MD5 Authentication

(default), ChunkedResponse

- Easily extendible to handle any type of content
- Supports Continue 100
- Async WebSocket plugin offering different locations without extra servers or ports
- Async EventSource (Server-Sent Events) plugin to send events to the browser
- URL Rewrite plugin for conditional and permanent url rewrites
- ServeStatic plugin that supports cache, Last-Modified, default index and more
- Simple template processing engine to handle templates

## Important things to remember

---

- This is fully asynchronous server and as such does not run on the loop thread.
- You can not use yield or delay or any function that uses them inside the callbacks
- The server is smart enough to know when to close the connection and free resources
- You can not send more than one response to a single request

## Principles of operation

---

### The Async Web server

- Listens for connections
- Wraps the new clients into **Request**
- Keeps track of clients and cleans memory
- Manages **Rewrites** and apply them on the request url
- Manages **Handlers** and attaches them to Requests

### Request Life Cycle

- TCP connection is received by the server
- The connection is wrapped inside **Request** object
- When the request head is received (type, url, get params, http version and host), the server goes through all **Rewrites** (in the order they were added) to rewrite the url and inject query

parameters, next, it goes through all attached **Handlers**(in the order they were added) trying to find one that **canHandle** the given request. If none are found, the default(catch-all) handler is attached.

- The rest of the request is received, calling the **handleUpload** or **handleBody** methods of the **Handler** if they are needed (POST+File/Body)
- When the whole request is parsed, the result is given to the **handleRequest** method of the **Handler** and is ready to be responded to
- In the **handleRequest** method, to the **Request** is attached a **Response** object (see below) that will serve the response data back to the client
- When the **Response** is sent, the client is closed and freed from the memory

## Rewrites and how do they work

- The **Rewrites** are used to rewrite the request url and/or inject get parameters for a specific request url path.
- All **Rewrites** are evaluated on the request in the order they have been added to the server.
- The **Rewrite** will change the request url only if the request url (excluding get parameters) is fully match the rewrite url, and when the optional **Filter** callback return true.
- Setting a **Filter** to the **Rewrite** enables to control when to apply the rewrite, decision can be based on request url, http version, request host/port/target host, get parameters or the request client's localIP or remoteIP.
- Two filter callbacks are provided: **ON\_AP\_FILTER** to execute the rewrite when request is made to the AP interface, **ON\_STA\_FILTER** to execute the rewrite when request is made to the STA interface.
- The **Rewrite** can specify a target url with optional get parameters, e.g. **/to-url?with=params**

## Handlers and how do they work

- The **Handlers** are used for executing specific actions to particular



requests

- One **Handler** instance can be attached to any request and lives together with the server
- Setting a **Filter** to the **Handler** enables to control when to apply the handler, decision can be based on request url, http version, request host/port/target host, get parameters or the request client's localIP or remoteIP.
- Two filter callbacks are provided: **ON\_AP\_FILTER** to execute the rewrite when request is made to the AP interface, **ON\_STA\_FILTER** to execute the rewrite when request is made to the STA interface.
- The **canHandle** method is used for handler specific control on whether the requests can be handled and for declaring any interesting headers that the **Request** should parse. Decision can be based on request method, request url, http version, request host/port/target host and get parameters
- Once a **Handler** is attached to given **Request** (**canHandle** returned true) that **Handler** takes care to receive any file/data upload and attach a **Response** once the **Request** has been fully parsed
- **Handlers** are evaluated in the order they are attached to the server. The **canHandle** is called only if the **Filter** that was set to the **Handler** return true.
- The first **Handler** that can handle the request is selected, not further **Filter** and **canHandle** are called.

## Responses and how do they work

- The **Response** objects are used to send the response data back to the client
- The **Response** object lives with the **Request** and is freed on end or disconnect
- Different techniques are used depending on the response type to send the data in packets returning back almost immediately and sending the next packet when this one is received. Any time in between is spent to run the user loop and handle other network packets
- Responding asynchronously is probably the most difficult thing for most to understand

- Many different options exist for the user to make responding a background task

## Template processing

- ESPAsyncWebserver contains simple template processing engine.
- Template processing can be added to most response types.
- Currently it supports only replacing template placeholders with actual values. No conditional processing, cycles, etc.
- Placeholders are delimited with % symbols. Like this:  
`%TEMPLATE_PLACEHOLDER%`.
- It works by extracting placeholder name from response text and passing it to user provided function which should return actual value to be used instead of placeholder.
- Since it's user provided function, it is possible for library users to implement conditional processing and cycles themselves.
- Since it's impossible to know the actual response size after template processing step in advance (and, therefore, to include it in response headers), the response becomes [chunked](#).

## Libraries and projects that use AsyncWebServer

---

- [WebSocketToSerial](#) - Debug serial devices through the web browser
- [Sattrack](#) - Track the ISS with ESP8266
- [ESP Radio](#) - Icecast radio based on ESP8266 and VS1053
- [VZero](#) - the Wireless zero-config controller for volkszaehler.org
- [ESPurna](#) - ESPurna ("spark" in Catalan) is a custom C firmware for ESP8266 based smart switches. It was originally developed with the ITEad Sonoff in mind.
- [fauxmoESP](#) - Belkin WeMo emulator library for ESP8266.
- [ESP-RFID](#) - MFRC522 RFID Access Control Management project for ESP8266.

## Request Variables

---

## Common Variables

```
request->version();           // uint8_t: 0 = HTTP/1.0, 1 = HTTP/1.1
request->method();            // enum:    HTTP_GET, HTTP_POST, HTTP_DELETE
request->url();               // String:  URL of the request (not including host)
request->host();              // String:  The requested host (can be IP or domain)
request->contentType();      // String:  ContentType of the request
request->contentLength();    // size_t:  ContentLength of the request
request->multipart();        // bool:    True if the request has content
```

## Headers

```
//List all collected headers
int headers = request->headers();
int i;
for(i=0;i<headers;i++){
    AsyncWebHeader* h = request->getHeader(i);
    Serial.printf("HEADER[%s]: %s\n", h->name().c_str(), h->value())
}
```

```
//get specific header by name
if(request->hasHeader("MyHeader")){
    AsyncWebHeader* h = request->getHeader("MyHeader");
    Serial.printf("MyHeader: %s\n", h->value().c_str());
}
```

```
//List all collected headers (Compatibility)
int headers = request->headers();
int i;
for(i=0;i<headers;i++){
    Serial.printf("HEADER[%s]: %s\n", request->headerName(i).c_str(),
}
```

```
//get specific header by name (Compatibility)
if(request->hasHeader("MyHeader")){
    Serial.printf("MyHeader: %s\n", request->header("MyHeader").c_str());
}
```

## GET, POST and FILE parameters

```

//List all parameters
int params = request->params();
for(int i=0;i<params;i++){
    AsyncWebParameter* p = request->getParam(i);
    if(p->isFile()){ //p->isPost() is also true
        Serial.printf("FILE[%s]: %s, size: %u\n", p->name().c_str(), |
    } else if(p->isPost()){
        Serial.printf("POST[%s]: %s\n", p->name().c_str(), p->value()
    } else {
        Serial.printf("GET[%s]: %s\n", p->name().c_str(), p->value().
    }
}

//Check if GET parameter exists
if(request->hasParam("download"))
    AsyncWebParameter* p = request->getParam("download");

//Check if POST (but not File) parameter exists
if(request->hasParam("download", true))
    AsyncWebParameter* p = request->getParam("download", true);

//Check if FILE was uploaded
if(request->hasParam("download", true, true))
    AsyncWebParameter* p = request->getParam("download", true, true

//List all parameters (Compatibility)
int args = request->args();
for(int i=0;i<args;i++){
    Serial.printf("ARG[%s]: %s\n", request->argName(i).c_str(), req
}

//Check if parameter exists (Compatibility)
if(request->hasArg("download"))
    String arg = request->arg("download");

```

## FILE Upload handling

```

void handleUpload(AsyncWebServerRequest *request, String filename,
    if(!index){
        Serial.printf("UploadStart: %s\n", filename.c_str());
    }
    for(size_t i=0; i<len; i++){

```

```

        Serial.write(data[i]);
    }
    if(final){
        Serial.printf("UploadEnd: %s, %u B\n", filename.c_str(), index);
    }
}

```

## Body data handling

```

void handleBody(AsyncWebServerRequest *request, uint8_t *data, size_t index, size_t len){
    if(!index){
        Serial.printf("BodyStart: %u B\n", total);
    }
    for(size_t i=0; i<len; i++){
        Serial.write(data[i]);
    }
    if(index + len == total){
        Serial.printf("BodyEnd: %u B\n", total);
    }
}

```

If needed, the `_tempObject` field on the request can be used to store a pointer to temporary data (e.g. from the body) associated with the request. If assigned, the pointer will automatically be freed along with the request.

## JSON body handling with ArduinoJson

Endpoints which consume JSON can use a special handler to get ready to use JSON data in the request callback:

```

#include "AsyncJson.h"
#include "ArduinoJson.h"

AsyncCallbackJsonWebHandler* handler = new AsyncCallbackJsonWebHandler(
    "/json", server, JsonBuffer<JsonObject>());
// ...
});
server.addHandler(handler);

```

# Responses

---

## Redirect to another URL

```
//to local url
request->redirect("/login");

//to external url
request->redirect("http://esp8266.com");
```

## Basic response with HTTP Code

```
request->send(404); //Sends 404 File Not Found
```

## Basic response with HTTP Code and extra headers

```
AsyncWebServerResponse *response = request->beginResponse(404); /
response->addHeader("Server", "ESP Async Web Server");
request->send(response);
```

## Basic response with string content

```
request->send(200, "text/plain", "Hello World!");
```

## Basic response with string content and extra headers

```
AsyncWebServerResponse *response = request->beginResponse(200, "t
response->addHeader("Server", "ESP Async Web Server");
request->send(response);
```

## Send large webpage from PROGMEM

```
const char index_html[] PROGMEM = "..."; // large char array, tes
request->send_P(200, "text/html", index_html);
```

## Send large webpage from PROGMEM and extra headers

```
const char index_html[] PROGMEM = "..."; // large char array, test
AsyncWebServerResponse *response = request->beginResponse_P(200,
response->addHeader("Server", "ESP Async Web Server");
request->send(response);
```

## Send large webpage from PROGMEM containing templates

```
String processor(const String& var)
{
    if(var == "HELLO_FROM_TEMPLATE")
        return F("Hello world!");
    return String();
}

// ...

const char index_html[] PROGMEM = "..."; // large char array, test
request->send_P(200, "text/html", index_html, processor);
```

## Send large webpage from PROGMEM containing templates and extra headers

```
String processor(const String& var)
{
    if(var == "HELLO_FROM_TEMPLATE")
        return F("Hello world!");
    return String();
}

// ...

const char index_html[] PROGMEM = "..."; // large char array, test
AsyncWebServerResponse *response = request->beginResponse_P(200,
response->addHeader("Server", "ESP Async Web Server");
request->send(response);
```

## Send binary content from PROGMEM

```
//File: favicon.ico.gz, Size: 726
#define favicon_ico_gz_len 726
const uint8_t favicon_ico_gz[] PROGMEM = {
    0x1F, 0x8B, 0x08, 0x08, 0x0B, 0x87, 0x90, 0x57, 0x00, 0x03, 0x66
    0x6E, 0x2E, 0x69, 0x63, 0x6F, 0x00, 0xCD, 0x53, 0x5F, 0x48, 0x9A
    0x86, 0x96, 0xA9, 0x64, 0xD3, 0xFE, 0xA8, 0x99, 0x65, 0x1A, 0xB4
    0xA8, 0x11, 0x49, 0x51, 0x8A, 0x34, 0x62, 0x93, 0x85, 0x31, 0x58
    0xF5, 0x52, 0x41, 0x10, 0x23, 0x82, 0xA0, 0x20, 0x98, 0x2F, 0xC1
    0x04, 0xD7, 0x83, 0x58, 0x20, 0x28, 0x04, 0xAB, 0xD1, 0x9B, 0x8C
    0x0E, 0x56, 0xBF, 0x9D, 0xEF, 0xF6, 0x30, 0x82, 0xED, 0xAD, 0x87
    0x73, 0xCF, 0xEF, 0x9C, 0xDF, 0x39, 0xBF, 0xFB, 0x31, 0x26, 0xA2
    0xCF, 0x9E, 0x67, 0x30, 0xA6, 0x66, 0x8C, 0x99, 0xC9, 0xC8, 0x45
    0xA6, 0x94, 0x5E, 0xDB, 0xFF, 0xB2, 0xE6, 0xE7, 0xE7, 0xF9, 0xDE
    0xD8, 0x78, 0xBF, 0xA1, 0xA1, 0xC1, 0xDA, 0xDC, 0xDC, 0x2C, 0xEB
    0xE6, 0x4A, 0x83, 0xC1, 0xE0, 0x2E, 0x29, 0x29, 0x99, 0xD6, 0x6A
    0x61, 0x75, 0x75, 0x95, 0xB5, 0xB7, 0xB7, 0xDF, 0xC8, 0xD1, 0xD4
    0xFA, 0x83, 0xD5, 0x6A, 0xFD, 0x5A, 0x5E, 0x5E, 0x9E, 0x28, 0x2D
    0x98, 0x78, 0x5E, 0x5E, 0xDE, 0x95, 0x42, 0xA1, 0x40, 0x4E, 0x4E
    0x47, 0xB5, 0x5A, 0x6D, 0x4F, 0x26, 0x93, 0xA2, 0xD6, 0xD6, 0x56
    0x11, 0x36, 0x62, 0xB1, 0x58, 0x60, 0x32, 0x99, 0xA0, 0xD7, 0xEB
    0xFC, 0x7C, 0x10, 0x16, 0x02, 0x56, 0x2E, 0x97, 0x43, 0x2A, 0x95
    0x33, 0x33, 0xAE, 0x52, 0xA9, 0x1E, 0x64, 0x65, 0x65, 0x71, 0x7C
    0xFE, 0x30, 0x1A, 0x8D, 0xE8, 0xEC, 0xEC, 0xC4, 0xE2, 0xE2, 0x22
    0x41, 0xB5, 0x38, 0x4E, 0xC8, 0x33, 0x3C, 0x3C, 0x0C, 0x87, 0xC3
    0xBC, 0xE9, 0xEB, 0xEB, 0x93, 0x5F, 0x5C, 0x5C, 0x30, 0x8A, 0x9D
    0x3E, 0x41, 0xBD, 0x21, 0x1E, 0x8F, 0x63, 0x6A, 0x6A, 0x0A, 0x81
    0x3D, 0x3D, 0x42, 0x3C, 0x96, 0x96, 0x96, 0x70, 0x7E, 0x7E, 0x8E
    0xFE, 0xB4, 0xD7, 0xEB, 0xF5, 0x8F, 0x8F, 0x8F, 0x5B, 0x68, 0x5E
    0xE8, 0xE8, 0x08, 0x27, 0x27, 0x27, 0xD8, 0xDF, 0xDF, 0xC7, 0xD9
    0x36, 0x36, 0x36, 0x38, 0x9F, 0x85, 0x85, 0x05, 0xAC, 0xAF, 0xAF
    0x48, 0x20, 0x16, 0x8B, 0xFD, 0xDA, 0xDA, 0xDA, 0x7A, 0x41, 0x33
    0x89, 0x89, 0x09, 0x84, 0xC3, 0x61, 0x6C, 0x6F, 0x6F, 0x23, 0x12
    0x43, 0x43, 0x43, 0x58, 0x5E, 0x5E, 0xE6, 0x9C, 0x7D, 0x3E, 0x1F
    0xBD, 0xBD, 0x3D, 0xE1, 0x3C, 0x1D, 0x0C, 0x06, 0x9F, 0x10, 0xB7
    0xF7, 0x63, 0x60, 0x60, 0x00, 0x83, 0x83, 0x83, 0x18, 0x19, 0x19
    0x17, 0x7C, 0xA4, 0x27, 0xE7, 0x34, 0x39, 0x39, 0x89, 0x9D, 0x9D
    0xE5, 0x34, 0x11, 0x37, 0x49, 0x51, 0x51, 0xD1, 0x4B, 0xA5, 0x52
    0x0A, 0xF3, 0x92, 0x48, 0x24, 0xA0, 0x6F, 0x14, 0x17, 0x17, 0xA3
    0x5D, 0x7C, 0x1E, 0xBB, 0xBB, 0xBB, 0x9C, 0xD7, 0xE1, 0xE1, 0x21
    0x45, 0x4C, 0x33, 0x12, 0x34, 0xCC, 0xA0, 0x19, 0x54, 0x92, 0x56
}
```



```

0xCF, 0x82, 0x56, 0xC2, 0xDC, 0xEB, 0xEA, 0xEA, 0x38, 0x7E, 0x6C
0xB8, 0xBF, 0xA7, 0xFA, 0xAF, 0x56, 0x56, 0x56, 0xEE, 0x6D, 0x6E
0x55, 0x55, 0x6C, 0x66, 0x66, 0x46, 0x44, 0xDA, 0x3B, 0x34, 0x1A
0x7B, 0x45, 0xBD, 0xA5, 0x5D, 0x2E, 0x57, 0x8C, 0x7A, 0x73, 0xD9
0xE7, 0x7D, 0xA6, 0x3A, 0x2C, 0x95, 0x4A, 0xB1, 0x8E, 0x8E, 0x0E
0xE9, 0x74, 0x3E, 0x73, 0xBB, 0xDD, 0x8F, 0x3C, 0x1E, 0x8F, 0xE6
0xAD, 0xDD, 0xDE, 0xCF, 0x73, 0x0B, 0x0B, 0xB8, 0xB6, 0xE0, 0x5D
0x4C, 0xF4, 0xF7, 0xD8, 0x59, 0xF2, 0x7F, 0xA3, 0xB8, 0xB4, 0xFC
0x16, 0x4A, 0x7E, 0x04, 0x00, 0x00
};

```

```

AsyncWebServerResponse *response = request->beginResponse_P(200,
response->addHeader("Content-Encoding", "gzip");
request->send(response);

```

## Respond with content coming from a Stream

```

//read 12 bytes from Serial and send them as Content Type text/pli
request->send(Serial, "text/plain", 12);

```

## Respond with content coming from a Stream and extra headers

```

//read 12 bytes from Serial and send them as Content Type text/pli
AsyncWebServerResponse *response = request->beginResponse(Serial,
response->addHeader("Server", "ESP Async Web Server");
request->send(response);

```

## Respond with content coming from a Stream containing templates

```
String processor(const String& var)
{
    if(var == "HELLO_FROM_TEMPLATE")
        return F("Hello world!");
    return String();
}

// ...

//read 12 bytes from Serial and send them as Content Type text/pla
request->send(Serial, "text/plain", 12, processor);
```

## Respond with content coming from a Stream containing templates and extra headers

```
String processor(const String& var)
{
    if(var == "HELLO_FROM_TEMPLATE")
        return F("Hello world!");
    return String();
}

// ...

//read 12 bytes from Serial and send them as Content Type text/pla
AsyncWebServerResponse *response = request->beginResponse(Serial,
response->addHeader("Server","ESP Async Web Server");
request->send(response);
```

## Respond with content coming from a File

```
//Send index.htm with default content type
request->send(SPIFFS, "/index.htm");

//Send index.htm as text
request->send(SPIFFS, "/index.htm", "text/plain");

//Download index.htm
request->send(SPIFFS, "/index.htm", String(), true);
```

## Respond with content coming from a File and extra headers

```
//Send index.htm with default content type
AsyncWebServerResponse *response = request->beginResponse(SPIFFS,

//Send index.htm as text
AsyncWebServerResponse *response = request->beginResponse(SPIFFS,

//Download index.htm
AsyncWebServerResponse *response = request->beginResponse(SPIFFS,

response->addHeader("Server", "ESP Async Web Server");
request->send(response);
```

## Respond with content coming from a File containing templates

Internally uses [Chunked Response](#).

Index.htm contents:

```
%HELLO_FROM_TEMPLATE%
```

Somewhere in source files:

```
String processor(const String& var)
{
    if(var == "HELLO_FROM_TEMPLATE")
        return F("Hello world!");
    return String();
}

// ...

//Send index.htm with template processor function
request->send(SPIFFS, "/index.htm", String(), false, processor);
```

## Respond with content using a callback

```

//send 128 bytes as plain text
request->send("text/plain", 128, [](uint8_t *buffer, size_t maxLen)
    //Write up to "maxLen" bytes into "buffer" and return the amount
    //index equals the amount of bytes that have been already sent
    //You will not be asked for more bytes once the content length is
    //Keep in mind that you can not delay or yield waiting for more
    //Send what you currently have and you will be asked for more again
    return mySource.read(buffer, maxLen);
});

```

## Respond with content using a callback and extra headers

```

//send 128 bytes as plain text
AsyncWebServerResponse *response = request->beginResponse("text/plain", 128,
    //Write up to "maxLen" bytes into "buffer" and return the amount
    //index equals the amount of bytes that have been already sent
    //You will not be asked for more bytes once the content length is
    //Keep in mind that you can not delay or yield waiting for more
    //Send what you currently have and you will be asked for more again
    return mySource.read(buffer, maxLen);
});
response->addHeader("Server", "ESP Async Web Server");
request->send(response);

```

## Respond with file content using a callback and extra headers

With this code your ESP is able to serve even large (large in terms of ESP, e.g. 100kB) files without memory problems.

You need to create a file handler in outer function (to have a single one for request) but use it in a lambda. The catch is that the lambda has it's own lifecycle which may/will cause it's called after the original function is over thus the original file handle is destroyed. Using the captured `&file` in the lambda then causes segfault (Hello, Exception 9!) and the whole ESP crashes.

By using this code, you tell the compiler to move the handle into the lambda so it won't be destroyed when outer function (that one

where you call `request->send(response)` ends.

```
const File file = ... // e.g. SPIFFS.open(path, "r");

const contentType = "application/javascript";

AsyncWebServerResponse *response = request->beginResponse(
    contentType,
    file.size(),
    [file](uint8_t *buffer, size_t maxLen, size_t total) mutable ->
        int bytes = file.read(buffer, maxLen);

    // close file at the end
    if (bytes + total == file.size()) file.close();

    return max(0, bytes); // return 0 even when no bytes were lo
};

if (gzipped) {
    response->addHeader(F("Content-Encoding"), F("gzip"));
}

request->send(response);
```

## Respond with content using a callback containing templates

```
String processor(const String& var)
{
    if(var == "HELLO_FROM_TEMPLATE")
        return F("Hello world!");
    return String();
}

// ...

//send 128 bytes as plain text
request->send("text/plain", 128, [](uint8_t *buffer, size_t maxLen)
    //Write up to "maxLen" bytes into "buffer" and return the amount
    //index equals the amount of bytes that have been already sent
    //You will not be asked for more bytes once the content length l
```

```

    //Keep in mind that you can not delay or yield waiting for more
    //Send what you currently have and you will be asked for more a
    return mySource.read(buffer, maxLen);
}, processor);

```

## Respond with content using a callback containing templates and extra headers

```

String processor(const String& var)
{
    if(var == "HELLO_FROM_TEMPLATE")
        return F("Hello world!");
    return String();
}

// ...

//send 128 bytes as plain text
AsyncWebServerResponse *response = request->beginResponse("text/p
    //Write up to "maxLen" bytes into "buffer" and return the amount
    //index equals the amount of bytes that have been already sent
    //You will not be asked for more bytes once the content length l
    //Keep in mind that you can not delay or yield waiting for more
    //Send what you currently have and you will be asked for more a
    return mySource.read(buffer, maxLen);
}, processor);
response->addHeader("Server", "ESP Async Web Server");
request->send(response);

```

## Chunked Response

Used when content length is unknown. Works best if the client supports HTTP/1.1

```

AsyncWebServerResponse *response = request->beginChunkedResponse(
    //Write up to "maxLen" bytes into "buffer" and return the amount
    //index equals the amount of bytes that have been already sent
    //You will be asked for more data until 0 is returned
    //Keep in mind that you can not delay or yield waiting for more
    return mySource.read(buffer, maxLen);
});

```

```
response->addHeader("Server", "ESP Async Web Server");
request->send(response);
```

## Chunked Response containing templates

Used when content length is unknown. Works best if the client supports HTTP/1.1

```
String processor(const String& var)
{
    if(var == "HELLO_FROM_TEMPLATE")
        return F("Hello world!");
    return String();
}

// ...

AsyncWebServerResponse *response = request->beginChunkedResponse(
    //Write up to "maxLen" bytes into "buffer" and return the amount
    //index equals the amount of bytes that have been already sent
    //You will be asked for more data until 0 is returned
    //Keep in mind that you can not delay or yield waiting for more
    return mySource.read(buffer, maxLen);
}, processor);
response->addHeader("Server", "ESP Async Web Server");
request->send(response);
```

## Print to response

```
AsyncResponseStream *response = request->beginResponseStream("text/html");
response->addHeader("Server", "ESP Async Web Server");
response->printf("<!DOCTYPE html><html><head><title>Webpage at %s", request->url());

response->print("<h2>Hello ");
response->print(request->client()->remoteIP());
response->print("</h2>");

response->print("<h3>General</h3>");
response->print("<ul>");
response->printf("<li>Version: HTTP/1.%u</li>", request->version());
response->printf("<li>Method: %s</li>", request->methodToString());
```

```

response->printf("<li>URL: %s</li>", request->url().c_str());
response->printf("<li>Host: %s</li>", request->host().c_str());
response->printf("<li>ContentType: %s</li>", request->contentType
response->printf("<li>ContentLength: %u</li>", request->contentLe
response->printf("<li>Multipart: %s</li>", request->multipart())?""
response->print("</ul>");

response->print("<h3>Headers</h3>");
response->print("<ul>");
int headers = request->headers();
for(int i=0;i<headers;i++){
    AsyncWebHeader* h = request->getHeader(i);
    response->printf("<li>%s: %s</li>", h->name().c_str(), h->value
}
response->print("</ul>");

response->print("<h3>Parameters</h3>");
response->print("<ul>");
int params = request->params();
for(int i=0;i<params;i++){
    AsyncWebParameter* p = request->getParam(i);
    if(p->isFile()){
        response->printf("<li>FILE[%s]: %s, size: %u</li>", p->name()
    } else if(p->isPost()){
        response->printf("<li>POST[%s]: %s</li>", p->name().c_str(), |
    } else {
        response->printf("<li>GET[%s]: %s</li>", p->name().c_str(), p
    }
}
response->print("</ul>");

response->print("</body></html>");
//send the response last
request->send(response);

```

## ArduinoJson Basic Response

This way of sending Json is great for when the result is below 4KB

```

#include "AsyncJson.h"
#include "ArduinoJson.h"

```



```

AsyncResponseStream *response = request->beginResponseStream("app:
DynamicJsonBuffer jsonBuffer;
JsonObject &root = jsonBuffer.createObject();
root["heap"] = ESP.getFreeHeap();
root["ssid"] = WiFi.SSID();
root.printTo(*response);
request->send(response);

```

## ArduinoJson Advanced Response

This response can handle really large json objects (tested to 40KB)  
 There isn't any noticeable speed decrease for small results with the method above. Since ArduinoJson does not allow reading parts of the string, the whole json has to be passed every time a chunk needs to be sent, which shows speed decrease proportional to the resulting json packets.

```

#include "AsyncJson.h"
#include "ArduinoJson.h"

AsyncJsonResponse * response = new AsyncJsonResponse();
response->addHeader("Server", "ESP Async Web Server");
JsonObject& root = response->getRoot();
root["heap"] = ESP.getFreeHeap();
root["ssid"] = WiFi.SSID();
response->setLength();
request->send(response);

```

## Serving static files

---

In addition to serving files from SPIFFS as described above, the server provides a dedicated handler that optimizes the performance of serving files from SPIFFS - `AsyncStaticWebHandler`. Use `server.serveStatic()` function to initialize and add a new instance of `AsyncStaticWebHandler` to the server. The handler will not handle the request if the file does not exist, e.g. the server will continue to look for another handler that can handle the request. Notice that you can chain setter functions to setup the handler, or keep a pointer to change it at a later time.

## Serving specific file by name

```
// Serve the file "/www/page.htm" when request url is "/page.htm"  
server.serveStatic("/page.htm", SPIFFS, "/www/page.htm");
```

## Serving files in directory

To serve files in a directory, the path to the files should specify a directory in SPIFFS and ends with "/".

```
// Serve files in directory "/www/" when request url starts with '/'  
// Request to the root or none existing files will try to server i  
// file name "index.htm" if exists  
server.serveStatic("/", SPIFFS, "/www/");
```

```
// Server with different default file  
server.serveStatic("/", SPIFFS, "/www/").setDefaultFile("default.l
```

## Serving static files with authentication

```
server  
  .serveStatic("/", SPIFFS, "/www/")  
  .setDefaultFile("default.html")  
  .setAuthentication("user", "pass");
```

## Specifying Cache-Control header

It is possible to specify Cache-Control header value to reduce the number of calls to the server once the client loaded the files. For more information on Cache-Control values see [Cache-Control](#)

```
// Cache responses for 10 minutes (600 seconds)
server.serveStatic("/", SPIFFS, "/www/").setCacheControl("max-age=600");

/** Change Cache-Control after server setup */

// During setup - keep a pointer to the handler
AsyncStaticWebHandler* handler = &server.serveStatic("/", SPIFFS, "/www/");

// At a later event - change Cache-Control
handler->setCacheControl("max-age=30");
```

## Specifying Date-Modified header

It is possible to specify Date-Modified header to enable the server to return Not-Modified (304) response for requests with "If-Modified-Since" header with the same value, instead of responding with the actual file content.

```
// Update the date modified string every time files are updated
server.serveStatic("/", SPIFFS, "/www/").setLastModified("Mon, 20 Aug 2018 12:00:00 GMT");

/** Change last modified value at a later stage */

// During setup - read last modified value from config or EEPROM
String date_modified = loadDateModified();
AsyncStaticWebHandler* handler = &server.serveStatic("/", SPIFFS, "/www/");
handler->setLastModified(date_modified);

// At a later event when files are updated
String date_modified = getNewDateModified();
saveDateModified(date_modified); // Save for next reset
handler->setLastModified(date_modified);
```

## Specifying Template Processor callback

It is possible to specify template processor for static files. For information on template processor see [Respond with content coming from a File containing templates](#).

```
String processor(const String& var)
```

```

{
    if(var == "HELLO_FROM_TEMPLATE")
        return F("Hello world!");
    return String();
}

// ...

server.ServeStatic("/", SPIFFS, "/www/").setTemplateProcessor(proc)

```

## Serving static files by custom handling

It may happen your static files are too big and the ESP will crash the request before it sends the whole file.

In that case, you can handle static files with custom file serving through not found handler.

This code below is more-or-less equivalent to this:

```

webServer.ServeStatic("/", SPIFFS, STATIC_FILES_PREFIX).setDefaultHandler(

```

First, declare the handling function:

```

bool handleStaticFile(AsyncWebServerRequest *request) {
    String path = STATIC_FILES_PREFIX + request->url();

    if (path.endsWith("/")) path += F("index.html");

    String contentType = getContentType(path);
    String pathWithGz = path + ".gz";

    if (SPIFFS.exists(pathWithGz) || SPIFFS.exists(path)) {
        bool gzipped = false;
        if (SPIFFS.exists(pathWithGz)) {
            gzipped = true;
            path += ".gz";
        }

        // TODO serve the file

        return true;
    }
}

```

```

    }

    return false;
}

```

And then configure your webserver:

```

webServer.onNotFound([](AsyncWebServerRequest *request) {
    if (handleStaticFile(request)) return;

    request->send(404);
});

```

You may want to try [Respond with file content using a callback and extra headers](#) For actual serving the file.

## Param Rewrite With Matching

---

It is possible to rewrite the request url with parameter match. Here is an example with one parameter: Rewrite for example "/radio/{frequency}" -> "/radio?f={frequency}"

```

class OneParamRewrite : public AsyncWebRewrite
{
protected:
    String _urlPrefix;
    int _paramIndex;
    String _paramsBackup;

public:
    OneParamRewrite(const char* from, const char* to)
        : AsyncWebRewrite(from, to) {

        _paramIndex = _from.indexOf('{');

        if( _paramIndex >=0 && _from.endsWith("}") ) {
            _urlPrefix = _from.substring(0, _paramIndex);
            int index = _params.indexOf('{');
            if(index >= 0) {
                _params = _params.substring(0, index);
            }
        }
    }
}

```

```

        } else {
            _urlPrefix = _from;
        }
        _paramsBackup = _params;
    }

    bool match(AsyncWebServerRequest *request) override {
        if(request->url().startsWith(_urlPrefix)) {
            if(_paramIndex >= 0) {
                _params = _paramsBackup + request->url().substring(_paramIndex);
            } else {
                _params = _paramsBackup;
            }
            return true;
        } else {
            return false;
        }
    }
};

```

Usage:

```
server.addRewrite( new OneParamRewrite("/radio/{frequency}", "/radio/{frequency}"));
```

## Using filters

---

Filters can be set to **Rewrite** or **Handler** in order to control when to apply the rewrite and consider the handler. A filter is a callback function that evaluates the request and return a boolean **true** to include the item or **false** to exclude it. Two filter callback are provided for convince:

- **ON\_STA\_FILTER** - return true when requests are made to the STA (station mode) interface.
- **ON\_AP\_FILTER** - return true when requests are made to the AP (access point) interface.

## Serve different site files in AP mode

```
server.serveStatic("/", SPIFFS, "/www/").setFilter(ON_STA_FILTER);
server.serveStatic("/", SPIFFS, "/ap/").setFilter(ON_AP_FILTER);
```

## Rewrite to different index on AP

```
// Serve the file "/www/index-ap.htm" in AP, and the file "/www/index.htm" in STA
server.rewrite("/", "index.htm");
server.rewrite("/index.htm", "index-ap.htm").setFilter(ON_AP_FILTER);
server.serveStatic("/", SPIFFS, "/www/");
```

## Serving different hosts

```
// Filter callback using request host
bool filterOnHost1(AsyncWebServerRequest *request) { return request->host() == "host1"; }

// Server setup: server files in "/host1/" to requests for "host1"
server.serveStatic("/", SPIFFS, "/host1/").setFilter(filterOnHost1);
server.serveStatic("/", SPIFFS, "/www/");
```

## Determine interface inside callbacks

```
String RedirectUrl = "http://";
if (ON_STA_FILTER(request)) {
    RedirectUrl += WiFi.localIP().toString();
} else {
    RedirectUrl += WiFi.softAPIP().toString();
}
RedirectUrl += "/index.htm";
request->redirect(RedirectUrl);
```

## Bad Responses

---

Some responses are implemented, but you should not use them, because they do not conform to HTTP. The following example will lead to unclean close of the connection and more time wasted than providing the length of the content

## Respond with content using a callback without

## content length to HTTP/1.0 clients

```
//This is used as fallback for chunked responses to HTTP/1.0 Client
request->send("text/plain", 0, [(uint8_t *buffer, size_t maxlen,
//Write up to "maxLen" bytes into "buffer" and return the amount
//You will be asked for more data until 0 is returned
//Keep in mind that you can not delay or yield waiting for more
return mySource.read(buffer, maxlen);
});
```

## Async WebSocket Plugin

---

The server includes a web socket plugin which lets you define different WebSocket locations to connect to without starting another listening service or using different port

### Async WebSocket Event

```
void onEvent(AsyncWebSocket * server, AsyncWebSocketClient * client,
if(type == WS_EVT_CONNECT){
    //client connected
    os_printf("ws[%s][%u] connect\n", server->url(), client->id());
    client->printf("Hello Client %u :)", client->id());
    client->ping();
} else if(type == WS_EVT_DISCONNECT){
    //client disconnected
    os_printf("ws[%s][%u] disconnect: %u\n", server->url(), client->id(), client->id());
} else if(type == WS_EVT_ERROR){
    //error was received from the other end
    os_printf("ws[%s][%u] error(%u): %s\n", server->url(), client->id(), client->id(), client->error());
} else if(type == WS_EVT_PONG){
    //pong message was received (in response to a ping request may be)
    os_printf("ws[%s][%u] pong[%u]: %s\n", server->url(), client->id(), client->id(), client->pong());
} else if(type == WS_EVT_DATA){
    //data packet
    AwsFrameInfo * info = (AwsFrameInfo*)arg;
    if(info->final && info->index == 0 && info->len == len){
        //the whole message is in a single frame and we got all of it
        os_printf("ws[%s][%u] %s-message[%llu]: ", server->url(), client->id(), client->id(), client->len());
        if(info->opcode == WS_TEXT){
            client->printf("%s", client->data());
        } else if(info->opcode == WS_BINARY){
            client->printf("%u bytes of binary data", client->len());
        }
    }
}
```



```

        data[len] = 0;
        os_printf("%s\n", (char*)data);
    } else {
        for(size_t i=0; i < info->len; i++){
            os_printf("%02x ", data[i]);
        }
        os_printf("\n");
    }
    if(info->opcode == WS_TEXT)
        client->text("I got your text message");
    else
        client->binary("I got your binary message");
} else {
    //message is comprised of multiple frames or the frame is split
    if(info->index == 0){
        if(info->num == 0)
            os_printf("ws[%s][%u] %s-message start\n", server->url(),
            os_printf("ws[%s][%u] frame[%u] start[%llu]\n", server->url(),
        }

        os_printf("ws[%s][%u] frame[%u] %s[%llu - %llu]: ", server->url(),
        if(info->message_opcode == WS_TEXT){
            data[len] = 0;
            os_printf("%s\n", (char*)data);
        } else {
            for(size_t i=0; i < len; i++){
                os_printf("%02x ", data[i]);
            }
            os_printf("\n");
        }
    }

    if((info->index + len) == info->len){
        os_printf("ws[%s][%u] frame[%u] end[%llu]\n", server->url(),
        if(info->final){
            os_printf("ws[%s][%u] %s-message end\n", server->url(),
            if(info->message_opcode == WS_TEXT)
                client->text("I got your text message");
            else
                client->binary("I got your binary message");
        }
    }
}
}
}
}

```

# Methods for sending data to a socket client

```
//Server methods
AsyncWebSocket ws("/ws");
//printf to a client
ws.printf((uint32_t)client_id, arguments...);
//printf to all clients
ws.printfAll(arguments...);
//printf_P to a client
ws.printf_P((uint32_t)client_id, PSTR(format), arguments...);
//printfAll_P to all clients
ws.printfAll_P(PSTR(format), arguments...);
//send text to a client
ws.text((uint32_t)client_id, (char*)text);
ws.text((uint32_t)client_id, (uint8_t*)text, (size_t)len);
//send text from PROGMEM to a client
ws.text((uint32_t)client_id, PSTR("text"));
const char flash_text[] PROGMEM = "Text to send"
ws.text((uint32_t)client_id, FPSTR(flash_text));
//send text to all clients
ws.textAll((char*)text);
ws.textAll((uint8_t*)text, (size_t)len);
//send binary to a client
ws.binary((uint32_t)client_id, (char*)binary);
ws.binary((uint32_t)client_id, (uint8_t*)binary, (size_t)len);
//send binary from PROGMEM to a client
const uint8_t flash_binary[] PROGMEM = { 0x01, 0x02, 0x03, 0x04 }
ws.binary((uint32_t)client_id, flash_binary, 4);
//send binary to all clients
ws.binaryAll((char*)binary);
ws.binaryAll((uint8_t*)binary, (size_t)len);
//HTTP Authenticate before switch to Websocket protocol
ws.setAuthentication("user", "pass");

//client methods
AsyncWebSocketClient * client;
//printf
client->printf(arguments...);
//printf_P
client->printf_P(PSTR(format), arguments...);
```

```

//send text
client->text((char*)text);
client->text((uint8_t*)text, (size_t)len);
//send text from PROGMEM
client->text(PSTR("text"));
const char flash_text[] PROGMEM = "Text to send";
client->text(FPSTR(flash_text));
//send binary
client->binary((char*)binary);
client->binary((uint8_t*)binary, (size_t)len);
//send binary from PROGMEM
const uint8_t flash_binary[] PROGMEM = { 0x01, 0x02, 0x03, 0x04 }
client->binary(flash_binary, 4);

```

## Direct access to web socket message buffer

When sending a web socket message using the above methods a buffer is created. Under certain circumstances you might want to manipulate or populate this buffer directly from your application, for example to prevent unnecessary duplications of the data. This example below shows how to create a buffer and print data to it from an ArduinoJson object then send it.

```

void sendDataWs(AsyncWebSocketClient * client)
{
    DynamicJsonBuffer jsonBuffer;
    JsonObject& root = jsonBuffer.createObject();
    root["a"] = "abc";
    root["b"] = "abcd";
    root["c"] = "abcde";
    root["d"] = "abcdef";
    root["e"] = "abcdefg";
    size_t len = root.measureLength();
    AsyncWebSocketMessageBuffer * buffer = ws.makeBuffer(len); //
    if (buffer) {
        root.printTo((char *)buffer->get(), len + 1);
        if (client) {
            client->text(buffer);
        } else {
            ws.textAll(buffer);
        }
    }
}

```

```
}
```

## Limiting the number of web socket clients

Browsers sometimes do not correctly close the websocket connection, even when the `close()` function is called in javascript. This will eventually exhaust the web server's resources and will cause the server to crash. Periodically calling the `cleanClients()` function from the main `loop()` function limits the number of clients by closing the oldest client when the maximum number of clients has been exceeded. This can be called every cycle, however, if you wish to use less power, then calling as infrequently as once per second is sufficient.

```
void loop(){
  ws.cleanupClients();
}
```

## Async Event Source Plugin

---

The server includes EventSource (Server-Sent Events) plugin which can be used to send short text events to the browser. Difference between EventSource and WebSockets is that EventSource is single direction, text-only protocol.

## Setup Event Source on the server

```
AsyncWebServer server(80);
AsyncEventSource events("/events");

void setup(){
  // setup .....
  events.onConnect([](AsyncEventSourceClient *client){
    if(client->lastId()){
      Serial.printf("Client reconnected! Last message ID that it ");
    }
    //send event with message "hello!", id current millis
    // and set reconnect delay to 1 second
    client->send("hello!", NULL, millis(), 1000);
  });
}
```

```

});
//HTTP Basic authentication
events.setAuthentication("user", "pass");
server.addHandler(&events);
// setup .....
}

void loop(){
  if(eventTriggered){ // your logic here
    //send event "myevent"
    events.send("my event content","myevent",millis());
  }
}

```

## Setup Event Source in the browser

```

if (!!window.EventSource) {
  var source = new EventSource("/events");

  source.addEventListener(
    "open",
    function (e) {
      console.log("Events Connected");
    },
    false
  );

  source.addEventListener(
    "error",
    function (e) {
      if (e.target.readyState != EventSource.OPEN) {
        console.log("Events Disconnected");
      }
    },
    false
  );

  source.addEventListener(
    "message",
    function (e) {
      console.log("message", e.data);
    },
    false
  );
}

```

```

);

source.addEventListener(
    "myevent",
    function (e) {
        console.log("myevent", e.data);
    },
    false
);
}

```

## Scanning for available WiFi Networks

---

```

//First request will return 0 results unless you start scan from :
//Do not request more often than 3-5 seconds
server.on("/scan", HTTP_GET, [])(AsyncWebServerRequest *request){
    String json = "[";
    int n = WiFi.scanComplete();
    if(n == -2){
        WiFi.scanNetworks(true);
    } else if(n){
        for (int i = 0; i < n; ++i){
            if(i) json += ",";
            json += "{";
            json += "\"rssi\":\""+String(WiFi.RSSI(i));
            json += "\",\"ssid\":\""+WiFi.SSID(i)+"\"";
            json += "\",\"bssid\":\""+WiFi.BSSIDstr(i)+"\"";
            json += "\",\"channel\":\""+String(WiFi.channel(i));
            json += "\",\"secure\":\""+String(WiFi.encryptionType(i));
            json += "\",\"hidden\":\""+String(WiFi.isHidden(i))?"true":"false";
            json += "}";
        }
        WiFi.scanDelete();
        if(WiFi.scanComplete() == -2){
            WiFi.scanNetworks(true);
        }
    }
    json += "]";
    request->send(200, "application/json", json);
    json = String();
});

```

## Remove handlers and rewrites

---

Server goes through handlers in same order as they were added. You can't simply add handler with same path to override them. To remove handler:

```
// save callback for particular URL path
auto handler = server.on("/some/path", [](AsyncWebServerRequest *r) {
    //do something useful
});
// when you don't need handler anymore remove it
server.removeHandler(&handler);

// same with rewrites
server.removeRewrite(&someRewrite);

server.onNotFound([](AsyncWebServerRequest *request){
    request->send(404);
});

// remove server.onNotFound handler
server.onNotFound(NULL);

// remove all rewrites, handlers and onNotFound/onFileUpload/onRequestBody
server.reset();
```

## Setting up the server

---

```
#include "ESPAsyncTCP.h"
#include "ESPAsyncWebServer.h"

AsyncWebServer server(80);
AsyncWebSocket ws("/ws"); // access at ws://[esp ip]/ws
AsyncEventSource events("/events"); // event source (Server-Sent Events)

const char* ssid = "your-ssid";
const char* password = "your-pass";
const char* http_username = "admin";
const char* http_password = "admin";

//flag to use from web update to reboot the ESP
```

```

bool shouldReboot = false;

void onRequest(AsyncWebServerRequest *request){
    //Handle Unknown Request
    request->send(404);
}

void onBody(AsyncWebServerRequest *request, uint8_t *data, size_t
    //Handle body
}

void onUpload(AsyncWebServerRequest *request, String filename, si
    //Handle upload
}

void onEvent(AsyncWebSocket * server, AsyncWebSocketClient * clien
    //Handle WebSocket event
}

void setup(){
    Serial.begin(115200);
    WiFi.mode(WIFI_STA);
    WiFi.begin(ssid, password);
    if (WiFi.waitForConnectResult() != WL_CONNECTED) {
        Serial.printf("WiFi Failed!\n");
        return;
    }

    // attach AsyncWebSocket
    ws.onEvent(onEvent);
    server.addHandler(&ws);

    // attach AsyncEventSource
    server.addHandler(&events);

    // respond to GET requests on URL /heap
    server.on("/heap", HTTP_GET, [] (AsyncWebServerRequest *request){
        request->send(200, "text/plain", String(ESP.getFreeHeap()));
    });

    // upload a file to /upload
    server.on("/upload", HTTP_POST, [] (AsyncWebServerRequest *request:
        request->send(200);
    }, onUpload);

```



```

// send a file when /index is requested
server.on("/index", HTTP_ANY, [](AsyncWebServerRequest *request) {
    request->send(SPIFFS, "/index.htm");
});

// HTTP basic authentication
server.on("/login", HTTP_GET, [](AsyncWebServerRequest *request) {
    if(!request->authenticate(http_username, http_password))
        return request->requestAuthentication();
    request->send(200, "text/plain", "Login Success!");
});

// Simple Firmware Update Form
server.on("/update", HTTP_GET, [](AsyncWebServerRequest *request) {
    request->send(200, "text/html", "<form method='POST' action='");
});
server.on("/update", HTTP_POST, [](AsyncWebServerRequest *request) {
    shouldReboot = !Update.hasError();
    AsyncWebServerResponse *response = request->beginResponse(200,
    response->addHeader("Connection", "close");
    request->send(response);
}, [](AsyncWebServerRequest *request, String filename, size_t index, bool
    if(!index){
        Serial.printf("Update Start: %s\n", filename.c_str());
        Update.runAsync(true);
        if(!Update.begin((ESP.getFreeSketchSpace() - 0x1000) & 0xFF0000))
            Update.printError(Serial);
        }
    }
    if(!Update.hasError()){
        if(Update.write(data, len) != len){
            Update.printError(Serial);
        }
    }
    if(final){
        if(Update.end(true)){
            Serial.printf("Update Success: %uB\n", index+len);
        } else {
            Update.printError(Serial);
        }
    }
});

```

```

// attach filesystem root at URL /fs
server.serveStatic("/fs", SPIFFS, "/");

// Catch-All Handlers
// Any request that can not find a Handler that canHandle it
// ends in the callbacks below.
server.onNotFound(onRequest);
server.onFileUpload(onUpload);
server.onRequestBody(onBody);

server.begin();
}

void loop(){
  if(shouldReboot){
    Serial.println("Rebooting...");
    delay(100);
    ESP.restart();
  }
  static char temp[128];
  sprintf(temp, "Seconds since boot: %u", millis()/1000);
  events.send(temp, "time"); //send event "time"
}

```

## Setup global and class functions as request handlers

```

#include <Arduino.h>
#include <ESPAsyncWebserver.h>
#include <Hash.h>
#include <functional>

void handleRequest(AsyncWebServerRequest *request){}

class WebClass {
public :
  AsyncWebServer classWebServer = AsyncWebServer(81);

  WebClass(){};

  void classRequest (AsyncWebServerRequest *request){}

  void begin(){
    // attach global request handler

```

```

classWebServer.on("/example", HTTP_ANY, handleRequest);

// attach class request handler
classWebServer.on("/example", HTTP_ANY, std::bind(&WebClass::c
}
};

AsyncWebServer globalWebServer(80);
WebClass webClassInstance;

void setup() {
    // attach global request handler
    globalWebServer.on("/example", HTTP_ANY, handleRequest);

    // attach class request handler
    globalWebServer.on("/example", HTTP_ANY, std::bind(&WebClass::c
}

void loop() {

}

```

## Methods for controlling websocket connections

```

// Disable client connections if it was activated
if ( ws.enabled() )
    ws.enable(false);

// enable client connections if it was disabled
if ( !ws.enabled() )
    ws.enable(true);

```

### Example of OTA code

```

// OTA callbacks
ArduinoOTA.onStart([]() {
    // Clean SPIFFS
    SPIFFS.end();

    // Disable client connections
    ws.enable(false);

```

```
// Advertise connected clients what's going on
ws.textAll("OTA Update Started");

// Close them
ws.closeAll();

});
```

## Adding Default Headers

In some cases, such as when working with CORS, or with some sort of custom authentication system, you might need to define a header that should get added to all responses (including static, websocket and EventSource). The DefaultHeaders singleton allows you to do this.

Example:

```
DefaultHeaders::Instance().addHeader("Access-Control-Allow-Origin")
webServer.begin();
```

*NOTE:* You will still need to respond to the OPTIONS method for CORS pre-flight in most cases. (unless you are only using GET)

This is one option:

```
webServer.onNotFound([](AsyncWebServerRequest *request) {
    if (request->method() == HTTP_OPTIONS) {
        request->send(200);
    } else {
        request->send(404);
    }
});
```

## Path variable

With path variable you can create a custom regex rule for a specific parameter in a route. For example we want a `sensorId` parameter in a route rule to match only a integer.

```
server.on("^\\/sensor\\/([0-9]+)$", HTTP_GET, [] (AsyncWebServerRequest request) {
    String sensorId = request->pathArg(0);
});
```

*NOTE:* All regex patterns starts with ^ and ends with \$

To enable the **Path variable** support, you have to define the buildflag **-DASYNCWEBSERVER\_REGEX**.

For Arduino IDE create/update **platform.local.txt**:

**Windows:** C:

\Users(username)\AppData\Local\Arduino15\packages\{espxxxx}\hardware\espxxxx\{version}\platform.local.txt

**Linux:** ~/.arduino15/packages/{espxxxx}/hardware/{espxxxx}/{version}/platform.local.txt

Add/Update the following line:

```
compiler.cpp.extra_flags=-DASYNCWEBSERVER_REGEX
```

For platformio modify **platformio.ini**:

```
[env:myboard]
build_flags =
    -DASYNCWEBSERVER_REGEX
```

*NOTE:* By enabling **ASYNCWEBSERVER\_REGEX**, **<regex>** will be included. This will add an 100k to your binary.

Homepage

 [github.com/mathieucarbou/ESPAsyncWebServer](https://github.com/mathieucarbou/ESPAsyncWebServer)

Repository

 [github.com/mathieucarbou/ESPAsyncWebServer.git](https://github.com/mathieucarbou/ESPAsyncWebServer.git)

Popularity

 36 ▼ 2

License

LGPL-3.0

Stars

☆ 109

Issues

0

Last Modified

2 months ago

Installed Size

388 KiB

Versions 44

 3.6.0 Latest

2 months ago

[+ 43 versions](#)

Authors 3



Dependencies 3

 [ESPAsyncTCP-esphome](#)

 [AsyncTCP](#)

 [AsyncTCP\\_RP2040W](#)

Used By 10

 [ElegantOTA](#)

 [WebSerialLite](#)

 [CODROB\\_IOTBOT](#)

[+ 7 dependents](#)

Badge



[HTML](#) • [Markdown](#)

Share



Report

[Claim Ownership](#)

[Report Abuse](#)



# PlatformIO Registry

Designed secure. Built for anything.



© 2014–2025 PlatformIO Labs OÜ.

## Company

[About](#)

[Blog](#)

[Advertising](#)

[Trademarks](#)

[Code of Conduct](#)

[Privacy & Terms](#)

[Contact Us](#)

## For Developers

[PlatformIO IDE](#)

[Debugging](#)

[Unit Testing](#)

[Static Code Analysis](#)

[Remote Development](#)

[Continuous Integration](#)

## For Businesses

[Technology as a Service](#)

[In-house IDE or Next-Gen?](#)

[Next-Gen IDE Framework](#)

[Modern UI Toolkit](#)

[Trusted Package Registry](#)

## Support

[Docs](#)

[Community Forum](#)

[Premium Support](#)

[Report a Bug](#)

[Donate](#)

[Status](#)