

Cowboy User Guide

Contents

I	Rationale	1
1	The modern Web	2
1.1	HTTP/2	2
1.2	HTTP/1.1	2
1.3	Websocket	2
1.4	Long-lived requests	3
1.5	REST	3
2	Erlang and the Web	4
2.1	The Web is concurrent	4
2.2	The Web is soft real time	4
2.3	The Web is asynchronous	5
2.4	The Web is omnipresent	5
2.5	Learn Erlang	6
2.5.1	The Erlanger Playbook	6
2.5.2	Programming Erlang	6
2.5.3	Learn You Some Erlang for Great Good!	6
II	Introduction	7
3	Introduction	8
3.1	Prerequisites	8
3.2	Supported platforms	8
3.3	License	8
3.4	Versioning	9
3.5	Conventions	9

4	Getting started	10
4.1	Prerequisites	10
4.2	Bootstrap	10
4.3	Cowboy setup	11
4.4	Listening for connections	11
4.5	Handling requests	11
5	Flow diagram	13
5.1	Overview	13
5.2	Protocol-specific headers	14
5.3	Number of processes per connection	14
5.4	Date header	14
5.5	Binaries	14
III	Configuration	16
6	Listeners	17
6.1	Clear TCP listener	17
6.2	Secure TLS listener	17
6.3	Protocol configuration	18
7	Routing	19
7.1	Syntax	19
7.2	Match syntax	19
7.3	Constraints	21
7.4	Compilation	21
7.5	Live update	21
8	Constraints	22
8.1	Syntax	22
8.2	Built-in constraints	22
8.3	Custom constraints	23
IV	Handlers	24
9	Handlers	25
9.1	Plain HTTP handlers	25
9.2	Other handlers	25
9.3	Cleaning up	26

10 Loop handlers	27
10.1 Initialization	27
10.2 Receive loop	27
10.3 Streaming loop	28
10.4 Cleaning up	28
10.5 Hibernate	28
11 Static files	29
11.1 Serve one file	29
11.2 Serve all files from a directory	29
11.3 Customize the mimetype detection	30
11.4 Generate an etag	30
V Request and response	31
12 The Req object	32
12.1 Direct access	32
12.2 Introduction to the cowboy_req interface	32
12.3 Request method	33
12.4 HTTP version	33
12.5 Effective request URI	33
12.6 Bindings	34
12.7 Query parameters	35
12.8 Headers	35
12.9 Peer	36
13 Reading the request body	37
13.1 Request body presence	37
13.2 Request body length	37
13.3 Reading the body	37
13.4 Streaming the body	38
13.5 Reading a form urlencoded body	38
14 Sending a response	39
14.1 Reply	39
14.2 Stream reply	40
14.3 Preset response headers	40
14.4 Overriding headers	41
14.5 Preset response body	41
14.6 Sending files	41
14.7 Push	42

15 Using cookies	43
15.1 Setting cookies	43
15.2 Reading cookies	44
16 Multipart requests	45
16.1 Form-data	45
16.2 Checking for multipart messages	45
16.3 Reading a multipart message	45
16.4 Skipping unwanted parts	46
VI REST	48
17 REST principles	49
17.1 REST architecture	49
17.2 Resources and resource identifiers	49
17.3 Resource representations	50
17.4 Self-descriptive messages	50
17.5 Hypermedia as the engine of application state	50
18 REST handlers	51
18.1 Initialization	51
18.2 Methods	51
18.3 Callbacks	51
18.4 Meta data	52
18.5 Response headers	52
19 REST flowcharts	54
19.1 Start	54
19.2 OPTIONS method	56
19.3 Content negotiation	56
19.4 GET and HEAD methods	58
19.5 PUT, POST and PATCH methods	60
19.6 DELETE method	62
19.7 Conditional requests	64
20 Designing a resource handler	67
20.1 The service	67
20.2 Type of resource handler	67
20.3 Collection handler	67
20.4 Single resource handler	68
20.5 The resource	68

20.6 Representations	68
20.7 Redirections	69
20.8 The request	69
20.9 OPTIONS method	69
20.10 GET and HEAD methods	69
20.11 PUT, POST and PATCH methods	69
20.12 DELETE methods	69

VII Websocket 70

21 The Websocket protocol 71

21.1 Description	71
21.2 Websocket vs HTTP/2	71
21.3 Implementation	71

22 Websocket handlers 72

22.1 Upgrade	72
22.2 Subprotocol	72
22.3 Post-upgrade initialization	73
22.4 Receiving frames	73
22.5 Receiving Erlang messages	73
22.6 Sending frames	74
22.7 Keeping the connection alive	74
22.8 Saving memory	75
22.9 Closing the connection	75

VIII Advanced 76

23 Streams 77

23.1 Stream handlers	77
23.2 Built-in handlers	77

24 Middlewares 78

24.1 Usage	78
24.2 Configuration	78
24.3 Routing middleware	79
24.4 Handler middleware	79

IX	Additional information	80
A	Migrating from Cowboy 1.0 to 2.0	81
A.1	Compatibility	81
A.2	Features added	81
A.3	Features removed	82
A.4	Changed behaviors	82
A.5	New functions	82
A.6	Changed functions	83
A.7	Removed functions	83
A.8	Other changes	83
B	HTTP and other specifications	84
B.1	HTTP	84
B.1.1	IANA Registries	84
B.1.2	Current	84
B.1.3	Upcoming	86
B.1.4	Informative	87
B.1.5	Related	87
B.1.6	Seemingly obsolete	88
B.2	URL	88
B.3	WebDAV	88
B.4	CoAP	89

Part I

Rationale

Chapter 1

The modern Web

Cowboy is a server for the modern Web. This chapter explains what it means and details all the standards involved.

Cowboy supports all the standards listed in this document.

1.1 HTTP/2

HTTP/2 is the most efficient protocol for consuming Web services. It enables clients to keep a connection open for long periods of time; to send requests concurrently; to reduce the size of requests through HTTP headers compression; and more. The protocol is binary, greatly reducing the resources needed to parse it.

HTTP/2 also enables the server to push messages to the client. This can be used for various purposes, including the sending of related resources before the client requests them, in an effort to reduce latency. This can also be used to enable bidirectional communication.

Cowboy provides transparent support for HTTP/2. Clients that know it can use it; others fall back to HTTP/1.1 automatically.

HTTP/2 is compatible with the HTTP/1.1 semantics.

HTTP/2 is defined by RFC 7540 and RFC 7541.

1.2 HTTP/1.1

HTTP/1.1 is the previous version of the HTTP protocol. The protocol itself is text-based and suffers from numerous issues and limitations. In particular it is not possible to execute requests concurrently (though pipelining is sometimes possible), and it's also sometimes difficult to detect that a client disconnected.

HTTP/1.1 does provide very good semantics for interacting with Web services. It defines the standard methods, headers and status codes used by HTTP/1.1 and HTTP/2 clients and servers.

HTTP/1.1 also defines compatibility with an older version of the protocol, HTTP/1.0, which was never really standardized across implementations.

The core of HTTP/1.1 is defined by RFC 7230, RFC 7231, RFC 7232, RFC 7233, RFC 7234 and RFC 7235. Numerous RFCs and other specifications exist defining additional HTTP methods, status codes, headers or semantics.

1.3 WebSocket

[WebSocket](#) is a protocol built on top of HTTP/1.1 that provides a two-ways communication channel between the client and the server. Communication is asynchronous and can occur concurrently.

It consists of a Javascript object allowing setting up a Websocket connection to the server, and a binary based protocol for sending data to the server or the client.

Websocket connections can transfer either UTF-8 encoded text data or binary data. The protocol also includes support for implementing a ping/pong mechanism, allowing the server and the client to have more confidence that the connection is still alive.

A Websocket connection can be used to transfer any kind of data, small or big, text or binary. Because of this Websocket is sometimes used for communication between systems.

Websocket messages have no semantics on their own. Websocket is closer to TCP in that aspect, and requires you to design and implement your own protocol on top of it; or adapt an existing protocol to Websocket.

Cowboy provides an interface known as [Websocket handlers](#) that gives complete control over a Websocket connection.

The Websocket protocol is defined by RFC 6455.

1.4 Long-lived requests

Cowboy provides an interface that can be used to support long-polling or to stream large amounts of data reliably, including using Server-Sent Events.

Long-polling is a mechanism in which the client performs a request which may not be immediately answered by the server. It allows clients to request resources that may not currently exist, but are expected to be created soon, and which will be returned as soon as they are.

Long-polling is essentially a hack, but it is widely used to overcome limitations on older clients and servers.

Server-Sent Events is a small protocol defined as a media type, `text/event-stream`, along with a new HTTP header, `Last-Event-ID`. It is defined in the EventSource W3C specification.

Cowboy provides an interface known as [loop handlers](#) that facilitates the implementation of long-polling or stream mechanisms. It works regardless of the underlying protocol.

1.5 REST

[REST, or REpresentational State Transfer](#), is a style of architecture for loosely connected distributed systems. It can easily be implemented on top of HTTP.

REST is essentially a set of constraints to be followed. Many of these constraints are purely architectural and solved by simply using HTTP. Some constraints must be explicitly followed by the developer.

Cowboy provides an interface known as [REST handlers](#) that simplifies the implementation of a REST API on top of the HTTP protocol.

Chapter 2

Erlang and the Web

Erlang is the ideal platform for writing Web applications. Its features are a perfect match for the requirements of modern Web applications.

2.1 The Web is concurrent

When you access a website there is little concurrency involved. A few connections are opened and requests are sent through these connections. Then the web page is displayed on your screen. Your browser will only open up to 4 or 8 connections to the server, depending on your settings. This isn't much.

But think about it. You are not the only one accessing the server at the same time. There can be hundreds, if not thousands, if not millions of connections to the same server at the same time.

Even today a lot of systems used in production haven't solved the C10K problem (ten thousand concurrent connections). And the ones who did are trying hard to get to the next step, C100K, and are pretty far from it.

Erlang meanwhile has no problem handling millions of connections. At the time of writing there are application servers written in Erlang that can handle more than two million connections on a single server in a real production application, with spare memory and CPU!

The Web is concurrent, and Erlang is a language designed for concurrency, so it is a perfect match.

Of course, various platforms need to scale beyond a few million connections. This is where Erlang's built-in distribution mechanisms come in. If one server isn't enough, add more! Erlang allows you to use the same code for talking to local processes or to processes in other parts of your cluster, which means you can scale very quickly if the need arises.

The Web has large userbases, and the Erlang platform was designed to work in a distributed setting, so it is a perfect match.

Or is it? Surely you can find solutions to handle that many concurrent connections with your favorite language... But all these solutions will break down in the next few years. Why? Firstly because servers don't get any more powerful, they instead get a lot more cores and memory. This is only useful if your application can use them properly, and Erlang is light-years away from anything else in that area. Secondly, today your computer and your phone are online, tomorrow your watch, goggles, bike, car, fridge and tons of other devices will also connect to various applications on the Internet.

Only Erlang is prepared to deal with what's coming.

2.2 The Web is soft real time

What does soft real time mean, you ask? It means we want the operations done as quickly as possible, and in the case of web applications, it means we want the data propagated fast.

In comparison, hard real time has a similar meaning, but also has a hard time constraint, for example an operation needs to be done in under N milliseconds otherwise the system fails entirely.

Users aren't that needy yet, they just want to get access to their content in a reasonable delay, and they want the actions they make to register at most a few seconds after they submitted them, otherwise they'll start worrying about whether it successfully went through.

The Web is soft real time because taking longer to perform an operation would be seen as bad quality of service.

Erlang is a soft real time system. It will always run processes fairly, a little at a time, switching to another process after a while and preventing a single process to steal resources from all others. This means that Erlang can guarantee stable low latency of operations.

Erlang provides the guarantees that the soft real time Web requires.

2.3 The Web is asynchronous

Long ago, the Web was synchronous because HTTP was synchronous. You fired a request, and then waited for a response. Not anymore. It all began when XMLHttpRequest started being used. It allowed the client to perform asynchronous calls to the server.

Then Websocket appeared and allowed both the server and the client to send data to the other endpoint completely asynchronously. The data is contained within frames and no response is necessary.

Erlang processes work the same. They send each other data contained within messages and then continue running without needing a response. They tend to spend most of their time inactive, waiting for a new message, and the Erlang VM happily activate them when one is received.

It is therefore quite easy to imagine Erlang being good at receiving Websocket frames, which may come in at unpredictable times, pass the data to the responsible processes which are always ready waiting for new messages, and perform the operations required by only activating the required parts of the system.

The more recent Web technologies, like Websocket of course, but also HTTP/2.0, are all fully asynchronous protocols. The concept of requests and responses is retained of course, but anything could be sent in between, by both the client or the browser, and the responses could also be received in a completely different order.

Erlang is by nature asynchronous and really good at it thanks to the great engineering that has been done in the VM over the years. It's only natural that it's so good at dealing with the asynchronous Web.

2.4 The Web is omnipresent

The Web has taken a very important part of our lives. We're connected at all times, when we're on our phone, using our computer, passing time using a tablet while in the bathroom. . . And this isn't going to slow down, every single device at home or on us will be connected.

All these devices are always connected. And with the number of alternatives to give you access to the content you seek, users tend to not stick around when problems arise. Users today want their applications to be always available and if it's having too many issues they just move on.

Despite this, when developers choose a product to use for building web applications, their only concern seems to be "Is it fast?", and they look around for synthetic benchmarks showing which one is the fastest at sending "Hello world" with only a handful concurrent connections. Web benchmarks haven't been representative of reality in a long time, and are drifting further away as time goes on.

What developers should really ask themselves is "Can I service all my users with no interruption?" and they'd find that they have two choices. They can either hope for the best, or they can use Erlang.

Erlang is built for fault tolerance. When writing code in any other language, you have to check all the return values and act accordingly to avoid any unforeseen issues. If you're lucky, you won't miss anything important. When writing Erlang code, you can just check the success condition and ignore all errors. If an error happens, the Erlang process crashes and is then restarted by a special process called a supervisor.

Erlang developers thus have no need to fear unhandled errors, and can focus on handling only the errors that should give some feedback to the user and let the system take care of the rest. This also has the advantage of allowing them to write a lot less code, and let them sleep at night.

Erlang's fault tolerance oriented design is the first piece of what makes it the best choice for the omnipresent, always available Web.

The second piece is Erlang's built-in distribution. Distribution is a key part of building a fault tolerant system, because it allows you to handle bigger failures, like a whole server going down, or even a data center entirely.

Fault tolerance and distribution are important today, and will be vital in the future of the Web. Erlang is ready.

2.5 Learn Erlang

If you are new to Erlang, you may want to grab a book or two to get started. Those are my recommendations as the author of Cowboy.

2.5.1 The Erlanger Playbook

The Erlanger Playbook is an ebook I am currently writing, which covers a number of different topics from code to documentation to testing Erlang applications. It also has an Erlang section where it covers directly the building blocks and patterns, rather than details like the syntax.

You can most likely read it as a complete beginner, but you will need a companion book to make the most of it. Buy it from the [Nine Nines website](#).

2.5.2 Programming Erlang

This book is from one of the creator of Erlang, Joe Armstrong. It provides a very good explanation of what Erlang is and why it is so. It serves as a very good introduction to the language and platform.

The book is [Programming Erlang](#), and it also features a chapter on Cowboy.

2.5.3 Learn You Some Erlang for Great Good!

[LYSE](#) is a much more complete book covering many aspects of Erlang, while also providing stories and humor. Be warned: it's pretty verbose. It comes with a free online version and a more refined paper and ebook version.

Part II

Introduction

Chapter 3

Introduction

Cowboy is a small, fast and modern HTTP server for Erlang/OTP.

Cowboy aims to provide a complete [modern Web stack](#). This includes HTTP/1.1, HTTP/2, Websocket, Server-Sent Events and Webmachine-based REST.

Cowboy comes with functions for introspection and tracing, enabling developers to know precisely what is happening at any time. Its modular design also easily enable developers to add instrumentation.

Cowboy is a high quality project. It has a small code base, is very efficient (both in latency and memory use) and can easily be embedded in another application.

Cowboy is clean Erlang code. It includes hundreds of tests and its code is fully compliant with the Dialyzer. It is also well documented and features a Function Reference, a User Guide and numerous Tutorials.

3.1 Prerequisites

Beginner Erlang knowledge is recommended for reading this guide.

Knowledge of the HTTP protocol is recommended but not required, as it will be detailed throughout the guide.

3.2 Supported platforms

Cowboy is tested and supported on Linux, FreeBSD, Windows and OSX.

Cowboy has been reported to work on other platforms, but we make no guarantee that the experience will be safe and smooth. You are advised to perform the necessary testing and security audits prior to deploying on other platforms.

Cowboy is developed for Erlang/OTP 19.0 and newer.

3.3 License

Cowboy uses the ISC License.

```
Copyright (c) 2011-2017, Loïc Hoguin <essen@ninenines.eu>
```

```
Permission to use, copy, modify, and/or distribute this software for any  
purpose with or without fee is hereby granted, provided that the above  
copyright notice and this permission notice appear in all copies.
```

```
THE SOFTWARE IS PROVIDED "AS IS" AND THE AUTHOR DISCLAIMS ALL WARRANTIES
```

WITH REGARD TO THIS SOFTWARE INCLUDING ALL IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS. IN NO EVENT SHALL THE AUTHOR BE LIABLE FOR ANY SPECIAL, DIRECT, INDIRECT, OR CONSEQUENTIAL DAMAGES OR ANY DAMAGES WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS, WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE OR PERFORMANCE OF THIS SOFTWARE.

3.4 Versioning

Cowboy uses [Semantic Versioning 2.0.0](#).

3.5 Conventions

In the HTTP protocol, the method name is case sensitive. All standard method names are uppercase.

Header names are case insensitive. When using HTTP/1.1, Cowboy converts all the request header names to lowercase. HTTP/2 requires clients to send them as lowercase. Any other header name is expected to be provided lowercased, including when querying information about the request or when sending responses.

The same applies to any other case insensitive value.

Chapter 4

Getting started

Erlang is more than a language, it is also an operating system for your applications. Erlang developers rarely write standalone modules, they write libraries or applications, and then bundle those into what is called a release. A release contains the Erlang VM plus all applications required to run the node, so it can be pushed to production directly.

This chapter walks you through all the steps of setting up Cowboy, writing your first application and generating your first release. At the end of this chapter you should know everything you need to push your first Cowboy application to production.

4.1 Prerequisites

We are going to use the [Erlang.mk](#) build system. If you are using Windows, please check the [Installation instructions](#) to get your environment setup before you continue.

4.2 Bootstrap

First, let's create the directory for our application.

```
$ mkdir hello_erlang
$ cd hello_erlang
```

Then we need to download Erlang.mk. Either use the following command or download it manually.

```
$ wget https://erlang.mk/erlang.mk
```

We can now bootstrap our application. Since we are going to generate a release, we will also bootstrap it at the same time.

```
$ make -f erlang.mk bootstrap bootstrap-rel
```

This creates a Makefile, a base application, and the release files necessary for creating the release. We can already build and start this release.

```
$ make run
...
(hello_erlang@127.0.0.1)1>
```

Entering the command `i()` will show the running processes, including one called `hello_erlang_sup`. This is the supervisor for our application.

The release currently does nothing. In the rest of this chapter we will add Cowboy as a dependency and write a simple "Hello world!" handler.

4.3 Cowboy setup

We will modify the *Makefile* to tell the build system it needs to fetch and compile Cowboy:

```
PROJECT = hello_erlang

DEPS = cowboy
dep_cowboy_commit = 2.0.0

DEP_PLUGINS = cowboy

include erlang.mk
```

We also tell the build system to load the plugins Cowboy provides. These include predefined templates that we will use soon.

If you do `make run` now, Cowboy will be included in the release and started automatically. This is not enough however, as Cowboy doesn't do anything by default. We still need to tell Cowboy to listen for connections.

4.4 Listening for connections

First we define the routes that Cowboy will use to map requests to handler modules, and then we start the listener. This is best done at application startup.

Open the `src/hello_erlang_app.erl` file and add the necessary code to the `start/2` function to make it look like this:

```
start(_Type, _Args) ->
    Dispatch = cowboy_router:compile([
        {'_', [{"/", hello_handler, []}]}
    ]),
    {ok, _} = cowboy:start_clear(my_http_listener,
        [{port, 8080}],
        #{env => #{dispatch => Dispatch}}
    ),
    hello_erlang_sup:start_link().
```

Routes are explained in details in the [Routing](#) chapter. For this tutorial we map the path `/` to the handler module `hello_handler`. This module doesn't exist yet.

Build and start the release, then open <http://localhost:8080> in your browser. You will get a 500 error because the module is missing. Any other URL, like <http://localhost:8080/test>, will result in a 404 error.

4.5 Handling requests

Cowboy features different kinds of handlers, including REST and Websocket handlers. For this tutorial we will use a plain HTTP handler.

Generate a handler from a template:

```
$ make new t=cowboy.http n=hello_handler
```

Then, open the `src/hello_handler.erl` file and modify the `init/2` function like this to send a reply.

```
init(Req0, State) ->
    Req = cowboy_req:reply(200,
        #{<<"content-type">> => <<"text/plain">>},
        <<"Hello Erlang!">>,
        Req0),
    {ok, Req, State}.
```

What the above code does is send a 200 OK reply, with the Content-type header set to `text/plain` and the response body set to `Hello Erlang!`.

If you run the release and open <http://localhost:8080> in your browser, you should get a nice `Hello Erlang!` displayed!

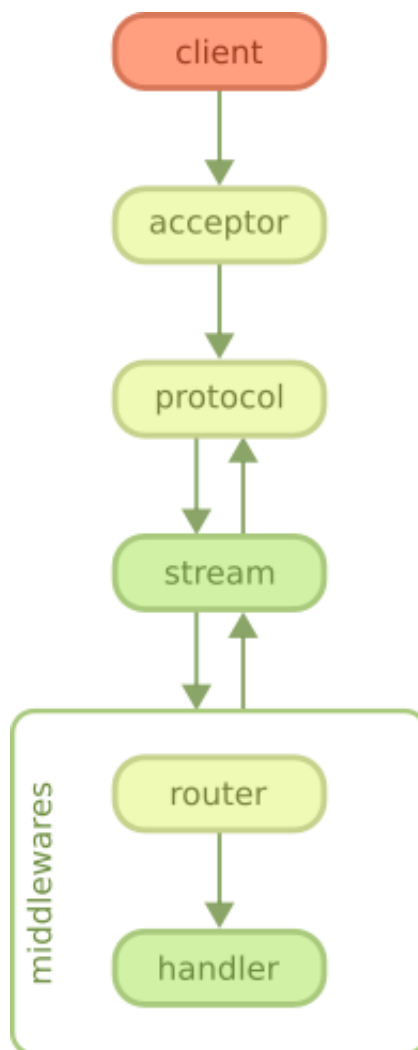
Chapter 5

Flow diagram

Cowboy is a lightweight HTTP server with support for HTTP/1.1, HTTP/2 and Websocket.

It is built on top of Ranch. Please see the Ranch guide for more information about how the network connections are handled.

5.1 Overview



As you can see on the diagram, the client begins by connecting to the server. This step is handled by a Ranch acceptor, which is a process dedicated to accepting new connections.

After Ranch accepts a new connection, whether it is an HTTP/1.1 or HTTP/2 connection, Cowboy starts receiving requests and handling them.

In HTTP/1.1 all requests come sequentially. In HTTP/2 the requests may arrive and be processed concurrently.

When a request comes in, Cowboy creates a stream, which is a set of request/response and all the events associated with them. The protocol code in Cowboy defers the handling of these streams to stream handler modules. When you configure Cowboy you may define one or more module that will receive all events associated with a stream, including the request, response, bodies, Erlang messages and more.

By default Cowboy comes configured with a stream handler called `cowboy_stream_h`. This stream handler will create a new process for every request coming in, and then communicate with this process to read the body or send a response back. The request process executes middlewares which, by default, including the router and then the execution of handlers. Like stream handlers, middlewares may also be customized.

A response may be sent at almost any point in this diagram. If the response must be sent before the stream is initialized (because an error occurred early, for example) then stream handlers receive a special event indicating this error.

5.2 Protocol-specific headers

Cowboy takes care of protocol-specific headers and prevents you from sending them manually. For HTTP/1.1 this includes the `transfer-encoding` and `connection` headers. For HTTP/2 this includes the colon headers like `:status`.

Cowboy will also remove protocol-specific headers from requests before passing them to stream handlers. Cowboy tries to hide the implementation details of all protocols as well as possible.

5.3 Number of processes per connection

By default, Cowboy will use one process per connection, plus one process per set of request/response (called a stream, internally).

The reason it creates a new process for every request is due to the requirements of HTTP/2 where requests are executed concurrently and independently from the connection. The frames from the different requests end up interleaved on the single TCP connection.

The request processes are never reused. There is therefore no need to perform any cleanup after the response has been sent. The process will terminate and Erlang/OTP will reclaim all memory at once.

Cowboy ultimately does not require more than one process per connection. It is possible to interact with the connection directly from a stream handler, a low level interface to Cowboy. They are executed from within the connection process, and can handle the incoming requests and send responses. This is however not recommended in normal circumstances, as a stream handler taking too long to execute could have a negative impact on concurrent requests or the state of the connection itself.

5.4 Date header

Because querying for the current date and time can be expensive, Cowboy generates one *Date* header value every second, shares it to all other processes, which then simply copy it in the response. This allows compliance with HTTP/1.1 with no actual performance loss.

5.5 Binaries

Cowboy makes extensive use of binaries.

Binaries are more efficient than lists for representing strings because they take less memory space. Processing performance can vary depending on the operation. Binaries are known for generally getting a great boost if the code is compiled natively. Please see the HiPE documentation for more details.

Binaries may end up being shared between processes. This can lead to some large memory usage when one process keeps the binary data around forever without freeing it. If you see some weird memory usage in your application, this might be the cause.

Part III

Configuration

Chapter 6

Listeners

A listener is a set of processes that listens on a port for new connections. Incoming connections get handled by Cowboy. Depending on the connection handshake, one or another protocol may be used.

This chapter is specific to Cowboy. Please refer to the [Ranch User Guide](#) for more information about listeners.

Cowboy provides two types of listeners: one listening for clear TCP connections, and one listening for secure TLS connections. Both of them support the HTTP/1.1 and HTTP/2 protocols.

6.1 Clear TCP listener

The clear TCP listener will accept connections on the given port. A typical HTTP server would listen on port 80. Port 80 requires special permissions on most platforms however so a common alternative is port 8080.

The following snippet starts listening for connections on port 8080:

```
start(_Type, _Args) ->
    Dispatch = cowboy_router:compile([
        {'_', [{"/", hello_handler, []}]}
    ]),
    {ok, _} = cowboy:start_clear(my_http_listener,
        [{port, 8080}],
        #{env => #{dispatch => Dispatch}}
    ),
    hello_erlang_sup:start_link().
```

The [Getting Started](#) chapter uses a clear TCP listener.

Clients connecting to Cowboy on the clear listener port are expected to use either HTTP/1.1 or HTTP/2.

Cowboy supports both methods of initiating a clear HTTP/2 connection: through the Upgrade mechanism ([RFC 7540 3.2](#)) or by sending the preface directly ([RFC 7540 3.4](#)).

Compatibility with HTTP/1.0 is provided by Cowboy's HTTP/1.1 implementation.

6.2 Secure TLS listener

The secure TLS listener will accept connections on the given port. A typical HTTPS server would listen on port 443. Port 443 requires special permissions on most platforms however so a common alternative is port 8443.

The function provided by Cowboy will ensure that the TLS options given are following the HTTP/2 RFC with regards to security. For example some TLS extensions or ciphers may be disabled. This also applies to HTTP/1.1 connections on this listener. If this is not desirable, Ranch can be used directly to setup a custom listener.

```
start(_Type, _Args) ->
    Dispatch = cowboy_router:compile([
        {'_', [{"_", hello_handler, []}]}
    ]),
    {ok, _} = cowboy:start_tls(my_http_listener,
        [
            {port, 8443},
            {certfile, "/path/to/certfile"},
            {keyfile, "/path/to/keyfile"}
        ],
        #{env => #{dispatch => Dispatch}}
    ),
    hello_erlang_sup:start_link().
```

Clients connecting to Cowboy on the secure listener are expected to use the ALPN TLS extension to indicate what protocols they understand. Cowboy always prefers HTTP/2 over HTTP/1.1 when both are supported. When neither are supported by the client, or when the ALPN extension was missing, Cowboy expects HTTP/1.1 to be used.

Cowboy also advertises HTTP/2 support through the older NPN TLS extension for compatibility. Note however that this support will likely not be enabled by default when Cowboy 2.0 gets released.

Compatibility with HTTP/1.0 is provided by Cowboy's HTTP/1.1 implementation.

6.3 Protocol configuration

The HTTP/1.1 and HTTP/2 protocols share the same semantics; only their framing differs. The first is a text protocol and the second a binary protocol.

Cowboy doesn't separate the configuration for HTTP/1.1 and HTTP/2. Everything goes into the same map. Many options are shared.

Chapter 7

Routing

Cowboy does nothing by default.

To make Cowboy useful, you need to map URIs to Erlang modules that will handle the requests. This is called routing.

When Cowboy receives a request, it tries to match the requested host and path to the configured routes. When there's a match, the route's associated handler is executed.

Routes need to be compiled before they can be used by Cowboy. The result of the compilation is the dispatch rules.

7.1 Syntax

The general structure for the routes is defined as follow.

```
Routes = [Host1, Host2, ... HostN].
```

Each host contains matching rules for the host along with optional constraints, and a list of routes for the path component.

```
Host1 = {HostMatch, PathsList}.  
Host2 = {HostMatch, Constraints, PathsList}.
```

The list of routes for the path component is defined similar to the list of hosts.

```
PathsList = [Path1, Path2, ... PathN].
```

Finally, each path contains matching rules for the path along with optional constraints, and gives us the handler module to be used along with its initial state.

```
Path1 = {PathMatch, Handler, InitialState}.  
Path2 = {PathMatch, Constraints, Handler, InitialState}.
```

Continue reading to learn more about the match syntax and the optional constraints.

7.2 Match syntax

The match syntax is used to associate host names and paths with their respective handlers.

The match syntax is the same for host and path with a few subtleties. Indeed, the segments separator is different, and the host is matched starting from the last segment going to the first. All examples will feature both host and path match rules and explain the differences when encountered.

Excluding special values that we will explain at the end of this section, the simplest match value is a host or a path. It can be given as either a `string()` or a `binary()`.

```
PathMatch1 = "/" .
PathMatch2 = "/path/to/resource" .

HostMatch1 = "cowboy.example.org" .
```

As you can see, all paths defined this way must start with a slash character. Note that these two paths are identical as far as routing is concerned.

```
PathMatch2 = "/path/to/resource" .
PathMatch3 = "/path/to/resource/" .
```

Hosts with and without a trailing dot are equivalent for routing. Similarly, hosts with and without a leading dot are also equivalent.

```
HostMatch1 = "cowboy.example.org" .
HostMatch2 = "cowboy.example.org." .
HostMatch3 = ".cowboy.example.org" .
```

It is possible to extract segments of the host and path and to store the values in the `Req` object for later use. We call these kind of values bindings.

The syntax for bindings is very simple. A segment that begins with the `:` character means that what follows until the end of the segment is the name of the binding in which the segment value will be stored.

```
PathMatch = "/hats/:name/prices" .
HostMatch = ":subdomain.example.org" .
```

If these two end up matching when routing, you will end up with two bindings defined, `subdomain` and `name`, each containing the segment value where they were defined. For example, the URL `http://test.example.org/hats/wild_cowboy_legendary` will result in having the value `test` bound to the name `subdomain` and the value `wild_cowboy_legendary` bound to the name `name`. They can later be retrieved using `cowboy_req:binding/{2,3}`. The binding name must be given as an atom.

There is a special binding name you can use to mimic the underscore variable in Erlang. Any match against the `_` binding will succeed but the data will be discarded. This is especially useful for matching against many domain names in one go.

```
HostMatch = "ninenines.:_".
```

Similarly, it is possible to have optional segments. Anything between brackets is optional.

```
PathMatch = "/hats/[page/:number]" .
HostMatch = "[www.]ninenines.eu" .
```

You can also have imbricated optional segments.

```
PathMatch = "/hats/[page/[:number]]" .
```

You can retrieve the rest of the host or path using `[...]`. In the case of hosts it will match anything before, in the case of paths anything after the previously matched segments. It is a special case of optional segments, in that it can have zero, one or many segments. You can then find the segments using `cowboy_req:host_info/1` and `cowboy_req:path_info/1` respectively. They will be represented as a list of segments.

```
PathMatch = "/hats/[...]" .
HostMatch = "[...]ninenines.eu" .
```

If a binding appears twice in the routing rules, then the match will succeed only if they share the same value. This copies the Erlang pattern matching behavior.

```
PathMatch = "/hats/:name/:name" .
```

This is also true when an optional segment is present. In this case the two values must be identical only if the segment is available.

```
PathMatch = "/hats/:name/[:name]" .
```

If a binding is defined in both the host and path, then they must also share the same value.

```
PathMatch = "[:user/[...]]".
HostMatch = ":user.github.com".
```

Finally, there are two special match values that can be used. The first is the atom `'_'` which will match any host or path.

```
PathMatch = '_'.
HostMatch = '_'.
```

The second is the special host match `"*"` which will match the wildcard path, generally used alongside the `OPTIONS` method.

```
HostMatch = "*".
```

7.3 Constraints

After the matching has completed, the resulting bindings can be tested against a set of constraints. Constraints are only tested when the binding is defined. They run in the order you defined them. The match will succeed only if they all succeed. If the match fails, then Cowboy tries the next route in the list.

The format used for constraints is the same as match functions in `cowboy_req`: they are provided as a list of fields which may have one or more constraints. While the router accepts the same format, it will skip fields with no constraints and will also ignore default values, if any.

Read more about [constraints](#).

7.4 Compilation

The routes must be compiled before Cowboy can use them. The compilation step normalizes the routes to simplify the code and speed up the execution, but the routes are still looked up one by one in the end. Faster compilation strategies could be to compile the routes directly to Erlang code, but would require heavier dependencies.

To compile routes, just call the appropriate function:

```
Dispatch = cowboy_router:compile([
    %% {HostMatch, list({PathMatch, Handler, InitialState})}
    {'_', [{['_', my_handler, #{}]}]
}],
%% Name, NbAcceptors, TransOpts, ProtoOpts
cowboy:start_clear(my_http_listener,
    [{port, 8080}],
    #{env => #{dispatch => Dispatch}}
).
```

7.5 Live update

You can use the `cowboy:set_env/3` function for updating the dispatch list used by routing. This will apply to all new connections accepted by the listener:

```
Dispatch = cowboy_router:compile(Routes),
cowboy:set_env(my_http_listener, dispatch, Dispatch).
```

Note that you need to compile the routes again before updating.

Chapter 8

Constraints

Constraints are validation and conversion functions applied to user input.

They are used in various places in Cowboy, including the router and the `cowboy_req` match functions.

8.1 Syntax

Constraints are provided as a list of fields. For each field in the list, specific constraints can be applied, as well as a default value if the field is missing.

A field can take the form of an atom `field`, a tuple with constraints `{field, Constraints}` or a tuple with constraints and a default value `{field, Constraints, Default}`. The `field` form indicates the field is mandatory.

Note that when used with the router, only the second form makes sense, as it does not use the default and the field is always defined.

Constraints for each field are provided as an ordered list of atoms or funs to apply. Built-in constraints are provided as atoms, while custom constraints are provided as funs.

When multiple constraints are provided, they are applied in the order given. If the value has been modified by a constraint then the next one receives the new value.

For example, the following constraints will first validate and convert the field `my_value` to an integer, and then check that the integer is positive:

```
PositiveFun = fun
  (_, V) when V > 0 ->
    {ok, V};
  (_, _) ->
    {error, not_positive}
end,
{my_value, [int, PositiveFun]}.
```

We ignore the first fun argument in this snippet. We shouldn't. We will simply learn what it is later in this chapter.

When there's only one constraint, it can be provided directly without wrapping it into a list:

```
{my_value, int}
```

8.2 Built-in constraints

Built-in constraints are specified as an atom:

Constraint	Description
int	Converts binary value to integer.
nonempty	Ensures the binary value is non-empty.

8.3 Custom constraints

Custom constraints are specified as a fun. This fun takes two arguments. The first argument indicates the operation to be performed, and the second is the value. What the value is and what must be returned depends on the operation.

Cowboy currently defines three operations. The operation used for validating and converting user input is the `forward` operation.

```
int(forward, Value) ->
  try
    {ok, binary_to_integer(Value)}
  catch _:_ ->
    {error, not_an_integer}
end;
```

The value must be returned even if it is not converted by the constraint.

The `reverse` operation does the opposite: it takes a converted value and changes it back to what the user input would have been.

```
int(reverse, Value) ->
  try
    {ok, integer_to_binary(Value)}
  catch _:_ ->
    {error, not_an_integer}
end;
```

Finally, the `format_error` operation takes an error returned by any other operation and returns a formatted human-readable error message.

```
int(format_error, {not_an_integer, Value}) ->
  io_lib:format("The value ~p is not an integer.", [Value]).
```

Notice that for this case you get both the error and the value that was given to the constraint that produced this error.

Cowboy will not catch exceptions coming from constraint functions. They should be written to not emit any exceptions.

Part IV

Handlers

Chapter 9

Handlers

Handlers are Erlang modules that handle HTTP requests.

9.1 Plain HTTP handlers

The most basic handler in Cowboy implements the mandatory `init/2` callback, manipulates the request, optionally sends a response and then returns.

This callback receives the [Req object](#) and the initial state defined in the [router configuration](#).

A handler that does nothing would look like this:

```
init(Req, State) ->
    {ok, Req, State}.
```

Despite sending no reply, a 204 No Content response will be sent to the client, as Cowboy makes sure that a response is sent for every request.

We need to use the Req object to reply.

```
init(Req0, State) ->
    Req = cowboy_req:reply(200, #{
        <<"content-type">> => <<"text/plain">>
    }, <<"Hello World!">>, Req0),
    {ok, Req, State}.
```

Cowboy will immediately send a response when `cowboy:reply/4` is called.

We then return a 3-tuple. `ok` means that the handler ran successfully. We also give the modified Req back to Cowboy.

The last value of the tuple is a state that will be used in every subsequent callbacks to this handler. Plain HTTP handlers only have one additional callback, the optional and rarely used `terminate/3`.

9.2 Other handlers

The `init/2` callback can also be used to inform Cowboy that this is a different kind of handler and that Cowboy should switch to it. To do this you simply need to return the module name of the handler type you want to switch to.

Cowboy comes with three handler types you can switch to: [cowboy_rest](#), [cowboy_websocket](#) and [cowboy_loop](#). In addition to those you can define your own handler types.

Switching is simple. Instead of returning `ok`, you simply return the name of the handler type you want to use. The following snippet switches to a Websocket handler:

```
init(Req, State) ->
    {cowboy_websocket, Req, State}.
```

9.3 Cleaning up

All handler types provide the optional `terminate/3` callback.

```
terminate(_Reason, _Req, _State) ->
    ok.
```

This callback is strictly reserved for any required cleanup. You cannot send a response from this function. There is no other return value.

This callback is optional because it is rarely necessary. Cleanup should be done in separate processes directly (by monitoring the handler process to detect when it exits).

Cowboy does not reuse processes for different requests. The process will terminate soon after this call returns.

Chapter 10

Loop handlers

Loop handlers are a special kind of HTTP handlers used when the response can not be sent right away. The handler enters instead a receive loop waiting for the right message before it can send a response.

Loop handlers are used for requests where a response might not be immediately available, but where you would like to keep the connection open for a while in case the response arrives. The most known example of such practice is known as long polling.

Loop handlers can also be used for requests where a response is partially available and you need to stream the response body while the connection is open. The most known example of such practice is server-sent events.

While the same can be accomplished using plain HTTP handlers, it is recommended to use loop handlers because they are well-tested and allow using built-in features like hibernation and timeouts.

Loop handlers essentially wait for one or more Erlang messages and feed these messages to the `info/3` callback. It also features the `init/2` and `terminate/3` callbacks which work the same as for plain HTTP handlers.

10.1 Initialization

The `init/2` function must return a `cowboy_loop` tuple to enable loop handler behavior. This tuple may optionally contain a timeout value and/or the atom `hibernate` to make the process enter hibernation until a message is received.

This snippet enables the loop handler:

```
init(Req, State) ->
    {cowboy_loop, Req, State}.
```

This also makes the process hibernate:

```
init(Req, State) ->
    {cowboy_loop, Req, State, hibernate}.
```

10.2 Receive loop

Once initialized, Cowboy will wait for messages to arrive in the process' mailbox. When a message arrives, Cowboy calls the `info/3` function with the message, the `Req` object and the handler's state.

The following snippet sends a reply when it receives a `reply` message from another process, or waits for another message otherwise.

```
info({reply, Body}, Req, State) ->
    cowboy_req:reply(200, #{}, Body, Req),
    {stop, Req, State};
info(_Msg, Req, State) ->
    {ok, Req, State, hibernate}.
```

Do note that the `reply` tuple here may be any message and is simply an example.

This callback may perform any necessary operation including sending all or parts of a reply, and will subsequently return a tuple indicating if more messages are to be expected.

The callback may also choose to do nothing at all and just skip the message received.

If a reply is sent, then the `stop` tuple should be returned. This will instruct Cowboy to end the request.

Otherwise an `ok` tuple should be returned.

10.3 Streaming loop

Another common case well suited for loop handlers is streaming data received in the form of Erlang messages. This can be done by initiating a chunked reply in the `init/2` callback and then using `cowboy_req:chunk/2` every time a message is received.

The following snippet does exactly that. As you can see a chunk is sent every time an `event` message is received, and the loop is stopped by sending an `eof` message.

```
init(Req, State) ->
    Req2 = cowboy_req:stream_reply(200, Req),
    {cowboy_loop, Req2, State}.

info(eof, Req, State) ->
    {stop, Req, State};
info({event, Data}, Req, State) ->
    cowboy_req:stream_body(Data, nofin, Req),
    {ok, Req, State};
info(_Msg, Req, State) ->
    {ok, Req, State}.
```

10.4 Cleaning up

It is recommended that you set the connection header to `close` when replying, as this process may be reused for a subsequent request.

Please refer to the [Handlers chapter](#) for general instructions about cleaning up.

10.5 Hibernate

To save memory, you may hibernate the process in between messages received. This is done by returning the atom `hibernate` as part of the `loop` tuple callbacks normally return. Just add the atom at the end and Cowboy will hibernate accordingly.

Chapter 11

Static files

Cowboy comes with a ready to use handler for serving static files. It is provided as a convenience for serving files during development.

For systems in production, consider using one of the many Content Distribution Network (CDN) available on the market, as they are the best solution for serving files.

The static handler can serve either one file or all files from a given directory. The etag generation and mime types can be configured.

11.1 Serve one file

You can use the static handler to serve one specific file from an application's private directory. This is particularly useful to serve an *index.html* file when the client requests the `/` path, for example. The path configured is relative to the given application's private directory.

The following rule will serve the file *static/index.html* from the application `my_app`'s `priv` directory whenever the path `/` is accessed:

```
("/", cowboy_static, {priv_file, my_app, "static/index.html"})
```

You can also specify the absolute path to a file, or the path to the file relative to the current directory:

```
("/", cowboy_static, {file, "/var/www/index.html"})
```

11.2 Serve all files from a directory

You can also use the static handler to serve all files that can be found in the configured directory. The handler will use the `path_info` information to resolve the file location, which means that your route must end with a `[...]` pattern for it to work. All files are served, including the ones that may be found in subfolders.

You can specify the directory relative to an application's private directory.

The following rule will serve any file found in the application `my_app`'s `priv` directory inside the `static/assets` folder whenever the requested path begins with `/assets/`:

```
"/assets/[...]", cowboy_static, {priv_dir, my_app, "static/assets"}
```

You can also specify the absolute path to the directory or set it relative to the current directory:

```
"/assets/[...]", cowboy_static, {dir, "/var/www/assets"}
```

11.3 Customize the mimetype detection

By default, Cowboy will attempt to recognize the mimetype of your static files by looking at the extension.

You can override the function that figures out the mimetype of the static files. It can be useful when Cowboy is missing a mimetype you need to handle, or when you want to reduce the list to make lookups faster. You can also give a hard-coded mimetype that will be used unconditionally.

Cowboy comes with two functions built-in. The default function only handles common file types used when building Web applications. The other function is an extensive list of hundreds of mimetypes that should cover almost any need you may have. You can of course create your own function.

To use the default function, you should not have to configure anything, as it is the default. If you insist, though, the following will do the job:

```
{"/assets/...", cowboy_static, {priv_dir, my_app, "static/assets",
    [{mimetypes, cow_mimetypes, web}]}}
```

As you can see, there is an optional field that may contain a list of less used options, like mimetypes or etag. All option types have this optional field.

To use the function that will detect almost any mimetype, the following configuration will do:

```
{"/assets/...", cowboy_static, {priv_dir, my_app, "static/assets",
    [{mimetypes, cow_mimetypes, all}]}}
```

You probably noticed the pattern by now. The configuration expects a module and a function name, so you can use any of your own functions instead:

```
{"/assets/...", cowboy_static, {priv_dir, my_app, "static/assets",
    [{mimetypes, Module, Function}]}}
```

The function that performs the mimetype detection receives a single argument that is the path to the file on disk. It is recommended to return the mimetype in tuple form, although a binary string is also allowed (but will require extra processing). If the function can't figure out the mimetype, then it should return {<<"application">>, <<"octet-stream">>, []}.

When the static handler fails to find the extension, it will send the file as application/octet-stream. A browser receiving such file will attempt to download it directly to disk.

Finally, the mimetype can be hard-coded for all files. This is especially useful in combination with the file and priv_file options as it avoids needless computation:

```
{"/", cowboy_static, {priv_file, my_app, "static/index.html",
    [{mimetypes, {<<"text">>, <<"html">>, []}]}}
```

11.4 Generate an etag

By default, the static handler will generate an etag header value based on the size and modified time. This solution can not be applied to all systems though. It would perform rather poorly over a cluster of nodes, for example, as the file metadata will vary from server to server, giving a different etag on each server.

You can however change the way the etag is calculated:

```
{"/assets/...", cowboy_static, {priv_dir, my_app, "static/assets",
    [{etag, Module, Function}]}}
```

This function will receive three arguments: the path to the file on disk, the size of the file and the last modification time. In a distributed setup, you would typically use the file path to retrieve an etag value that is identical across all your servers.

You can also completely disable etag handling:

```
{"/assets/...", cowboy_static, {priv_dir, my_app, "static/assets",
    [{etag, false}]}}
```

Part V

Request and response

Chapter 12

The Req object

The Req object is a variable used for obtaining information about a request, read its body or send a response.

It is not really an object in the object-oriented sense. It is a simple map that can be directly accessed or used when calling functions from the `cowboy_req` module.

The Req object is the subject of a few different chapters. In this chapter we will learn about the Req object and look at how to retrieve information about the request.

12.1 Direct access

The Req map contains a number of fields which are documented and can be accessed directly. They are the fields that have a direct mapping to HTTP: the request method; the HTTP version used; the effective URI components scheme, host, port, path and qs; the request headers; and the connection peer address and port.

Note that the `version` field can be used to determine whether a connection is using HTTP/2.

To access a field, you can simply match in the function head. The following example sends a simple "Hello world!" response when the method is GET, and a 405 error otherwise.

```
init(Req0=#{method := <<"GET">>}, State) ->
  Req = cowboy_req:reply(200, #{
    <<"content-type">> => <<"text/plain">>
  }, <<"Hello world!">>, Req0),
  {ok, Req, State};
init(Req0, State) ->
  Req = cowboy_req:reply(405, #{
    <<"allow">> => <<"GET">>
  }, Req0),
  {ok, Req, State}.
```

Any other field is internal and should not be accessed. They may change in future releases, including maintenance releases, without notice.

Modifying the Req object, while allowed, is not recommended unless strictly necessary. If adding new fields, make sure to namespace the field names so that no conflict can occur with future Cowboy updates or third party projects.

12.2 Introduction to the cowboy_req interface

Functions in the `cowboy_req` module provide access to the request information but also various operations that are common when dealing with HTTP requests.

All the functions that begin with a verb indicate an action. Other functions simply return the corresponding value (sometimes that value does need to be built, but the cost of the operation is equivalent to retrieving a value).

Some of the `cowboy_req` functions return an updated Req object. They are the read, reply, set and delete functions. While ignoring the returned Req will not cause incorrect behavior for some of them, it is highly recommended to always keep and use the last returned Req object. The manual for `cowboy_req` details these functions and what modifications are done to the Req object.

Some of the calls to `cowboy_req` have side effects. This is the case of the read and reply functions. Cowboy reads the request body or replies immediately when the function is called.

All functions will crash if something goes wrong. There is usually no need to catch these errors, Cowboy will send the appropriate 4xx or 5xx response depending on where the crash occurred.

12.3 Request method

The request method can be retrieved directly:

```
#{method := Method} = Req.
```

Or using a function:

```
Method = cowboy_req:method(Req).
```

The method is a case sensitive binary string. Standard methods include GET, HEAD, OPTIONS, PATCH, POST, PUT or DELETE.

12.4 HTTP version

The HTTP version is informational. It does not indicate that the client implements the protocol well or fully.

There is typically no need to change behavior based on the HTTP version: Cowboy already does it for you.

It can be useful in some cases, though. For example, one may want to redirect HTTP/1.1 clients to use Websocket, while HTTP/2 clients keep using HTTP/2.

The HTTP version can be retrieved directly:

```
#{version := Version} = Req.
```

Or using a function:

```
Version = cowboy_req:version(Req).
```

Cowboy defines the 'HTTP/1.0', 'HTTP/1.1' and 'HTTP/2' versions. Custom protocols can define their own values as atoms.

12.5 Effective request URI

The scheme, host, port, path and query string components of the effective request URI can all be retrieved directly:

```
#{
  scheme := Scheme,
  host := Host,
  port := Port,
  path := Path,
  qs := Qs
} = Req.
```

Or using the related functions:

```
Scheme = cowboy_req:scheme(Req),
Host = cowboy_req:host(Req),
Port = cowboy_req:port(Req),
Path = cowboy_req:path(Req).
Qs = cowboy_req:qs(Req).
```

The scheme and host are lowercased case insensitive binary strings. The port is an integer representing the port number. The path and query string are case sensitive binary strings.

Cowboy defines only the `<<"http">>` and `<<"https">>` schemes. They are chosen so that the scheme will only be `<<"https">>` for requests on secure HTTP/1.1 or HTTP/2 connections.

The effective request URI itself can be reconstructed with the `cowboy_req:uri/1, 2` function. By default, an absolute URI is returned:

```
%% scheme://host[:port]/path[?qs]
URI = cowboy_req:uri(Req).
```

Options are available to either disable or replace some or all of the components. Various URIs or URI formats can be generated this way, including the origin form:

```
%% /path[?qs]
URI = cowboy_req:uri(Req, #{host => undefined}).
```

The protocol relative form:

```
%% //host[:port]/path[?qs]
URI = cowboy_req:uri(Req, #{scheme => undefined}).
```

The absolute URI without a query string:

```
URI = cowboy_req:uri(Req, #{qs => undefined}).
```

A different host:

```
URI = cowboy_req:uri(Req, #{host => <<"example.org">>}).
```

And any other combination.

12.6 Bindings

Bindings are the host and path components that you chose to extract when defining the routes of your application. They are only available after the routing.

Cowboy provides functions to retrieve one or all bindings.

To retrieve a single value:

```
Value = cowboy_req:binding(userid, Req).
```

When attempting to retrieve a value that was not bound, `undefined` will be returned. A different default value can be provided:

```
Value = cowboy_req:binding(userid, Req, 42).
```

To retrieve everything that was bound:

```
Bindings = cowboy_req:bindings(Req).
```

They are returned as a map, with keys being atoms.

The Cowboy router also allows you to capture many host or path segments at once using the `...` qualifier.

To retrieve the segments captured from the host name:

```
HostInfo = cowboy_req:host_info(Req).
```

And the path segments:

```
PathInfo = cowboy_req:path_info(Req).
```

Cowboy will return `undefined` if `...` was not used in the route.

12.7 Query parameters

Cowboy provides two functions to access query parameters. You can use the first to get the entire list of parameters.

```
QsVals = cowboy_req:parse_qs(Req),  
{_, Lang} = lists:keyfind(<<"lang">>, 1, QsVals).
```

Cowboy will only parse the query string, and not do any transformation. This function may therefore return duplicates, or parameter names without an associated value. The order of the list returned is undefined.

When a query string is `key=1&key=2`, the list returned will contain two parameters of name `key`.

The same is true when trying to use the PHP-style suffix `[]`. When a query string is `key[]=1&key[]=2`, the list returned will contain two parameters of name `key[]`.

When a query string is simply `key`, Cowboy will return the list `[<<"key">>, true]`, using `true` to indicate that the parameter `key` was defined, but with no value.

The second function Cowboy provides allows you to match out only the parameters you are interested in, and at the same time do any post processing you require using [constraints](#). This function returns a map.

```
#{id := ID, lang := Lang} = cowboy_req:match_qs([id, lang], Req).
```

Constraints can be applied automatically. The following snippet will crash when the `id` parameter is not an integer, or when the `lang` parameter is empty. At the same time, the value for `id` will be converted to an integer term:

```
QsMap = cowboy_req:match_qs([{id, int}, {lang, nonempty}], Req).
```

A default value may also be provided. The default will be used if the `lang` key is not found. It will not be used if the key is found but has an empty value.

```
#{lang := Lang} = cowboy_req:match_qs([{lang, [], <<"en-US">>}], Req).
```

If no default is provided and the value is missing, the query string is deemed invalid and the process will crash.

When the query string is `key=1&key=2`, the value for `key` will be the list `[1, 2]`. Parameter names do not need to include the PHP-style suffix. Constraints may be used to ensure that only one value was passed through.

12.8 Headers

Header values can be retrieved either as a binary string or parsed into a more meaningful representation.

The get the raw value:

```
HeaderVal = cowboy_req:header(<<"content-type">>, Req).
```

Cowboy expects all header names to be provided as lowercase binary strings. This is true for both requests and responses, regardless of the underlying protocol.

When the header is missing from the request, `undefined` will be returned. A different default can be provided:

```
HeaderVal = cowboy_req:header(<<"content-type">>, Req, <<"text/plain">>).
```

All headers can be retrieved at once, either directly:

```
#{headers := AllHeaders} = Req.
```

Or using a function:

```
AllHeaders = cowboy_req:headers(Req).
```

Cowboy provides equivalent functions to parse individual headers. There is no function to parse all headers at once.

To parse a specific header:

```
ParsedVal = cowboy_req:parse_header(<<"content-type">>, Req).
```

An exception will be thrown if it doesn't know how to parse the given header, or if the value is invalid. The list of known headers and default values can be found in the manual.

When the header is missing, `undefined` is returned. You can change the default value. Note that it should be the parsed value directly:

```
ParsedVal = cowboy_req:parse_header(<<"content-type">>, Req,  
  {<<"text">>, <<"plain">>, []}).
```

12.9 Peer

The peer address and port number for the connection can be retrieved either directly or using a function.

To retrieve the peer directly:

```
#{peer := {IP, Port}} = Req.
```

And using a function:

```
{IP, Port} = cowboy_req:peer(Req).
```

Note that the peer corresponds to the remote end of the connection to the server, which may or may not be the client itself. It may also be a proxy or a gateway.

Chapter 13

Reading the request body

The request body can be read using the `Req` object.

Cowboy will not attempt to read the body until requested. You need to call the body reading functions in order to retrieve it.

Cowboy will not cache the body, it is therefore only possible to read it once.

You are not required to read it, however. If a body is present and was not read, Cowboy will either cancel or skip its download, depending on the protocol.

Cowboy provides functions for reading the body raw, and read and parse form urlencoded or [multipart bodies](#). The latter is covered in its own chapter.

13.1 Request body presence

Not all requests come with a body. You can check for the presence of a request body with this function:

```
cowboy_req:has_body(Req) .
```

It returns `true` if there is a body; `false` otherwise.

In practice, this function is rarely used. When the method is `POST`, `PUT` or `PATCH`, the request body is often required by the application, which should just attempt to read it directly.

13.2 Request body length

You can obtain the length of the body:

```
Length = cowboy_req:body_length(Req) .
```

Note that the length may not be known in advance. In that case `undefined` will be returned. This can happen with HTTP/1.1's chunked transfer-encoding, or HTTP/2 when no content-length was provided.

Cowboy will update the body length in the `Req` object once the body has been read completely. A length will always be returned when attempting to call this function after reading the body completely.

13.3 Reading the body

You can read the entire body with one function call:

```
{ok, Data, Req} = cowboy_req:read_body(Req0) .
```

Cowboy returns an `ok` tuple when the body has been read fully.

By default, Cowboy will attempt to read up to 8MB of data, for up to 15 seconds. The call will return once Cowboy has read at least 8MB of data, or at the end of the 15 seconds period.

These values can be customized. For example, to read only up to 1MB for up to 5 seconds:

```
{ok, Data, Req} = cowboy_req:read_body(Req0,  
  #{length => 1000000, period => 5000}).
```

You may also disable the length limit:

```
{ok, Data, Req} = cowboy_req:read_body(Req0, #{length => infinity}).
```

This makes the function wait 15 seconds and return with whatever arrived during that period. This is not recommended for public facing applications.

These two options can effectively be used to control the rate of transmission of the request body.

13.4 Streaming the body

When the body is too large, the first call will return a `more` tuple instead of `ok`. You can call the function again to read more of the body, reading it one chunk at a time.

```
read_body_to_console(Req0) ->  
  case cowboy_req:read_body(Req0) of  
    {ok, Data, Req} ->  
      io:format("~s", [Data]),  
      Req;  
    {more, Data, Req} ->  
      io:format("~s", [Data]),  
      read_body_to_console(Req)  
  end.
```

The `length` and `period` options can also be used. They need to be passed for every call.

13.5 Reading a form urlencoded body

Cowboy provides a convenient function for reading and parsing bodies sent as `application/x-www-form-urlencoded`.

```
{ok, KeyValues, Req} = cowboy_req:read_urlencoded_body(Req0).
```

This function returns a list of key/values, exactly like the function `cowboy_req:parse_qs/1`.

The defaults for this function are different. Cowboy will read for up to 64KB and up to 5 seconds. They can be modified:

```
{ok, KeyValues, Req} = cowboy_req:read_urlencoded_body(Req0,  
  #{length => 4096, period => 3000}).
```

Chapter 14

Sending a response

The response must be sent using the Req object.

Cowboy provides two different ways of sending responses: either directly or by streaming the body. Response headers and body may be set in advance. The response is sent as soon as one of the reply or stream reply function is called.

Cowboy also provides a simplified interface for sending files. It can also send only specific parts of a file.

While only one response is allowed for every request, HTTP/2 introduced a mechanism that allows the server to push additional resources related to the response. This chapter also describes how this feature works in Cowboy.

14.1 Reply

Cowboy provides three functions for sending the entire reply, depending on whether you need to set headers and body. In all cases, Cowboy will add any headers required by the protocol (for example the date header will always be sent).

When you need to set only the status code, use `cowboy_req:reply/2`:

```
Req = cowboy_req:reply(200, Req0).
```

When you need to set response headers at the same time, use `cowboy_req:reply/3`:

```
Req = cowboy_req:reply(303, #{  
  <<"location">> => <<"https://ninenines.eu">>  
}, Req0).
```

Note that the header name must always be a lowercase binary.

When you also need to set the response body, use `cowboy_req:reply/4`:

```
Req = cowboy_req:reply(200, #{  
  <<"content-type">> => <<"text/plain">>  
}, "Hello world!", Req0).
```

You should always set the content-type header when the response has a body. There is however no need to set the content-length header; Cowboy does it automatically.

The response body and the header values must be either a binary or an iolist. An iolist is a list containing binaries, characters, strings or other iolists. This allows you to build a response from different parts without having to do any concatenation:

```
Title = "Hello world!",  
Body = <<"Hats off!">>,  
Req = cowboy_req:reply(200, #{  
  <<"content-type">> => <<"text/html">>  
}, ["<html><head><title>", Title, "</title></head>",  
  "<body><p>", Body, "</p></body></html>"], Req0).
```

This method of building responses is more efficient than concatenating. Behind the scenes, each element of the list is simply a pointer, and those pointers are used directly when writing to the socket.

14.2 Stream reply

Cowboy provides two functions for initiating a response, and an additional function for streaming the response body. Cowboy will add any required headers to the response.

When you need to set only the status code, use `cowboy_req:stream_reply/2`:

```
Req = cowboy_req:stream_reply(200, Req0),

cowboy_req:stream_body("Hello...", nofin, Req),
cowboy_req:stream_body("chunked...", nofin, Req),
cowboy_req:stream_body("world!!", fin, Req).
```

The second argument to `cowboy_req:stream_body/3` indicates whether this data terminates the body. Use `fin` for the final flag, and `nofin` otherwise.

This snippet does not set a content-type header. This is not recommended. All responses with a body should have a content-type. The header can be set beforehand, or using the `cowboy_req:stream_reply/3`:

```
Req = cowboy_req:stream_reply(200, #{
  <<"content-type">> => <<"text/html">>
}, Req0),

cowboy_req:stream_body("<html><head>Hello world!</head>", nofin, Req),
cowboy_req:stream_body("<body><p>Hats off!</p></body></html>", fin, Req).
```

HTTP provides a few different ways to stream response bodies. Cowboy will select the most appropriate one based on the HTTP version and the request and response headers.

While not required by any means, it is recommended that you set the content-length header in the response if you know it in advance. This will ensure that the best response method is selected and help clients understand when the response is fully received.

14.3 Preset response headers

Cowboy provides functions to set response headers without immediately sending them. They are stored in the Req object and sent as part of the response when a reply function is called.

To set response headers:

```
Req = cowboy_req:set_resp_header(<<"allow">>, "GET", Req0).
```

Header names must be a lowercase binary.

Do not use this function for setting cookies. Refer to the [Cookies](#) chapter for more information.

To check if a response header has already been set:

```
cowboy_req:has_resp_header(<<"allow">>, Req).
```

It returns `true` if the header was set, `false` otherwise.

To delete a response header that was set previously:

```
Req = cowboy_req:delete_resp_header(<<"allow">>, Req0).
```

14.4 Overriding headers

As Cowboy provides different ways of setting response headers and body, clashes may occur, so it's important to understand what happens when a header is set twice.

Headers come from five different origins:

- Protocol-specific headers (for example HTTP/1.1's connection header)
- Other required headers (for example the date header)
- Preset headers
- Headers given to the reply function
- Set-cookie headers

Cowboy does not allow overriding protocol-specific headers.

Set-cookie headers will always be appended at the end of the list of headers before sending the response.

Headers given to the reply function will always override preset headers and required headers. If a header is found in two or three of these, then the one in the reply function is picked and the others are dropped.

Similarly, preset headers will always override required headers.

To illustrate, look at the following snippet. Cowboy by default sends the server header with the value "Cowboy". We can override it:

```
Req = cowboy_req:reply(200, #{  
    <<"server">> => <<"yaws">>  
}, Req0).
```

14.5 Preset response body

Cowboy provides functions to set the response body without immediately sending it. It is stored in the Req object and sent when the reply function is called.

To set the response body:

```
Req = cowboy_req:set_resp_body("Hello world!", Req0).
```

To check if a response body has already been set:

```
cowboy_req:has_resp_body(Req).
```

It returns `true` if the body was set and is non-empty, `false` otherwise.

The preset response body is only sent if the reply function used is `cowboy_req:reply/2` or `cowboy_req:reply/3`.

14.6 Sending files

Cowboy provides a shortcut for sending files. When using `cowboy_req:reply/4`, or when presetting the response header, you can give a `sendfile` tuple to Cowboy:

```
{sendfile, Offset, Length, Filename}
```

Depending on the values for `Offset` or `Length`, the entire file may be sent, or just a part of it.

The length is required even for sending the entire file. Cowboy sends it in the content-length header.

To send a file while replying:

```
Req = cowboy_req:reply(200, #{
  <<"content-type">> => "image/png"
}, {sendfile, 0, 12345, "path/to/logo.png"}, Req0).
```

14.7 Push

The HTTP/2 protocol introduced the ability to push resources related to the one sent in the response. Cowboy provides two functions for that purpose: `cowboy_req:push/3, 4`.

Push is only available for HTTP/2. Cowboy will automatically ignore push requests if the protocol doesn't support it.

The push function must be called before any of the reply functions. Doing otherwise will result in a crash.

To push a resource, you need to provide the same information as a client performing a request would. This includes the HTTP method, the URI and any necessary request headers.

Cowboy by default only requires you to give the path to the resource and the request headers. The rest of the URI is taken from the current request (excluding the query string, set to empty) and the method is GET by default.

The following snippet pushes a CSS file that is linked to in the response:

```
cowboy_req:push("/static/style.css", #{
  <<"accept">> => <<"text/css">>
}, Req0),
Req = cowboy_req:reply(200, #{
  <<"content-type">> => <<"text/html">>
}, ["<html><head><title>My web page</title>",
  "<link rel='stylesheet' type='text/css' href='/static/style.css'>",
  "<body><p>Welcome to Erlang!</p></body></html>"], Req0).
```

To override the method, scheme, host, port or query string, simply pass in a fourth argument. The following snippet uses a different host name:

```
cowboy_req:push("/static/style.css", #{
  <<"accept">> => <<"text/css">>
}, #{host => <<"cdn.example.org">>}, Req),
```

Pushed resources don't have to be files. As long as the push request is cacheable, safe and does not include a body, the resource can be pushed.

Under the hood, Cowboy handles pushed requests the same as normal requests: a different process is created which will ultimately send a response to the client.

Chapter 15

Using cookies

Cookies are a mechanism allowing applications to maintain state on top of the stateless HTTP protocol.

Cookies are a name/value store where the names and values are stored in plain text. They expire either after a delay or when the browser closes. They can be configured on a specific domain name or path, and restricted to secure resources (sent or downloaded over HTTPS), or restricted to the server (disallowing access from client-side scripts).

Cookie names are de facto case sensitive.

Cookies are stored client-side and sent with every subsequent request that matches the domain and path for which they were stored, until they expire. This can create a non-negligible cost.

Cookies should not be considered secure. They are stored on the user's computer in plain text, and can be read by any program. They can also be read by proxies when using clear connections. Always validate the value before using it, and never store any sensitive information inside it.

Cookies set by the server are only available in requests following the client reception of the response containing them.

Cookies may be sent repeatedly. This is often useful to update the expiration time and avoid losing a cookie.

15.1 Setting cookies

By default cookies are defined for the duration of the session:

```
SessionID = generate_session_id(),
Req = cowboy_req:set_resp_cookie(<<"sessionid">>, SessionID, Req0).
```

They can also be set for a duration in seconds:

```
SessionID = generate_session_id(),
Req = cowboy_req:set_resp_cookie(<<"sessionid">>, SessionID, Req0,
    #{max_age => 3600}).
```

To delete cookies, set max_age to 0:

```
SessionID = generate_session_id(),
Req = cowboy_req:set_resp_cookie(<<"sessionid">>, SessionID, Req0,
    #{max_age => 0}).
```

To restrict cookies to a specific domain and path, the options of the same name can be used:

```
Req = cowboy_req:set_resp_cookie(<<"inaccount">>, <<"1">>, Req0,
    #{domain => "my.example.org", path => "/account"}).
```

Cookies will be sent with requests to this domain and all its subdomains, and to resources on this path or deeper in the path hierarchy.

To restrict cookies to secure channels (typically resources available over HTTPS):

```
SessionID = generate_session_id(),
Req = cowboy_req:set_resp_cookie(<<"sessionid">>, SessionID, Req0,
    #{secure => true}).
```

To prevent client-side scripts from accessing a cookie:

```
SessionID = generate_session_id(),
Req = cowboy_req:set_resp_cookie(<<"sessionid">>, SessionID, Req0,
    #{http_only => true}).
```

Cookies may also be set client-side, for example using Javascript.

15.2 Reading cookies

The client only ever sends back the cookie name and value. All other options that can be set are never sent back.

Cowboy provides two functions for reading cookies. Both involve parsing the cookie header(s) and so should not be called repeatedly.

You can get all cookies as a key/value list:

```
Cookies = cowboy_req:parse_cookies(Req),
{_, Lang} = lists:keyfind(<<"lang">>, 1, Cookies).
```

Or you can perform a match against cookies and retrieve only the ones you need, while at the same time doing any required post processing using [constraints](#). This function returns a map:

```
#{id := ID, lang := Lang} = cowboy_req:match_cookies([id, lang], Req).
```

You can use constraints to validate the values while matching them. The following snippet will crash if the `id` cookie is not an integer number or if the `lang` cookie is empty. Additionally the `id` cookie value will be converted to an integer term:

```
CookiesMap = cowboy_req:match_cookies([id, int], [lang, nonempty]), Req).
```

Note that if two cookies share the same name, then the map value will be a list of the two cookie values.

A default value can be provided. The default will be used if the `lang` cookie is not found. It will not be used if the cookie is found but has an empty value:

```
#{lang := Lang} = cowboy_req:match_cookies([lang, [], <<"en-US">>]), Req).
```

If no default is provided and the value is missing, an exception is thrown.

Chapter 16

Multipart requests

Multipart originates from MIME, an Internet standard that extends the format of emails.

A multipart message is a list of parts. A part contains headers and a body. The body of the parts may be of any media type, and contain text or binary data. It is possible for parts to contain a multipart media type.

In the context of HTTP, multipart is most often used with the `multipart/form-data` media type. It is what browsers use to upload files through HTML forms.

The `multipart/byteranges` is also common. It is the media type used to send arbitrary bytes from a resource, enabling clients to resume downloads.

16.1 Form-data

In the normal case, when a form is submitted, the browser will use the `application/x-www-form-urlencoded` content-type. This type is just a list of keys and values and is therefore not fit for uploading files.

That's where the `multipart/form-data` content-type comes in. When the form is configured to use this content-type, the browser will create a multipart message where each part corresponds to a field on the form. For files, it also adds some metadata in the part headers, like the file name.

A form with a text input, a file input and a select choice box will result in a multipart message with three parts, one for each field.

The browser does its best to determine the media type of the files it sends this way, but you should not rely on it for determining the contents of the file. Proper investigation of the contents is recommended.

16.2 Checking for multipart messages

The content-type header indicates the presence of a multipart message:

```
{<<"multipart">>, <<"form-data">>, _}  
  = cowboy_req:parse_header(<<"content-type">>, Req) .
```

16.3 Reading a multipart message

Cowboy provides two sets of functions for reading request bodies as multipart messages.

The `cowboy_req:read_part/1,2` functions return the next part's headers, if any.

The `cowboy_req:read_part_body/1,2` functions return the current part's body. For large bodies you may need to call the function multiple times.

To read a multipart message you need to iterate over all its parts:

```
multipart(Req0) ->
  case cowboy_req:read_part(Req0) of
    {ok, _Headers, Req1} ->
      {ok, _Body, Req} = cowboy_req:read_part_body(Req1),
      multipart(Req);
    {done, Req} ->
      Req
  end.
```

When part bodies are too large, Cowboy will return a more tuple, and allow you to loop until the part body has been fully read.

The function `cow_multipart:form_data/1` can be used to quickly obtain information about a part from a `multipart/form-data` message. The function returns a data or a file tuple depending on whether this is a normal field or a file being uploaded.

The following snippet will use this function and use different strategies depending on whether the part is a file:

```
multipart(Req0) ->
  case cowboy_req:read_part(Req0) of
    {ok, Headers, Req1} ->
      Req = case cow_multipart:form_data(Headers) of
        {data, _FieldName} ->
          {ok, _Body, Req2} = cowboy_req:read_part_body(Req1),
          Req2;
        {file, _FieldName, _Filename, _CType} ->
          stream_file(Req1)
      end,
      multipart(Req);
    {done, Req} ->
      Req
  end.

stream_file(Req0) ->
  case cowboy_req:read_part_body(Req0) of
    {ok, _LastBodyChunk, Req} ->
      Req;
    {more, _BodyChunk, Req} ->
      stream_file(Req)
  end.
```

Both the part header and body reading functions can take options that will be given to the request body reading functions. By default, `cowboy_req:read_part/1` reads up to 64KB for up to 5 seconds. `cowboy_req:read_part_body/1` has the same defaults as `cowboy_req:read_body/1`.

To change the defaults for part headers:

```
cowboy_req:read_part(Req, #{length => 128000}).
```

And for part bodies:

```
cowboy_req:read_part_body(Req, #{length => 1000000, period => 7000}).
```

16.4 Skipping unwanted parts

Part bodies do not have to be read. Cowboy will automatically skip it when you request the next part's body.

The following snippet reads all part headers and skips all bodies:

```
multipart(Req0) ->
  case cowboy_req:read_part(Req0) of
    {ok, _Headers, Req} ->
```

```
        multipart(Req);  
    {done, Req} ->  
        Req  
end.
```

Similarly, if you start reading the body and it ends up being too big, you can simply continue with the next part. Cowboy will automatically skip what remains.

While Cowboy can skip part bodies automatically, the read rate is not configurable. Depending on your application you may want to skip manually, in particular if you observe poor performance while skipping.

You do not have to read all parts either. You can stop reading as soon as you find the data you need.

Part VI

REST

Chapter 17

REST principles

This chapter will attempt to define the concepts behind REST and explain what makes a service RESTful.

REST is often confused with performing a distinct operation depending on the HTTP method, while using more than the GET and POST methods. That's highly misguided at best.

We will first attempt to define REST and will look at what it means in the context of HTTP and the Web. For a more in-depth explanation of REST, you can read [Roy T. Fielding's dissertation](#) as it does a great job explaining where it comes from and what it achieves.

17.1 REST architecture

REST is a **client-server** architecture. The client and the server both have a different set of concerns. The server stores and/or manipulates information and makes it available to the user in an efficient manner. The client takes that information and displays it to the user and/or uses it to perform subsequent requests for information. This separation of concerns allows both the client and the server to evolve independently as it only requires that the interface stays the same.

REST is **stateless**. That means the communication between the client and the server always contains all the information needed to perform the request. There is no session state in the server, it is kept entirely on the client's side. If access to a resource requires authentication, then the client needs to authenticate itself with every request.

REST is **cacheable**. The client, the server and any intermediary components can all cache resources in order to improve performance.

REST provides a **uniform interface** between components. This simplifies the architecture, as all components follow the same rules to speak to one another. It also makes it easier to understand the interactions between the different components of the system. A number of constraints are required to achieve this. They are covered in the rest of the chapter.

REST is a **layered system**. Individual components cannot see beyond the immediate layer with which they are interacting. This means that a client connecting to an intermediate component, like a proxy, has no knowledge of what lies beyond. This allows components to be independent and thus easily replaceable or extendable.

REST optionally provides **code on demand**. Code may be downloaded to extend client functionality. This is optional however because the client may not be able to download or run this code, and so a REST component cannot rely on it being executed.

17.2 Resources and resource identifiers

A resource is an abstract concept. In a REST system, any information that can be named may be a resource. This includes documents, images, a collection of resources and any other information. Any information that can be the target of a hypertext link can be a resource.

A resource is a conceptual mapping to a set of entities. The set of entities evolves over time; a resource doesn't. For example, a resource can map to "users who have logged in this past month" and another to "all users". At some point in time they may map

to the same set of entities, because all users logged in this past month. But they are still different resources. Similarly, if nobody logged in recently, then the first resource may map to the empty set. This resource exists regardless of the information it maps to. Resources are identified by uniform resource identifiers, also known as URIs. Sometimes internationalized resource identifiers, or IRIs, may also be used, but these can be directly translated into a URI.

In practice we will identify two kinds of resources. Individual resources map to a set of one element, for example "user Joe". Collection of resources map to a set of 0 to N elements, for example "all users".

17.3 Resource representations

The representation of a resource is a sequence of bytes associated with metadata.

The metadata comes as a list of key-value pairs, where the name corresponds to a standard that defines the value's structure and semantics. With HTTP, the metadata comes in the form of request or response headers. The headers' structure and semantics are well defined in the HTTP standard. Metadata includes representation metadata, resource metadata and control data.

The representation metadata gives information about the representation, such as its media type, the date of last modification, or even a checksum.

Resource metadata could be link to related resources or information about additional representations of the resource.

Control data allows parameterizing the request or response. For example, we may only want the representation returned if it is more recent than the one we have in cache. Similarly, we may want to instruct the client about how it should cache the representation. This isn't restricted to caching. We may, for example, want to store a new representation of a resource only if it wasn't modified since we first retrieved it.

The data format of a representation is also known as the media type. Some media types are intended for direct rendering to the user, while others are intended for automated processing. The media type is a key component of the REST architecture.

17.4 Self-descriptive messages

Messages must be self-descriptive. That means that the data format of a representation must always come with its media type (and similarly requesting a resource involves choosing the media type of the representation returned). If you are sending HTML, then you must say it is HTML by sending the media type with the representation. In HTTP this is done using the content-type header.

The media type is often an IANA registered media type, like `text/html` or `image/png`, but does not need to be. Exactly two things are important for respecting this constraint: that the media type is well specified, and that the sender and recipient agree about what the media type refers to.

This means that you can create your own media types, like `application/x-mine`, and that as long as you write the specifications for it and that both endpoints agree about it then the constraint is respected.

17.5 Hypermedia as the engine of application state

The last constraint is generally where services that claim to be RESTful fail. Interactions with a server must be entirely driven by hypermedia. The client does not need any prior knowledge of the service in order to use it, other than an entry point and of course basic understanding of the media type of the representations, at the very least enough to find and identify hyperlinks and link relations.

To give a simple example, if your service only works with the `application/json` media type then this constraint cannot be respected (as there are no concept of links in JSON) and thus your service isn't RESTful. This is the case for the majority of self-proclaimed REST services.

On the other hand if you create a JSON based media type that has a concept of links and link relations, then your service might be RESTful.

Respecting this constraint means that the entirety of the service becomes self-discoverable, not only the resources in it, but also the operations you can perform on it. This makes clients very thin as there is no need to implement anything specific to the service to operate on it.

Chapter 18

REST handlers

REST is implemented in Cowboy as a sub protocol. The request is handled as a state machine with many optional callbacks describing the resource and modifying the machine's behavior.

The REST handler is the recommended way to handle HTTP requests.

18.1 Initialization

First, the `init/2` callback is called. This callback is common to all handlers. To use REST for the current request, this function must return a `cowboy_rest` tuple.

```
init(Req, State) ->
    {cowboy_rest, Req, State}.
```

Cowboy will then switch to the REST protocol and start executing the state machine.

After reaching the end of the flowchart, the `terminate/3` callback will be called if it is defined.

18.2 Methods

The REST component has code for handling the following HTTP methods: HEAD, GET, POST, PATCH, PUT, DELETE and OPTIONS.

Other methods can be accepted, however they have no specific callback defined for them at this time.

18.3 Callbacks

All callbacks are optional. Some may become mandatory depending on what other defined callbacks return. The various flowcharts in the next chapter should be a useful to determine which callbacks you need.

All callbacks take two arguments, the `Req` object and the `State`, and return a three-element tuple of the form `{Value, Req, State}`.

All callbacks can also return `{stop, Req, State}` to stop execution of the request.

The following table summarizes the callbacks and their default values. If the callback isn't defined, then the default value will be used. Please look at the flowcharts to find out the result of each return value.

In the following table, "skip" means the callback is entirely skipped if it is undefined, moving directly to the next step. Similarly, "none" means there is no default value for this callback.

Callback name	Default value
allowed_methods	[<<"GET">>, <<"HEAD">>, <<"OPTIONS">>]
allow_missing_post	true
charsets_provided	skip
content_types_accepted	none
content_types_provided	[{{ <<"text">>, <<"html">>, '*' }, to_html}]
delete_completed	true
delete_resource	false
expires	undefined
forbidden	false
generate_etag	undefined
is_authorized	true
is_conflict	false
known_methods	[<<"GET">>, <<"HEAD">>, <<"POST">>, <<"PUT">>, <<"PATCH">>, <<"DELETE">>, <<"OPTIONS">>]
languages_provided	skip
last_modified	undefined
malformed_request	false
moved_permanently	false
moved_temporarily	false
multiple_choices	false
options	ok
previously_existed	false
resource_exists	true
service_available	true
uri_too_long	false
valid_content_headers	true
valid_entity_length	true
variances	[]

As you can see, Cowboy tries to move on with the request whenever possible by using well thought out default values.

In addition to these, there can be any number of user-defined callbacks that are specified through `content_types_accepted/2` and `content_types_provided/2`. They can take any name, however it is recommended to use a separate prefix for the callbacks of each function. For example, `from_html` and `to_html` indicate in the first case that we're accepting a resource given as HTML, and in the second case that we send one as HTML.

18.4 Meta data

Cowboy will set informative values to the Req object at various points of the execution. You can retrieve them by matching the Req object directly. The values are defined in the following table:

Key	Details
media_type	The content-type negotiated for the response entity.
language	The language negotiated for the response entity.
charset	The charset negotiated for the response entity.

They can be used to send a proper body with the response to a request that used a method other than HEAD or GET.

18.5 Response headers

Cowboy will set response headers automatically over the execution of the REST code. They are listed in the following table.

Header name	Details
content-language	Language used in the response body
content-type	Media type and charset of the response body
etag	Etag of the resource
expires	Expiration date of the resource
last-modified	Last modification date for the resource
location	Relative or absolute URI to the requested resource
vary	List of headers that may change the representation of the resource

Chapter 19

REST flowcharts

This chapter will explain the REST handler state machine through a number of different diagrams.

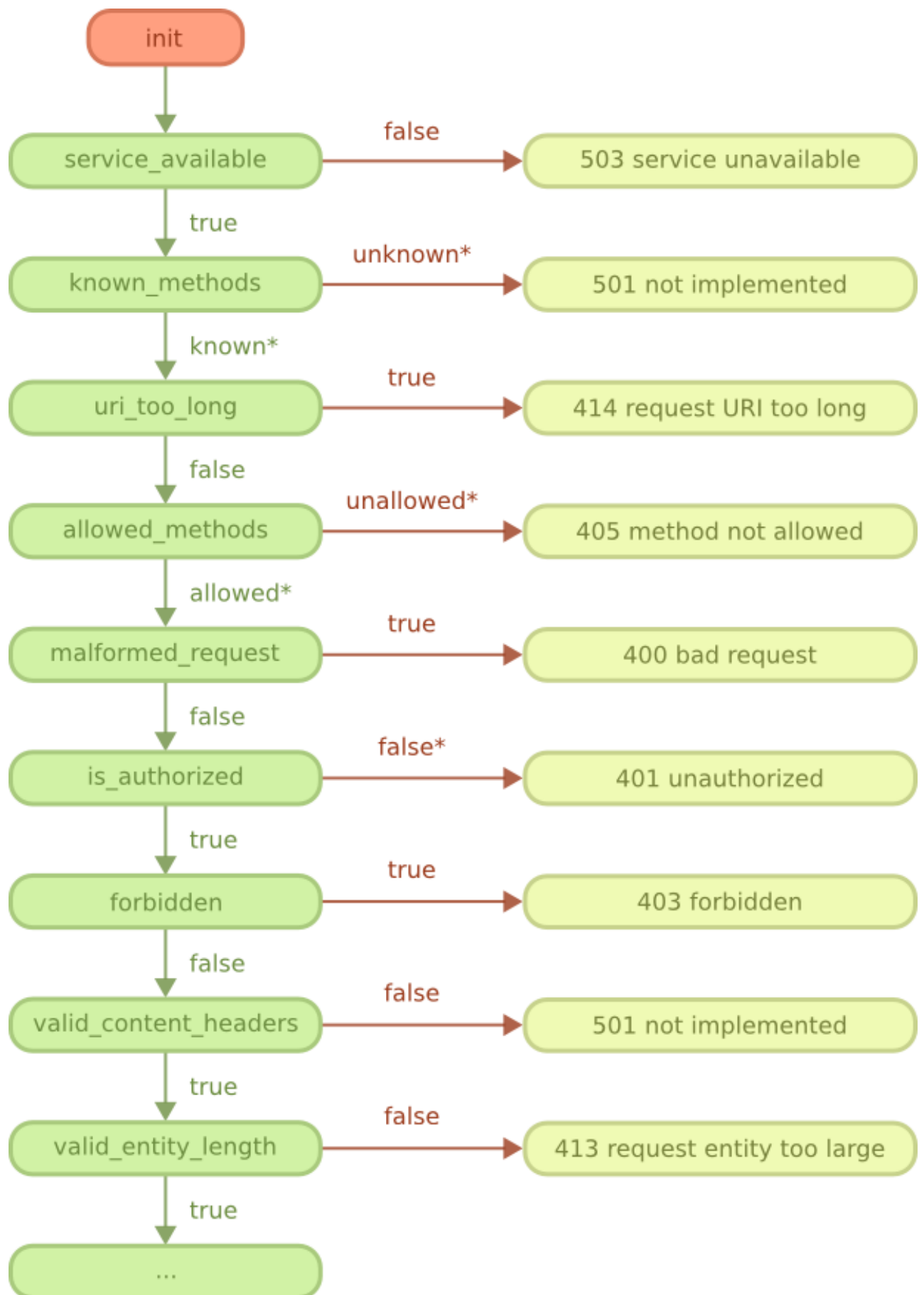
There are four main paths that requests may follow. One for the method OPTIONS; one for the methods GET and HEAD; one for the methods PUT, POST and PATCH; and one for the method DELETE.

All paths start with the "Start" diagram, and all paths excluding the OPTIONS path go through the "Content negotiation" diagram and optionally the "Conditional requests" diagram if the resource exists.

The red squares refer to another diagram. The light green squares indicate a response. Other squares may be either a callback or a question answered by Cowboy itself. Green arrows tend to indicate the default behavior if the callback is undefined.

19.1 Start

All requests start from here.



A series of callbacks are called in succession to perform a general checkup of the service, the request line and request headers.

The request body, if any, is not expected to have been received for any of these steps. It is only processed at the end of the "PUT, POST and PATCH methods" diagram, when all conditions have been met.

The `known_methods` and `allowed_methods` callbacks return a list of methods. Cowboy then checks if the request method is in the list, and stops otherwise.

The `is_authorized` callback may be used to check that access to the resource is authorized. Authentication may also be performed as needed. When authorization is denied, the return value from the callback must include a challenge applicable to the requested resource, which will be sent back to the client in the `www-authenticate` header.

This diagram is immediately followed by either the "OPTIONS method" diagram when the request method is `OPTIONS`, or the "Content negotiation" diagram otherwise.

19.2 OPTIONS method

This diagram only applies to `OPTIONS` requests.

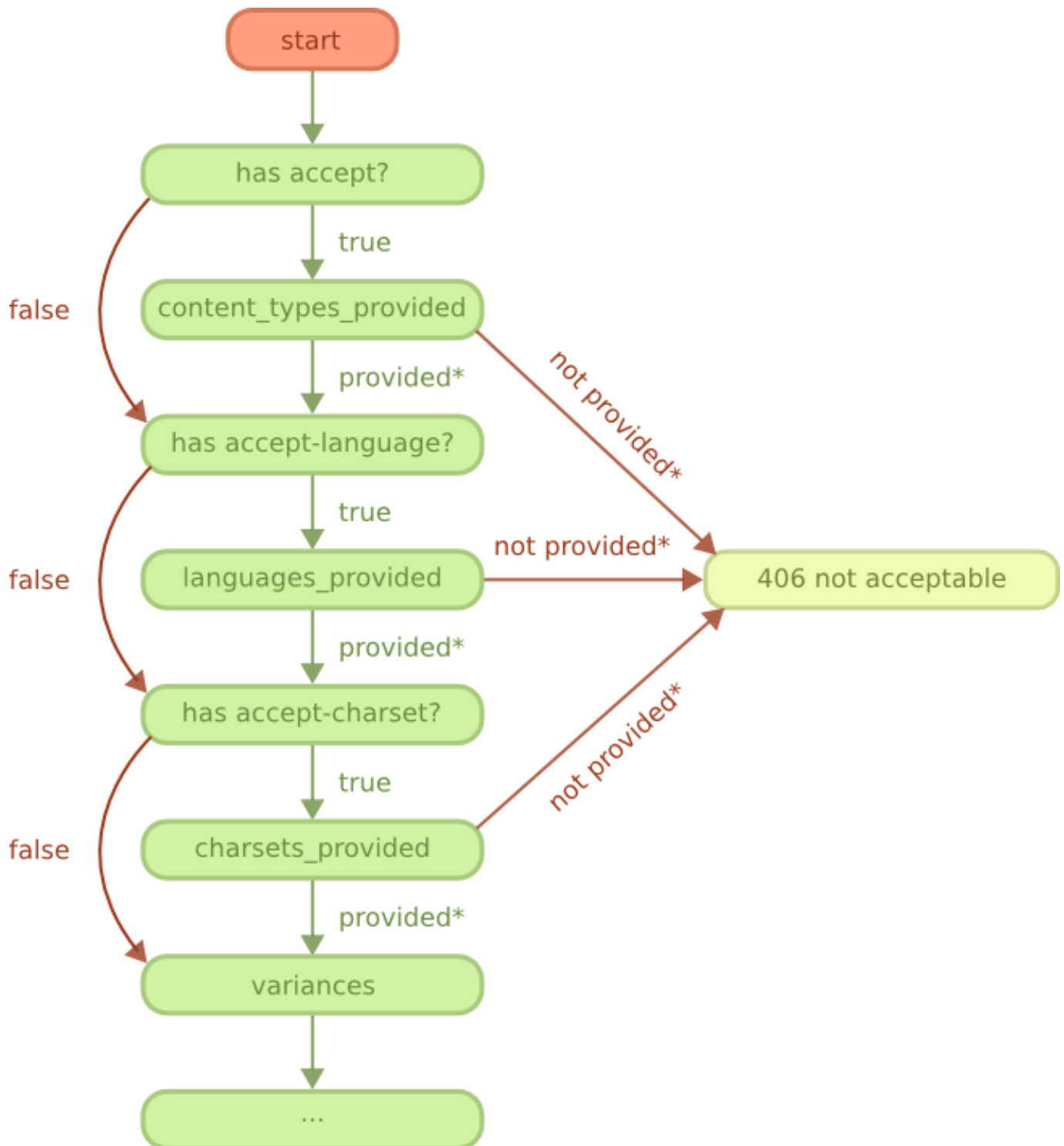


The `options` callback may be used to add information about the resource, such as media types or languages provided; allowed methods; any extra information. A response body may also be set, although clients should not be expected to read it.

If the `options` callback is not defined, Cowboy will send a response containing the list of allowed methods by default.

19.3 Content negotiation

This diagram applies to all request methods other than `OPTIONS`. It is executed right after the "Start" diagram is completed.



The purpose of these steps is to determine an appropriate representation to be sent back to the client.

The request may contain any of the accept header; the accept-language header; or the accept-charset header. When present, Cowboy will parse the headers and then call the corresponding callback to obtain the list of provided content-type, language or charset for this resource. It then automatically select the best match based on the request.

If a callback is not defined, Cowboy will select the content-type, language or charset that the client prefers.

The `content_types_provided` also returns the name of a callback for every content-type it accepts. This callback will only be called at the end of the "GET and HEAD methods" diagram, when all conditions have been met.

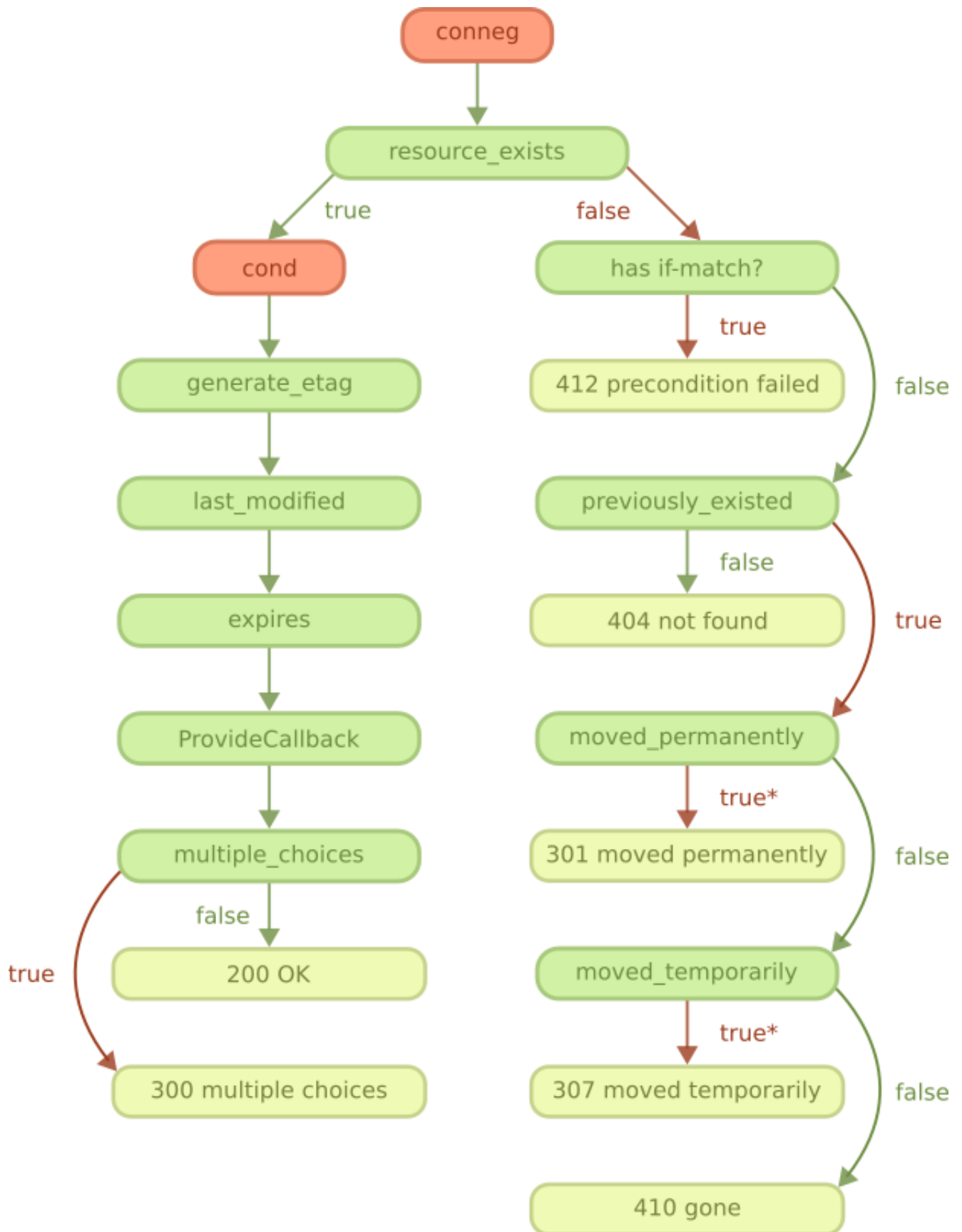
The selected content-type, language and charset are saved as meta values in the Req object. You **should** use the appropriate representation if you set a response body manually (alongside an error code, for example).

This diagram is immediately followed by the "GET and HEAD methods" diagram, the "PUT, POST and PATCH methods" diagram, or the "DELETE method" diagram, depending on the method.

19.4 GET and HEAD methods

This diagram only applies to GET and HEAD requests.

For a description of the `cond` step, please see the "Conditional requests" diagram.



When the resource exists, and the conditional steps succeed, the resource can be retrieved.

Cowboy prepares the response by first retrieving metadata about the representation, then by calling the `ProvideResource`

callback. This is the callback you defined for each content-types you returned from `content_types_provided`. This callback returns the body that will be sent back to the client, or a fun if the body must be streamed.

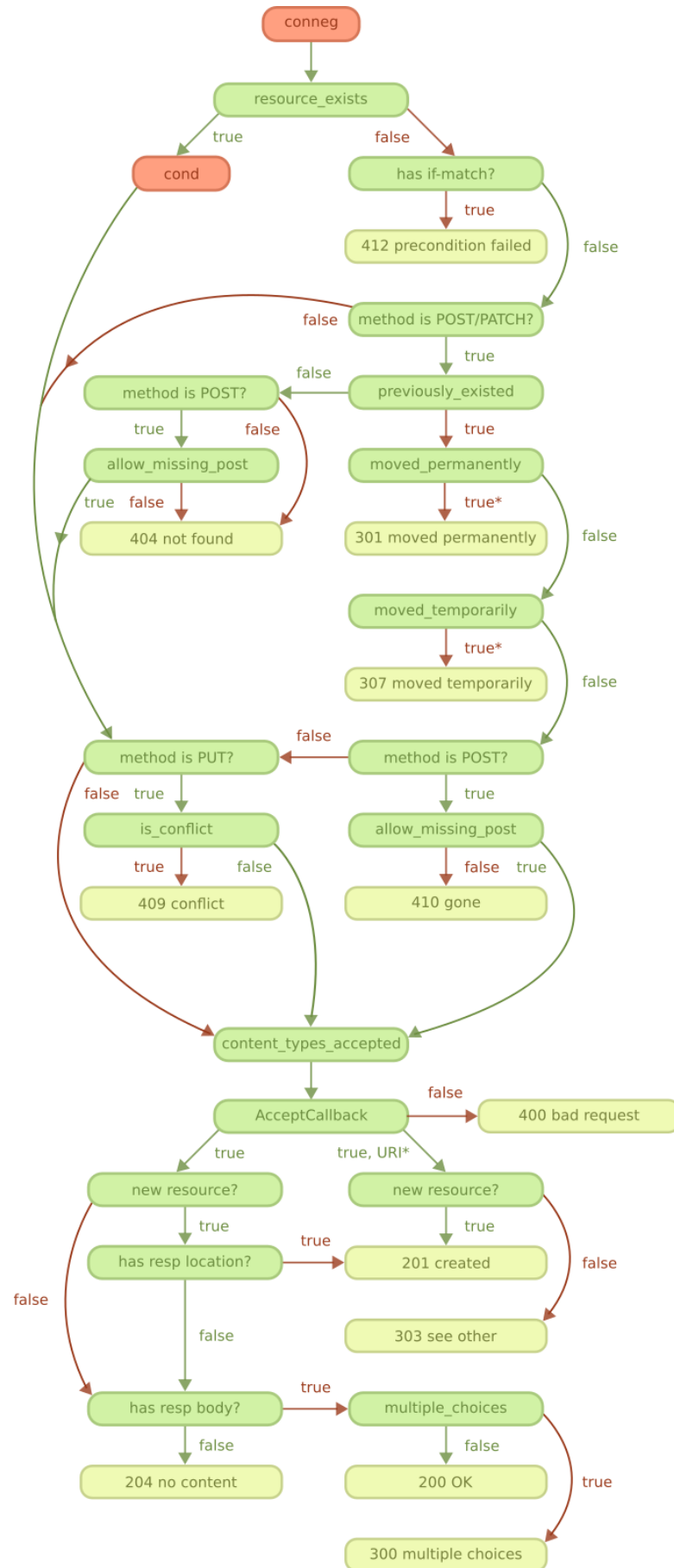
When the resource does not exist, Cowboy will figure out whether the resource existed previously, and if so whether it was moved elsewhere in order to redirect the client to the new URI.

The `moved_permanently` and `moved_temporarily` callbacks must return the new location of the resource if it was in fact moved.

19.5 PUT, POST and PATCH methods

This diagram only applies to PUT, POST and PATCH requests.

For a description of the `cond` step, please see the "Conditional requests" diagram.



When the resource exists, first the conditional steps are executed. When that succeeds, and the method is PUT, Cowboy will call the `is_conflict` callback. This function can be used to prevent potential race conditions, by locking the resource for example.

Then all three methods reach the `content_types_accepted` step that we will describe in a few paragraphs.

When the resource does not exist, and the method is PUT, Cowboy will check for conflicts and then move on to the `content_types_accepted` step. For other methods, Cowboy will figure out whether the resource existed previously, and if so whether it was moved elsewhere. If the resource is truly non-existent, the method is POST and the call for `allow_missing_post` returns `true`, then Cowboy will move on to the `content_types_accepted` step. Otherwise the request processing ends there.

The `moved_permanently` and `moved_temporarily` callbacks must return the new location of the resource if it was in fact moved.

The `content_types_accepted` returns a list of content-types it accepts, but also the name of a callback for each of them. Cowboy will select the appropriate callback for processing the request body and call it.

This callback may return one of three different return values.

If an error occurred while processing the request body, it must return `false` and Cowboy will send an appropriate error response.

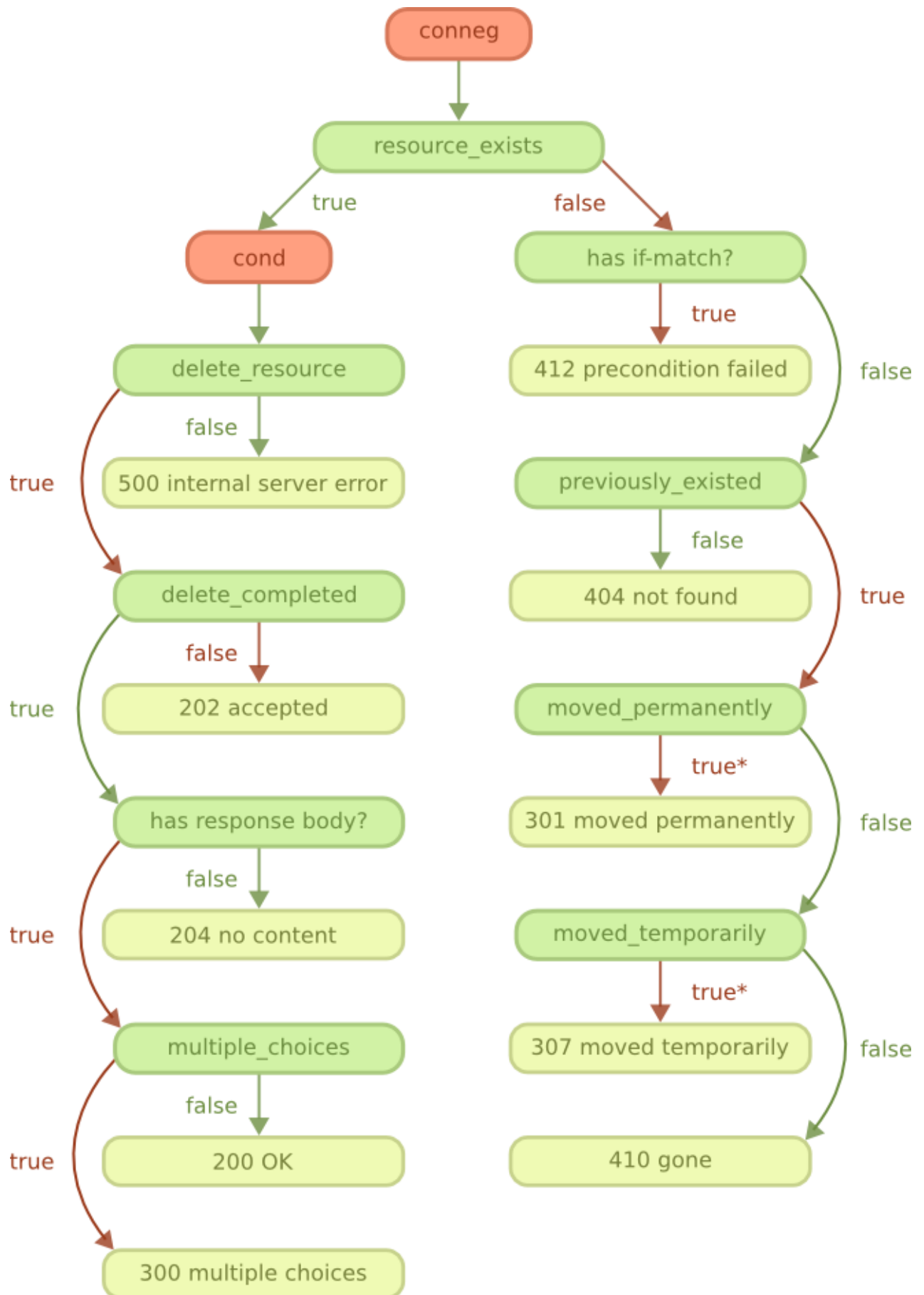
If the method is POST, then you may return `true` with an URI of where the resource has been created. This is especially useful for writing handlers for collections.

Otherwise, return `true` to indicate success. Cowboy will select the appropriate response to be sent depending on whether a resource has been created, rather than modified, and on the availability of a location header or a body in the response.

19.6 DELETE method

This diagram only applies to DELETE requests.

For a description of the `cond` step, please see the "Conditional requests" diagram.



When the resource exists, and the conditional steps succeed, the resource can be deleted.

Deleting the resource is a two steps process. First the callback `delete_resource` is executed. Use this callback to delete the resource.

Because the resource may be cached, you must also delete all cached representations of this resource in the system. This operation may take a while though, so you may return before it finished.

Cowboy will then call the `delete_completed` callback. If you know that the resource has been completely deleted from your system, including from caches, then you can return `true`. If any doubts persist, return `false`. Cowboy will assume `true` by default.

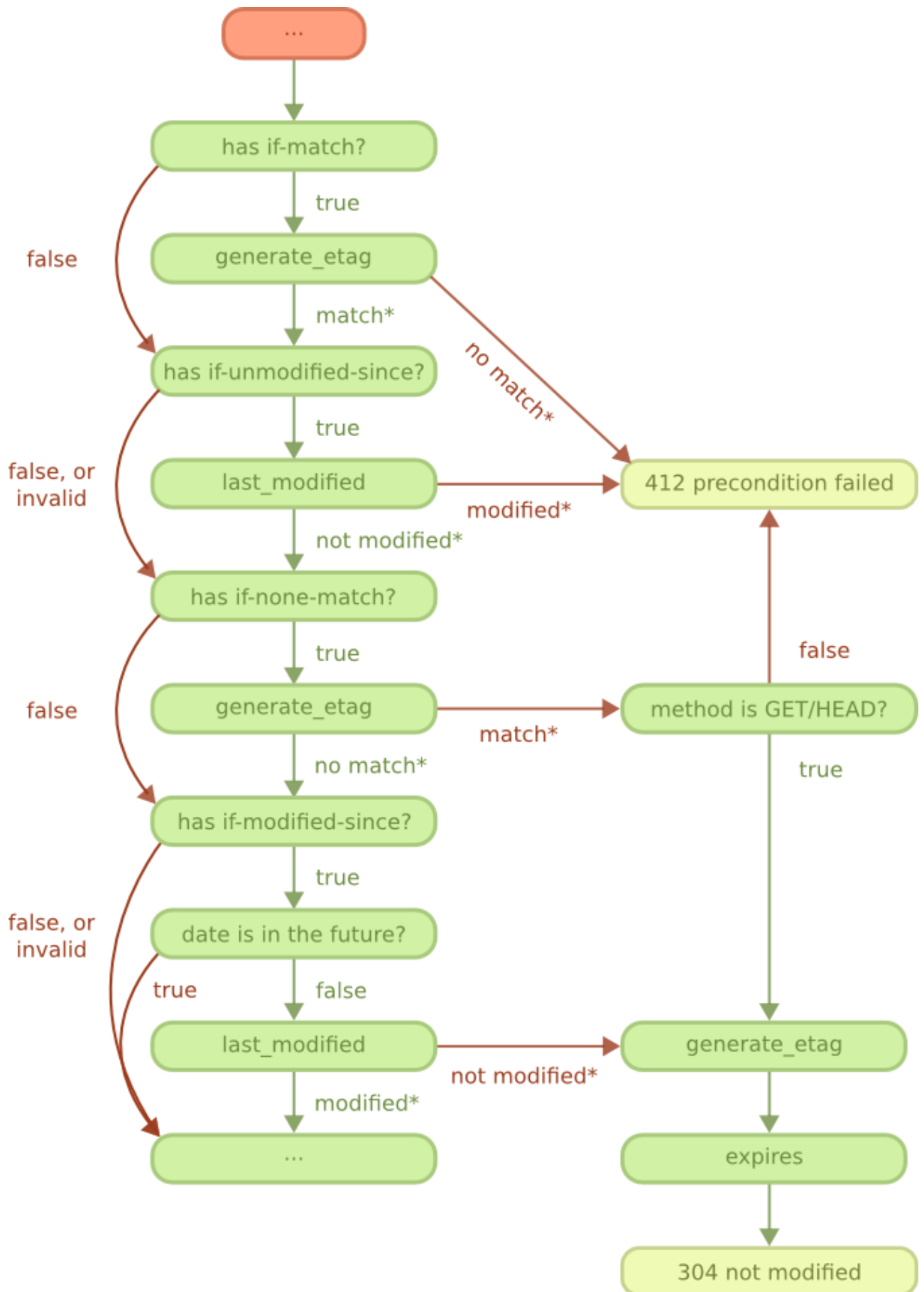
To finish, Cowboy checks if you set a response body, and depending on that, sends the appropriate response.

When the resource does not exist, Cowboy will figure out whether the resource existed previously, and if so whether it was moved elsewhere in order to redirect the client to the new URI.

The `moved_permanently` and `moved_temporarily` callbacks must return the new location of the resource if it was in fact moved.

19.7 Conditional requests

This diagram applies to all request methods other than `OPTIONS`. It is executed right after the `resource_exists` callback, when the resource exists.



A request becomes conditional when it includes either of the `if-match` header; the `if-unmodified-since` header; the `if-none-match` header; or the `if-modified-since` header.

If the condition fails, the request ends immediately without any retrieval or modification of the resource.

The `generate_etag` and `last_modified` are called as needed. Cowboy will only call them once and then cache the results for subsequent use.

Chapter 20

Designing a resource handler

This chapter aims to provide you with a list of questions you must answer in order to write a good resource handler. It is meant to be usable as a step by step guide.

20.1 The service

Can the service become unavailable, and when it does, can we detect it? For example, database connectivity problems may be detected early. We may also have planned outages of all or parts of the system. Implement the `service_available` callback.

What HTTP methods does the service implement? Do we need more than the standard `OPTIONS`, `HEAD`, `GET`, `PUT`, `POST`, `PATCH` and `DELETE`? Are we not using one of those at all? Implement the `known_methods` callback.

20.2 Type of resource handler

Am I writing a handler for a collection of resources, or for a single resource?

The semantics for each of these are quite different. You should not mix collection and single resource in the same handler.

20.3 Collection handler

Skip this section if you are not doing a collection.

Is the collection hardcoded or dynamic? For example, if you use the route `/users` for the collection of users then the collection is hardcoded; if you use `/forums/:category` for the collection of threads then it isn't. When the collection is hardcoded you can safely assume the resource always exists.

What methods should I implement?

`OPTIONS` is used to get some information about the collection. It is recommended to allow it even if you do not implement it, as Cowboy has a default implementation built-in.

`HEAD` and `GET` are used to retrieve the collection. If you allow `GET`, also allow `HEAD` as there's no extra work required to make it work.

`POST` is used to create a new resource inside the collection. Creating a resource by using `POST` on the collection is useful when resources may be created before knowing their URI, usually because parts of it are generated dynamically. A common case is some kind of auto incremented integer identifier.

The next methods are more rarely allowed.

`PUT` is used to create a new collection (when the collection isn't hardcoded), or replace the entire collection.

DELETE is used to delete the entire collection.

PATCH is used to modify the collection using instructions given in the request body. A PATCH operation is atomic. The PATCH operation may be used for such things as reordering; adding, modifying or deleting parts of the collection.

20.4 Single resource handler

Skip this section if you are doing a collection.

What methods should I implement?

OPTIONS is used to get some information about the resource. It is recommended to allow it even if you do not implement it, as Cowboy has a default implementation built-in.

HEAD and GET are used to retrieve the resource. If you allow GET, also allow HEAD as there's no extra work required to make it work.

POST is used to update the resource.

PUT is used to create a new resource (when it doesn't already exist) or replace the resource.

DELETE is used to delete the resource.

PATCH is used to modify the resource using instructions given in the request body. A PATCH operation is atomic. The PATCH operation may be used for adding, removing or modifying specific values in the resource.

20.5 The resource

Following the above discussion, implement the `allowed_methods` callback.

Does the resource always exist? If it may not, implement the `resource_exists` callback.

Do I need to authenticate the client before they can access the resource? What authentication mechanisms should I provide? This may include form-based, token-based (in the URL or a cookie), HTTP basic, HTTP digest, SSL certificate or any other form of authentication. Implement the `is_authorized` callback.

Do I need fine-grained access control? How do I determine that they are authorized access? Handle that in your `is_authorized` callback.

Can access to a resource be forbidden regardless of access being authorized? A simple example of that is censorship of a resource. Implement the `forbidden` callback.

Are there any constraints on the length of the resource URI? For example, the URI may be used as a key in storage and may have a limit in length. Implement `uri_too_long`.

20.6 Representations

What media types do I provide? If text based, what charsets are provided? What languages do I provide?

Implement the mandatory `content_types_provided`. Prefix the callbacks with `to_` for clarity. For example, `to_html` or `to_text`.

Implement the `languages_provided` or `charsets_provided` callbacks if applicable.

Is there any other header that may make the representation of the resource vary? Implement the `variances` callback.

Depending on your choices for caching content, you may want to implement one or more of the `generate_etag`, `last_modified` and `expires` callbacks.

Do I want the user or user agent to actively choose a representation available? Send a list of available representations in the response body and implement the `multiple_choices` callback.

20.7 Redirections

Do I need to keep track of what resources were deleted? For example, you may have a mechanism where moving a resource leaves a redirect link to its new location. Implement the `previously_existed` callback.

Was the resource moved, and is the move temporary? If it is explicitly temporary, for example due to maintenance, implement the `moved_temporarily` callback. Otherwise, implement the `moved_permanently` callback.

20.8 The request

Do you need to read the query string? Individual headers? Implement `malformed_request` and do all the parsing and validation in this function. Note that the body should not be read at this point.

May there be a request body? Will I know its size? What's the maximum size of the request body I'm willing to accept? Implement `valid_entity_length`.

Finally, take a look at the sections corresponding to the methods you are implementing.

20.9 OPTIONS method

Cowboy by default will send back a list of allowed methods. Do I need to add more information to the response? Implement the `options` method.

20.10 GET and HEAD methods

If you implement the methods GET and/or HEAD, you must implement one `ProvideResource` callback for each content-type returned by the `content_types_provided` callback.

20.11 PUT, POST and PATCH methods

If you implement the methods PUT, POST and/or PATCH, you must implement the `content_types_accepted` callback, and one `AcceptCallback` callback for each content-type it returns. Prefix the `AcceptCallback` callback names with `from_` for clarity. For example, `from_html` or `from_json`.

Do we want to allow the POST method to create individual resources directly through their URI (like PUT)? Implement the `allow_missing_post` callback. It is recommended to explicitly use PUT in these cases instead.

May there be conflicts when using PUT to create or replace a resource? Do we want to make sure that two updates around the same time are not cancelling one another? Implement the `is_conflict` callback.

20.12 DELETE methods

If you implement the method DELETE, you must implement the `delete_resource` callback.

When `delete_resource` returns, is the resource completely removed from the server, including from any caching service? If not, and/or if the deletion is asynchronous and we have no way of knowing it has been completed yet, implement the `delete_completed` callback.

Part VII

Websocket

Chapter 21

The Websocket protocol

This chapter explains what Websocket is and why it is a vital component of soft realtime Web applications.

21.1 Description

Websocket is an extension to HTTP that emulates plain TCP connections between the client, typically a Web browser, and the server. It uses the HTTP Upgrade mechanism to establish the connection.

Websocket connections are fully asynchronous, unlike HTTP/1.1 (synchronous) and HTTP/2 (asynchronous, but the server can only initiate streams in response to requests). With Websocket, the client and the server can both send frames at any time without any restriction. It is closer to TCP than any of the HTTP protocols.

Websocket is an IETF standard. Cowboy supports the standard and all drafts that were previously implemented by browsers, excluding the initial flawed draft sometimes known as "version 0".

21.2 Websocket vs HTTP/2

For a few years Websocket was the only way to have a bidirectional asynchronous connection with the server. This changed when HTTP/2 was introduced. While HTTP/2 requires the client to first perform a request before the server can push data, this is only a minor restriction as the client can do so just as it connects.

Websocket was designed as a kind-of-TCP channel to a server. It only defines the framing and connection management and lets the developer implement a protocol on top of it. For example you could implement IRC over Websocket and use a Javascript IRC client to speak to the server.

HTTP/2 on the other hand is just an improvement over the HTTP/1.1 connection and request/response mechanism. It has the same semantics as HTTP/1.1.

If all you need is to access an HTTP API, then HTTP/2 should be your first choice. On the other hand, if what you need is a different protocol, then you can use Websocket to implement it.

21.3 Implementation

Cowboy implements Websocket as a protocol upgrade. Once the upgrade is performed from the `init/2` callback, Cowboy switches to Websocket. Please consult the next chapter for more information on initiating and handling Websocket connections.

The implementation of Websocket in Cowboy is validated using the Autobahn test suite, which is an extensive suite of tests covering all aspects of the protocol. Cowboy passes the suite with 100% success, including all optional tests.

Cowboy's Websocket implementation also includes the permessage-deflate and x-webkit-deflate-frame compression extensions.

Cowboy will automatically use compression when the `compress` option is returned from the `init/2` function.

Chapter 22

Websocket handlers

Websocket handlers provide an interface for upgrading HTTP/1.1 connections to Websocket and sending or receiving frames on the Websocket connection.

As Websocket connections are established through the HTTP/1.1 upgrade mechanism, Websocket handlers need to be able to first receive the HTTP request for the upgrade, before switching to Websocket and taking over the connection. They can then receive or send Websocket frames, handle incoming Erlang messages or close the connection.

22.1 Upgrade

The `init/2` callback is called when the request is received. To establish a Websocket connection, you must switch to the `cowboy_websocket` module:

```
init(Req, State) ->
    {cowboy_websocket, Req, State}.
```

Cowboy will perform the Websocket handshake immediately. Note that the handshake will fail if the client did not request an upgrade to Websocket.

The `Req` object becomes unavailable after this function returns. Any information required for proper execution of the Websocket handler must be saved in the state.

22.2 Subprotocol

The client may provide a list of Websocket subprotocols it supports in the `sec-websocket-protocol` header. The server **must** select one of them and send it back to the client or the handshake will fail.

For example, a client could understand both STOMP and MQTT over Websocket, and provide the header:

```
sec-websocket-protocol: v12.stomp, mqtt
```

If the server only understands MQTT it can return:

```
sec-websocket-protocol: mqtt
```

This selection must be done in `init/2`. An example usage could be:

```
init(Req0, State) ->
    case cowboy_req:parse_header(<<"sec-websocket-protocol">>, Req0) of
        undefined ->
            {cowboy_websocket, Req0, State};
```

```

Subprotocols ->
  case lists:keymember(<<"mqtt">>, 1, Subprotocols) of
    true ->
      Req = cowboy_req:set_resp_header(<<"sec-websocket-protocol">>,
        <<"mqtt">>, Req0),
      {cowboy_websocket, Req, State};
    false ->
      Req = cowboy_req:reply(400, Req0),
      {ok, Req, State}
  end
end.

```

22.3 Post-upgrade initialization

Cowboy has separate processes for handling the connection and requests. Because Websocket takes over the connection, the Websocket protocol handling occurs in a different process than the request handling.

This is reflected in the different callbacks Websocket handlers have. The `init/2` callback is called from the temporary request process and the `websocket_` callbacks from the connection process.

This means that some initialization cannot be done from `init/2`. Anything that would require the current pid, or be tied to the current pid, will not work as intended. The optional `websocket_init/1` can be used instead:

```

websocket_init(State) ->
  erlang:start_timer(1000, self(), <<"Hello!">>),
  {ok, State}.

```

All Websocket callbacks share the same return values. This means that we can send frames to the client right after the upgrade:

```

websocket_init(State) ->
  {reply, {text, <<"Hello!">>}, State}.

```

22.4 Receiving frames

Cowboy will call `websocket_handle/2` whenever a text, binary, ping or pong frame arrives from the client.

The handler can handle or ignore the frames. It can also send frames back to the client or stop the connection.

The following snippet echoes back any text frame received and ignores all others:

```

websocket_handle(Frame = {text, _}, State) ->
  {reply, Frame, State};
websocket_handle(_Frame, State) ->
  {ok, State}.

```

Note that ping and pong frames require no action from the handler as Cowboy will automatically reply to ping frames. They are provided for informative purposes only.

22.5 Receiving Erlang messages

Cowboy will call `websocket_info/2` whenever an Erlang message arrives.

The handler can handle or ignore the messages. It can also send frames to the client or stop the connection.

The following snippet forwards log messages to the client and ignores all others:

```
websocket_info({log, Text}, State) ->
    {reply, {text, Text}, State};
websocket_info(_Info, State) ->
    {ok, State}.
```

22.6 Sending frames

All `websocket_` callbacks share return values. They may send zero, one or many frames to the client.

To send nothing, just return an ok tuple:

```
websocket_info(_Info, State) ->
    {ok, State}.
```

To send one frame, return a reply tuple with the frame to send:

```
websocket_info(_Info, State) ->
    {reply, {text, <<"Hello!">>}, State}.
```

You can send frames of any type: text, binary, ping, pong or close frames.

To send many frames at once, return a reply tuple with the list of frames to send:

```
websocket_info(_Info, State) ->
    {reply, [
        {text, "Hello"},
        {text, <<"world!">>},
        {binary, <<0:8000>>}
    ], State}.
```

They are sent in the given order.

22.7 Keeping the connection alive

Cowboy will automatically respond to ping frames sent by the client. They are still forwarded to the handler for informative purposes, but no further action is required.

Cowboy does not send ping frames itself. The handler can do it if required. A better solution in most cases is to let the client handle pings. Doing it from the handler would imply having an additional timer per connection and this can be a considerable cost for servers that need to handle large numbers of connections.

Cowboy can be configured to close idle connections automatically. It is highly recommended to configure a timeout here, to avoid having processes linger longer than needed.

The `init/2` callback can set the timeout to be used for the connection. For example, this would make Cowboy close connections idle for more than 30 seconds:

```
init(Req, State) ->
    {cowboy_websocket, Req, State, #{
        idle_timeout => 30000}}.
```

This value cannot be changed once it is set. It defaults to 60000.

22.8 Saving memory

The Websocket connection process can be set to hibernate after the callback returns.

Simply add an `hibernate` field to the `ok` or `reply` tuples:

```
websocket_init(State) ->
    {ok, State, hibernate}.

websocket_handle(_Frame, State) ->
    {ok, State, hibernate}.

websocket_info(_Info, State) ->
    {reply, {text, <<"Hello!">>}, State, hibernate}.
```

It is highly recommended to write your handlers with `hibernate` enabled, as this allows to greatly reduce the memory usage. Do note however that an increase in the CPU usage or latency can be observed instead, in particular for the more busy connections.

22.9 Closing the connection

The connection can be closed at any time, either by telling Cowboy to stop it or by sending a close frame.

To tell Cowboy to close the connection, use a `stop` tuple:

```
websocket_info(_Info, State) ->
    {stop, State}.
```

Sending a `close` frame will immediately initiate the closing of the Websocket connection. Note that when sending a list of frames that include a close frame, any frame found after the close frame will not be sent.

Part VIII

Advanced

Chapter 23

Streams

A stream is the set of messages that form an HTTP request/response pair.

The term stream comes from HTTP/2. In Cowboy, it is also used when talking about HTTP/1.1 or HTTP/1.0. It should not be confused with streaming the request or response body.

All versions of HTTP allow clients to initiate streams. HTTP/2 is the only one also allowing servers, through its server push feature. Both client and server-initiated streams go through the same process in Cowboy.

23.1 Stream handlers

Stream handlers must implement five different callbacks. Four of them are directly related; one is special.

All callbacks receives the stream ID as first argument.

Most of them can return a list of commands to be executed by Cowboy. When callbacks are chained, it is possible to intercept and modify these commands. This can be useful for modifying responses for example.

The `init/3` callback is invoked when a new request comes in. It receives the `Req` object and the protocol options for this listener.

The `data/4` callback is invoked when data from the request body is received. It receives both this data and a flag indicating whether more is to be expected.

The `info/3` callback is invoked when an Erlang message is received for this stream. They will typically be messages sent by the request process.

Finally the `terminate/3` callback is invoked with the terminate reason for the stream. The return value is ignored. Note that as with all terminate callbacks in Erlang, there is no strong guarantee that it will be called.

The special callback `early_error/5` is called when an error occurs before the request headers were fully received and Cowboy is sending a response. It receives the partial `Req` object, the error reason, the protocol options and the response Cowboy will send. This response must be returned, possibly modified.

23.2 Built-in handlers

Cowboy comes with two handlers.

`cowboy_stream_h` is the default stream handler. It is the core of much of the functionality of Cowboy. All chains of stream handlers should call it last.

`cowboy_compress_h` will automatically compress responses when possible. It is not enabled by default. It is a good example for writing your own handlers that will modify responses.

Chapter 24

Middleware

Cowboy delegates the request processing to middleware components. By default, two middlewares are defined, for the routing and handling of the request, as is detailed in most of this guide.

Middleware give you complete control over how requests are to be processed. You can add your own middlewares to the mix or completely change the chain of middlewares as needed.

Cowboy will execute all middlewares in the given order, unless one of them decides to stop processing.

24.1 Usage

Middleware only need to implement a single callback: `execute/2`. It is defined in the `cowboy_middleware` behavior.

This callback has two arguments. The first is the `Req` object. The second is the environment.

Middleware can return one of three different values:

- `{ok, Req, Env}` to continue the request processing
- `{suspend, Module, Function, Args}` to hibernate
- `{stop, Req}` to stop processing and move on to the next request

Of note is that when hibernating, processing will resume on the given MFA, discarding all previous stacktrace. Make sure you keep the `Req` and `Env` in the arguments of this MFA for later use.

If an error happens during middleware processing, Cowboy will not try to send an error back to the socket, the process will just crash. It is up to the middleware to make sure that a reply is sent if something goes wrong.

24.2 Configuration

The middleware environment is defined as the `env` protocol option. In the previous chapters we saw it briefly when we needed to pass the routing information. It is a list of tuples with the first element being an atom and the second any Erlang term.

Two values in the environment are reserved:

- `listener` contains the name of the listener
- `result` contains the result of the processing

The `listener` value is always defined. The `result` value can be set by any middleware. If set to anything other than `ok`, Cowboy will not process any subsequent requests on this connection.

The middlewares that come with Cowboy may define or require other environment values to perform.

You can update the environment by calling the `cowboy:set_env/3` convenience function, adding or replacing a value in the environment.

24.3 Routing middleware

The routing middleware requires the `dispatch` value. If routing succeeds, it will put the handler name and options in the `handler` and `handler_opts` values of the environment, respectively.

24.4 Handler middleware

The handler middleware requires the `handler` and `handler_opts` values. It puts the result of the request handling into `result`.

Part IX

Additional information

Appendix A

Migrating from Cowboy 1.0 to 2.0

A lot has changed between Cowboy 1.0 and 2.0. The `cowboy_req` interface in particular has seen a massive revamp. Hooks are gone, their functionality can now be achieved via stream handlers.

The documentation has seen great work, in particular the manual. Each module and each function now has its own dedicated manual page with full details and examples.

A.1 Compatibility

Compatibility with Erlang/OTP R16, 17 and 18 has been dropped. Erlang/OTP 19.0 or above is required. It is non-trivial to make Cowboy 2.0 work with older Erlang/OTP versions.

Cowboy 2.0 is not compatible with Cowlib versions older than 2.0. It should be compatible with Ranch 1.0 or above, however it has not been tested with Ranch versions older than 1.4.

Cowboy 2.0 is tested on Arch Linux, Ubuntu, FreeBSD, Windows and OSX. It is tested with every point release (latest patch release) and also with HiPE on the most recent release.

Cowboy 2.0 now comes with Erlang.mk templates.

A.2 Features added

- The HTTP/2 protocol is now supported.
 - Cowboy no longer uses only one process per connection. It now uses one process per connection plus one process per request by default. This is necessary for HTTP/2. There might be a slight drop in performance for HTTP/1.1 connections due to this change.
 - Cowboy internals have largely been reworked in order to support HTTP/2. This opened the way to stream handlers, which are a chain of modules that are called whenever something happens relating to a request/response.
 - The `cowboy_stream_h` stream handler has been added. It provides most of Cowboy's default behavior.
 - The `cowboy_compress_h` stream handler has been added. It compresses responses when possible. It's worth noting that it compresses in more cases than Cowboy 1.0 ever did.
 - Because of the many changes in the internals of Cowboy, many options have been added or modified. Of note is that the Websocket options are now given per handler rather than for the entire listener.
 - Websocket permessage-deflate compression is now supported via the `compress` option.
 - Static file handlers will now correctly find files found in `.ez` archives.
 - Constraints have been generalized and are now used not only in the router but also in some `cowboy_req` functions. Their interface has also been modified to allow for reverse operations and formatting of errors.
-

A.3 Features removed

- SPDY support has been removed. Use HTTP/2 instead.
- Hooks have been removed. Use [stream handlers](#) instead.
- The undocumented `waiting_stream` hack has been removed. It allowed disabling chunked transfer-encoding for HTTP/1.1. It has no equivalent in Cowboy 2.0. Open a ticket if necessary.
- Sub protocols still exist, but their interface has largely changed and they are no longer documented for the time being.

A.4 Changed behaviors

- The handler behaviors have been renamed and are now `cowboy_handler`, `cowboy_loop`, `cowboy_rest` and `cowboy_websocket`.
- Plain HTTP, loop, REST and Websocket handlers have had their `init` and `terminate` callbacks unified. They now all use the `init/2` and `terminate/3` callbacks. The latter is now optional. The `terminate` reason has now been documented for all handlers.
- The tuple returned to switch to a different handler type has changed. It now takes the form `{Module, Req, State}` or `{Module, Req, State, Opts}`, where `Opts` is a map of options to configure the handler. The `timeout` and `hibernate` options must now be specified using this map, where applicable.
- All behaviors that used to accept `halt` or `shutdown` tuples as a return value no longer do so. The return value is now a `stop` tuple, consistent across Cowboy.
- Middlewares can no longer return an `error` tuple. They have to send the response and return a `stop` tuple instead.
- The `known_content_type` REST handler callback has been removed as it was found unnecessary.
- Websocket handlers have both the normal `init/2` and an optional `websocket_init/1` function. The reason for that exception is that the `websocket_*` callbacks execute in a separate process from the `init/2` callback, and it was therefore not obvious how timers or monitors should be setup properly. They are effectively initializing the handler before and after the HTTP/1.1 upgrade.
- Websocket handlers can now send frames directly from `websocket_init/1`. The frames will be sent immediately after the handshake.
- Websocket handler callbacks no longer receive the `Req` argument. The `init/2` callback still receives it and can be used to extract relevant information. The `terminate/3` callback, if implemented, may still receive the `Req` (see next bullet point).
- Websocket handlers have a new `req_filter` option that can be used to customize how much information should be discarded from the `Req` object after the handshake. Note that the `Req` object is only available in `terminate/3` past that point.
- Websocket handlers have their `timeout` default changed from infinity to 60 seconds.

A.5 New functions

- The `cowboy_req:scheme/1` function has been added.
 - The `cowboy_req:uri/1,2` function has been added, replacing the less powerful functions `cowboy_req:url/1` and `cowboy_req:host_url/1`.
 - The functions `cowboy_req:match_qs/2` and `cowboy_req:match_cookies/2` allow matching query string and cookies against constraints.
 - The function `cowboy_req:set_resp_cookie/3` has been added to complement `cowboy_req:set_resp_cookie/4`.
-

- The functions `cowboy_req:resp_header/2, 3` and `cowboy_req:resp_headers/1` have been added. They can be used to retrieve response headers that were previously set.
- The function `cowboy_req:set_resp_headers/2` has been added. It allows setting many response headers at once.
- The functions `cowboy_req:push/3, 4` can be used to push resources for protocols that support it (by default only HTTP/2).

A.6 Changed functions

- The `cowboy:start_http/4` function was renamed to `cowboy:start_clear/3`.
- The `cowboy:start_https/4` function was renamed to `cowboy:start_tls/3`.
- Most, if not all, functions in the `cowboy_req` module have been modified. Please consult the changelog of each individual functions. The changes are mainly about simplifying and clarifying the interface. The `Req` is no longer returned when not necessary, maps are used wherever possible, and some functions have been renamed.
- The position of the `Opts` argument for `cowboy_req:set_resp_cookie/4` has changed to improve consistency. It is now the last argument.

A.7 Removed functions

- The functions `cowboy_req:url/1` and `cowboy_req:host_url/1` have been removed in favor of the new function `cowboy_req:uri/1, 2`.
- The functions `cowboy_req:meta/2, 3` and `cowboy_req:set_meta/3` have been removed. The `Req` object is now a public map, therefore they became unnecessary.
- The functions `cowboy_req:set_resp_body_fun/2, 3` have been removed. For sending files, the function `cowboy_req:set_resp_body/2` can now take a `sendfile` tuple.
- Remove many undocumented functions from `cowboy_req`, including the functions `cowboy_req:get/2` and `cowboy_req:set/2`.

A.8 Other changes

- The correct percent-decoding algorithm is now used for path elements during routing. It will no longer decode `+` characters.
 - The router will now properly handle path segments `.` and `...`.
 - Routing behavior has changed for URIs containing latin1 characters. They are no longer allowed. URIs are expected to be in UTF-8 once they are percent-decoded.
 - Etag comparison in REST handlers has been fixed. Some requests may now fail when they succeeded in the past.
 - The `If-*-Since` headers are now ignored in REST handlers if the corresponding `If-*-Match` header exist. The former is largely a backward compatible header and this shouldn't create any issue. The new behavior follows the current RFCs more closely.
 - The static file handler has been improved to handle more special characters on systems that accept them.
-

Appendix B

HTTP and other specifications

This chapter intends to list all the specification documents for or related to HTTP.

B.1 HTTP

B.1.1 IANA Registries

- [HTTP Method Registry](#)
- [HTTP Status Code Registry](#)
- [Message Headers](#)
- [HTTP Parameters](#)
- [HTTP Alt-Svc Parameter Registry](#)
- [HTTP Authentication Scheme Registry](#)
- [HTTP Cache Directive Registry](#)
- [HTTP Digest Algorithm Values](#)
- [HTTP Origin-Bound Authentication Device Identifier Types](#)
- [HTTP Upgrade Token Registry](#)
- [HTTP Warn Codes](#)
- [HTTP/2 Parameters](#)
- [WebSocket Protocol Registries](#)

B.1.2 Current

- [CORS](#): Cross-Origin Resource Sharing
 - [CSP2](#): Content Security Policy Level 2
 - [DNT](#): Tracking Preference Expression (DNT)
 - [eventsourcing](#): Server-Sent Events
 - [Form content types](#): Form content types
-

- [Preload](#): Preload
 - [REST](#): Fielding's Dissertation
 - [RFC 1945](#): HTTP/1.0
 - [RFC 1951](#): DEFLATE Compressed Data Format Specification version 1.3
 - [RFC 1952](#): GZIP file format specification version 4.3
 - [RFC 2046](#): Multipart media type (in MIME Part Two: Media Types)
 - [RFC 2295](#): Transparent Content Negotiation in HTTP
 - [RFC 2296](#): HTTP Remote Variant Selection Algorithm: RVSA/1.0
 - [RFC 2817](#): Upgrading to TLS Within HTTP/1.1
 - [RFC 2818](#): HTTP Over TLS
 - [RFC 3230](#): Instance Digests in HTTP
 - [RFC 4559](#): SPNEGO-based Kerberos and NTLM HTTP Authentication in Microsoft Windows
 - [RFC 5789](#): PATCH Method for HTTP
 - [RFC 5843](#): Additional Hash Algorithms for HTTP Instance Digests
 - [RFC 5861](#): HTTP Cache-Control Extensions for Stale Content
 - [RFC 5988](#): Web Linking
 - [RFC 6265](#): HTTP State Management Mechanism
 - [RFC 6266](#): Use of the Content-Disposition Header Field
 - [RFC 6454](#): The Web Origin Concept
 - [RFC 6455](#): The WebSocket Protocol
 - [RFC 6585](#): Additional HTTP Status Codes
 - [RFC 6750](#): The OAuth 2.0 Authorization Framework: Bearer Token Usage
 - [RFC 6797](#): HTTP Strict Transport Security (HSTS)
 - [RFC 6903](#): Additional Link Relation Types
 - [RFC 7034](#): HTTP Header Field X-Frame-Options
 - [RFC 7089](#): Time-Based Access to Resource States: Memento
 - [RFC 7230](#): HTTP/1.1 Message Syntax and Routing
 - [RFC 7231](#): HTTP/1.1 Semantics and Content
 - [RFC 7232](#): HTTP/1.1 Conditional Requests
 - [RFC 7233](#): HTTP/1.1 Range Requests
 - [RFC 7234](#): HTTP/1.1 Caching
 - [RFC 7235](#): HTTP/1.1 Authentication
 - [RFC 7239](#): Forwarded HTTP Extension
 - [RFC 7240](#): Prefer Header for HTTP
 - [RFC 7469](#): Public Key Pinning Extension for HTTP
-

- [RFC 7486](#): HTTP Origin-Bound Authentication (HOBA)
- [RFC 7538](#): HTTP Status Code 308 (Permanent Redirect)
- [RFC 7540](#): Hypertext Transfer Protocol Version 2 (HTTP/2)
- [RFC 7541](#): HPACK: Header Compression for HTTP/2
- [RFC 7578](#): Returning Values from Forms: multipart/form-data
- [RFC 7615](#): HTTP Authentication-Info and Proxy-Authentication-Info Response Header Fields
- [RFC 7616](#): HTTP Digest Access Authentication
- [RFC 7617](#): The *Basic* HTTP Authentication Scheme
- [RFC 7639](#): The ALPN HTTP Header Field
- [RFC 7692](#): Compression Extensions for WebSocket
- [RFC 7694](#): HTTP Client-Initiated Content-Encoding
- [RFC 7725](#): An HTTP Status Code to Report Legal Obstacles
- [RFC 7804](#): Salted Challenge Response HTTP Authentication Mechanism
- [RFC 7838](#): HTTP Alternative Services
- [RFC 7932](#): Brotli Compressed Data Format
- [RFC 7936](#): Clarifying Registry Procedures for the WebSocket Subprotocol Name Registry
- [RFC 8053](#): HTTP Authentication Extensions for Interactive Clients
- [RFC 8164](#): Opportunistic Security for HTTP/2
- [RFC 8187](#): Indicating Character Encoding and Language for HTTP Header Field Parameters
- [RFC 8188](#): Encrypted Content-Encoding for HTTP
- [RFC 8246](#): HTTP Immutable Responses
- [Webmention](#): Webmention

B.1.3 Upcoming

- [Content Security Policy: Cookie Controls](#)
 - [Content Security Policy: Embedded Enforcement](#)
 - [Content Security Policy Level 3](#)
 - [Content Security Policy Pinning](#)
 - [Referrer Policy](#)
 - [User Interface Security Directives for Content Security Policy](#)
-

B.1.4 Informative

- [Architecture of the World Wide Web](#)
- [RFC 2936](#): HTTP MIME Type Handler Detection
- [RFC 2964](#): Use of HTTP State Management
- [RFC 3143](#): Known HTTP Proxy/Caching Problems
- [RFC 6202](#): Known Issues and Best Practices for the Use of Long Polling and Streaming in Bidirectional HTTP
- [RFC 6838](#): Media Type Specifications and Registration Procedures
- [RFC 7478](#): Web Real-Time Communication Use Cases and Requirements

B.1.5 Related

- [app: URL Scheme](#)
 - [Beacon](#)
 - [File API](#)
 - [Generic Event Delivery Using HTTP Push](#)
 - [Good Practices for Capability URLs](#)
 - [HTML Living Standard](#)
 - [HTML Living Standard for Web developers](#)
 - [HTML4.01](#)
 - [HTML5](#)
 - [HTML5.1](#)
 - [HTML5.2](#)
 - [Media Fragments URI 1.0](#)
 - [RFC 6690](#): Constrained RESTful Environments (CoRE) Link Format
 - [RFC 7807](#): Problem Details for HTTP APIs
 - [RFC 6906](#): The *profile* Link Relation Type
 - [Subresource Integrity](#)
 - [Tracking Compliance and Scope](#)
 - [Use cases and requirements for Media Fragments](#)
 - [WebRTC 1.0: Real-time Communication Between Browsers](#)
 - [Websocket API](#)
 - [XMLHttpRequest Level 1](#)
 - [XMLHttpRequest Living Standard](#)
-

B.1.6 Seemingly obsolete

- [RFC 2227](#): Simple Hit-Metering and Usage-Limiting for HTTP
- [RFC 2310](#): The Safe Response Header Field
- [RFC 2324](#): Hyper Text Coffee Pot Control Protocol (HTCPCP/1.0)
- [RFC 2660](#): The Secure HyperText Transfer Protocol
- [RFC 2774](#): An HTTP Extension Framework
- [RFC 2965](#): HTTP State Management Mechanism (Cookie2)
- [RFC 3229](#): Delta encoding in HTTP
- [RFC 7168](#): The Hyper Text Coffee Pot Control Protocol for Tea Efflux Appliances (HTCPCP-TEA)
- [SPDY](#): SPDY Protocol
- [x-webkit-deflate-frame](#): Deprecated Websocket compression

B.2 URL

- [RFC 3986](#): URI Generic Syntax
- [RFC 6570](#): URI Template
- [RFC 6874](#): Representing IPv6 Zone Identifiers in Address Literals and URIs
- [RFC 7320](#): URI Design and Ownership
- [URL](#)
- [URL Living Standard](#)

B.3 WebDAV

- [RFC 3253](#): Versioning Extensions to WebDAV
 - [RFC 3648](#): WebDAV Ordered Collections Protocol
 - [RFC 3744](#): WebDAV Access Control Protocol
 - [RFC 4316](#): Datatypes for WebDAV Properties
 - [RFC 4331](#): Quota and Size Properties for DAV Collections
 - [RFC 4437](#): WebDAV Redirect Reference Resources
 - [RFC 4709](#): Mounting WebDAV Servers
 - [RFC 4791](#): Calendaring Extensions to WebDAV (CalDAV)
 - [RFC 4918](#): HTTP Extensions for WebDAV
 - [RFC 5323](#): WebDAV SEARCH
 - [RFC 5397](#): WebDAV Current Principal Extension
 - [RFC 5689](#): Extended MKCOL for WebDAV
 - [RFC 5842](#): Binding Extensions to WebDAV
-

- [RFC 5995](#): Using POST to Add Members to WebDAV Collections
- [RFC 6352](#): CardDAV: vCard Extensions to WebDAV
- [RFC 6578](#): Collection Synchronization for WebDAV
- [RFC 6638](#): Scheduling Extensions to CalDAV
- [RFC 6764](#): Locating Services for Calendaring Extensions to WebDAV (CalDAV) and vCard Extensions to WebDAV (CardDAV)
- [RFC 7809](#): Calendaring Extensions to WebDAV (CalDAV): Time Zones by Reference
- [RFC 7953](#): Calendar Availability
- [RFC 8144](#): Use of the Prefer Header Field in WebDAV

B.4 CoAP

- [RFC 7252](#): The Constrained Application Protocol (CoAP)
 - [RFC 7390](#): Group Communication for CoAP
 - [RFC 7641](#): Observing Resources in CoAP
 - [RFC 7650](#): A CoAP Usage for REsource LOcation And Discovery (RELOAD)
 - [RFC 7959](#): Block-Wise Transfers in CoAP
 - [RFC 7967](#): CoAP Option for No Server Response
 - [RFC 8075](#): Guidelines for Mapping Implementations: HTTP to CoAP
 - [RFC 8132](#): PATCH and FETCH Methods for CoAP
-