Cowboy User Guide

Contents

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 ahead of anything else in this respect. 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.

Chapter 3

Security model

Cowboy's security model is based on the principle that the server delegates security responsibilities to the underlying Erlang/OTP platform, the operating system and to the application code.

Cowboy and its main dependency Cowlib perform input validation and parsing according to the relevant protocol specifications. Cowlib's parsers are strict: invalid input typically results in a crash that Cowboy turns into a 4xx error response (for example a 400 Bad Request). Cowboy also provides numerous options to limit resource usage (timeouts, buffer sizes, header lengths, concurrent streams and so on) in order to mitigate denial of service attacks. These limits must be configured appropriately by the application.

Output construction for protocol elements (status lines, headers, bodies in some cases) is performed by Cowlib, which is lax when building data: it will output largely what it is given, without performing validation or sanitization. Cowboy itself performs only limited sanitization in some cases. For example when using the HTTP/1.1 protocol it will reject response header values containing CRLF characters by default (via the `invalid_response_headers` option set to `error_terminate`). It is the responsibility of the application (and by extension Cowboy when it calls into Cowlib) to sanitize any data provided for use in responses. Response data must be valid according to the relevant specifications.

Cowboy trusts the code it runs: handlers, middleware, stream handlers, router constraints and any other modules provided by the application. Any security decision (authentication, authorization, access control and so on) must be performed by this code. Cowboy does not implement any such mechanisms itself.

All code runs with the privileges of the Erlang virtual machine.

Part II

Introduction

Chapter 4

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.

4.1 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.

4.2 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 24.0 and newer.

4.3 License

Cowboy uses the ISC License.

```
Copyright (c) 2011-2026, 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
```

4.4 Versioning

Cowboy uses Semantic Versioning 2.0.0.

4.5 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 5

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.

5.1 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.

5.2 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.

5.3 Cowboy setup

We will modify the *Makefile* to tell the build system it needs to fetch and compile Cowboy, and that we will use releases:

```
PROJECT = hello_erlang

DEPS = cowboy
dep_cowboy_commit = 2.16.1

REL_DEPS = relx

DEP_PLUGINS = cowboy

include erlang.mk
```

The `DEP_PLUGINS` line tells the build system to load the plugins Cowboy provides. These include predefined templates that we will use soon.

The `REL_DEPS` line tells the build system to fetch and build `relx`, the library that will create the release.

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.

5.4 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.

5.5 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!

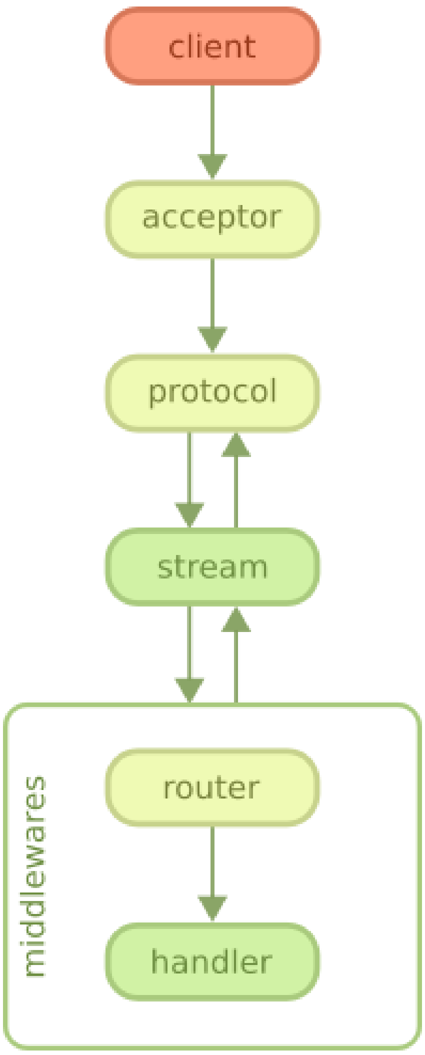
Chapter 6

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.

6.1 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. By default, the request process executes the router and then the 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.

6.2 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.

6.3 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.

6.4 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.

6.5 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 7

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.

7.1 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.

7.2 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 set up a custom listener.

```
start(_Type, _Args) ->
    Dispatch = cowboy_router:compile([
        {'_', [{"/", hello_handler, []}]}
    ]),
    {ok, _} = cowboy:start_tls(my_https_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.

7.3 Stopping the listener

When starting listeners along with the application it is a good idea to also stop the listener when the application stops. This can be done by calling `cowboy:stop_listener/1` in the application's stop function:

```
stop(_State) ->
    ok = cowboy:stop_listener(my_http_listener).
```

7.4 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 8

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.

Cowboy routes requests using the following algorithm:

- If no configured host matches the request URI, a 400 response is returned.
- Otherwise, the first configured host that matches the request URI will be used. Only the paths configured for this host will be considered.
- If none of the configured paths found in the previous step match the request URI, a 404 response is returned.
- Otherwise, the handler and its initial state are added to the environment and the request continues to be processed.

Note
It is possible to run into a situation where two hosts match a request URI, but only the paths on the second host match the request URI. In this case the expected result is a 404 response because the only paths used during routing are the paths from the first configured host that matches the request URI.

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

8.1 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.

8.2 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_legen` 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]]".
```

While Cowboy does not reject multiple brackets in a route, the behavior may be undefined if the route is under-specified. For example, this route requires constraints to determine what is a chapter and what is a page, since they are both optional:

```
PathMatch = "/book/[:chapter]/[:page]".
```

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 = "*".
```

8.3 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.

8.4 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, TransOpts, ProtoOpts
cowboy:start_clear(my_http_listener,
    [{port, 8080}],
    #{env => #{dispatch => Dispatch}}
).
```

8.5 Using persistent_term

The routes can be stored in `persistent_term` starting from Erlang/OTP 21.2. This may give a performance improvement when there are a large number of routes.

To use this functionality you need to compile the routes, store them in `persistent_term` and then inform Cowboy:

```
Dispatch = cowboy_router:compile([
    {'_', [{'_', my_handler, #{}}]}
]),
persistent_term:put(my_app_dispatch, Dispatch),
cowboy:start_clear(my_http_listener,
    [{port, 8080}],
    #{env => #{dispatch => {persistent_term, my_app_dispatch}}}
).
```

8.6 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.

When using `persistent_term` there is no need to call this function, you can simply put the new routes in the storage.

Chapter 9

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.

9.1 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}
```

9.2 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.

9.3 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 two other operations are currently experimental. They are meant to help implement HATEOAS type services, but proper support for HATEOAS is not expected to be available before Cowboy 3.0 because of Cowboy's current router's limitations.

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.

For the common case of constraints that only need to implement the `forward` operation, the `cowboy_constraints:from_fun/1` function can be used. It turns a 1-arity function (that returns a value on success or throws on failure) into a proper constraint.

```
Constraint = cowboy_constraints:from_fun(fun cow_http:ensure_token/1).
```

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

Part IV

Handlers

Chapter 10

Handlers

Handlers are Erlang modules that handle HTTP requests.

10.1 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`.

10.2 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}.
```

10.3 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 11

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, but it also applies to any response that takes a long time to send.

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.

11.1 Initialization

The `init/2` function must return a `cowboy_loop` tuple to enable loop handler behavior. This tuple may optionally contain the atom `hibernate` to make the process enter hibernation until a message is received. Alternatively, the tuple may optionally contain a positive integer to create a `timeout` message when the process has not received messages for too long.

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}.
```

This makes the process time out after 1000ms of idle time.

```
init(Req, State) ->
    {cowboy_loop, Req, State, 1000}.
```

11.2 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.

11.3 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}.
```

11.4 Cleaning up

Please refer to the Handlers chapter for general instructions about cleaning up.

11.5 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.

11.6 Idle timeout

You may activate timeout events by returning a positive integer `N` as part of the `loop` tuple callbacks return. The default value is `infinity`. The `info` callback will be called with the atom `timeout` unless a message is received within `N` milliseconds:

```
info(timeout, Req, State) ->
    %% Do something...
    {ok, Req, State, 1000}.
```

Chapter 12

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.

12.1 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"}}
```

12.2 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 the application's private directory (e.g. `my_app/priv`).

The following rule will serve any file found in the `my_app` application's private directory in the `my_app/priv/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"}}
```

12.3 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">>, []}}]}}
```

12.4 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 13

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.

13.1 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`; the connection `peer` address and port; and the TLS certificate `cert` when applicable.

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 is allowed, but extra caution must be used when modifying existing fields. You can add as many new fields as necessary, however. Just make sure to namespace the field names so that no conflict can occur with future Cowboy updates or with third party projects.

13.2 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.

13.3 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.

13.4 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.

13.5 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.

13.6 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.

13.7 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[]`. Cowboy does not require the `[]` suffix to properly handle repeated key names.

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 given. Constraints do not automatically look inside the list, a custom constraint must be written if that is necessary.

13.8 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">>, []}).
```

13.9 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 14

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.

14.1 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.

14.2 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.

14.3 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}).
```

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

It is also possible to asynchronously read the request body using auto mode:

```
Ref = make_ref(),
cowboy_req:cast({read_body, self(), Ref, auto, infinity}, Req).
```

Cowboy will wait indefinitely for data and then send a `request_body` message as soon as it has data available, regardless of length.

```
receive
    {request_body, Ref, nofin, Data} ->
        do_something(Data);
    {request_body, Ref, fin, _BodyLen, Data} ->
        do_something(Data)
end.
```

Asynchronous reading of data pairs well with loop handlers.

14.4 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.

14.5 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 15

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.

15.1 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.

15.2 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.

Cowboy also provides a function to send response trailers. Response trailers are semantically equivalent to the headers you send in the response, only they are sent at the end. This is especially useful to attach information to the response that could not be generated until the response body was fully generated.

Trailer fields must be listed in the trailer header. Any field not listed might be dropped by the client or an intermediary.

```
Req = cowboy_req:stream_reply(200, #{
    <<"content-type">> => <<"text/html">>,
    <<"trailer">> => <<"expires, content-md5">>
}, Req0),

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

cowboy_req:stream_trailers(#{
    <<"expires">> => <<"Sun, 10 Dec 2017 19:13:47 GMT">>,
    <<"content-md5">> => <<"c6081d20ff41a42ce17048ed1c0345e2">>
}, Req).
```

The stream ends with trailers. It is no longer possible to send data after sending trailers. You cannot send trailers after setting the `fin` flag when streaming the body.

15.3 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).
```

15.4 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).
```

15.5 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`.

15.6 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).
```

15.7 Informational responses

Cowboy allows you to send informational responses.

Informational responses are responses that have a status code between 100 and 199. Any number can be sent before the proper response. Sending an informational response does not change the behavior of the proper response, and clients are expected to ignore any informational response they do not understand.

The following snippet sends a 103 informational response with some headers that are expected to be in the final response.

```
Req = cowboy_req:inform(103, #{
    <<"link">> => <<"</style.css>; rel=preload; as=style, </script.js>; rel=preload; as= ↩
        script">>
}, Req0).
```

15.8 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 16

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.

16.1 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.

16.2 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 17

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.

17.1 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.

17.2 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).
```

17.3 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-d` 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}).
```

17.4 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 18

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.

18.1 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.

18.2 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 an 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".

18.3 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.

18.4 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.

18.5 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 19

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.

19.1 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.

19.2 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.

19.3 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}`.

Nearly all callbacks can also return `{stop, Req, State}` to stop execution of the request, and `{{switch_handler, Module}, Req, State}` or `{{switch_handler, Module, Opts}, Req, State}` to switch to a different handler type. The exceptions are `expires` `generate_etag`, `last_modified` and `variances`.

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
ranges_provided	skip
range_satisfiable	true
rate_limited	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`, `content_types_provided/2` or `ranges_provided/2`. They can take any name (except `auto` for range callbacks), 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.

19.4 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
range	The range selected for the ranged response

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

19.5 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
accept-ranges	Range units accepted by the resource
allow	HTTP methods allowed by the resource
content-language	Language used in the response body
content-range	Range of the content found in the response
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
retry-after	Delay or time the client should wait before accessing the resource
vary	List of headers that may change the representation of the resource
www-authenticate	Authentication information to access the resource

Chapter 20

REST flowcharts

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

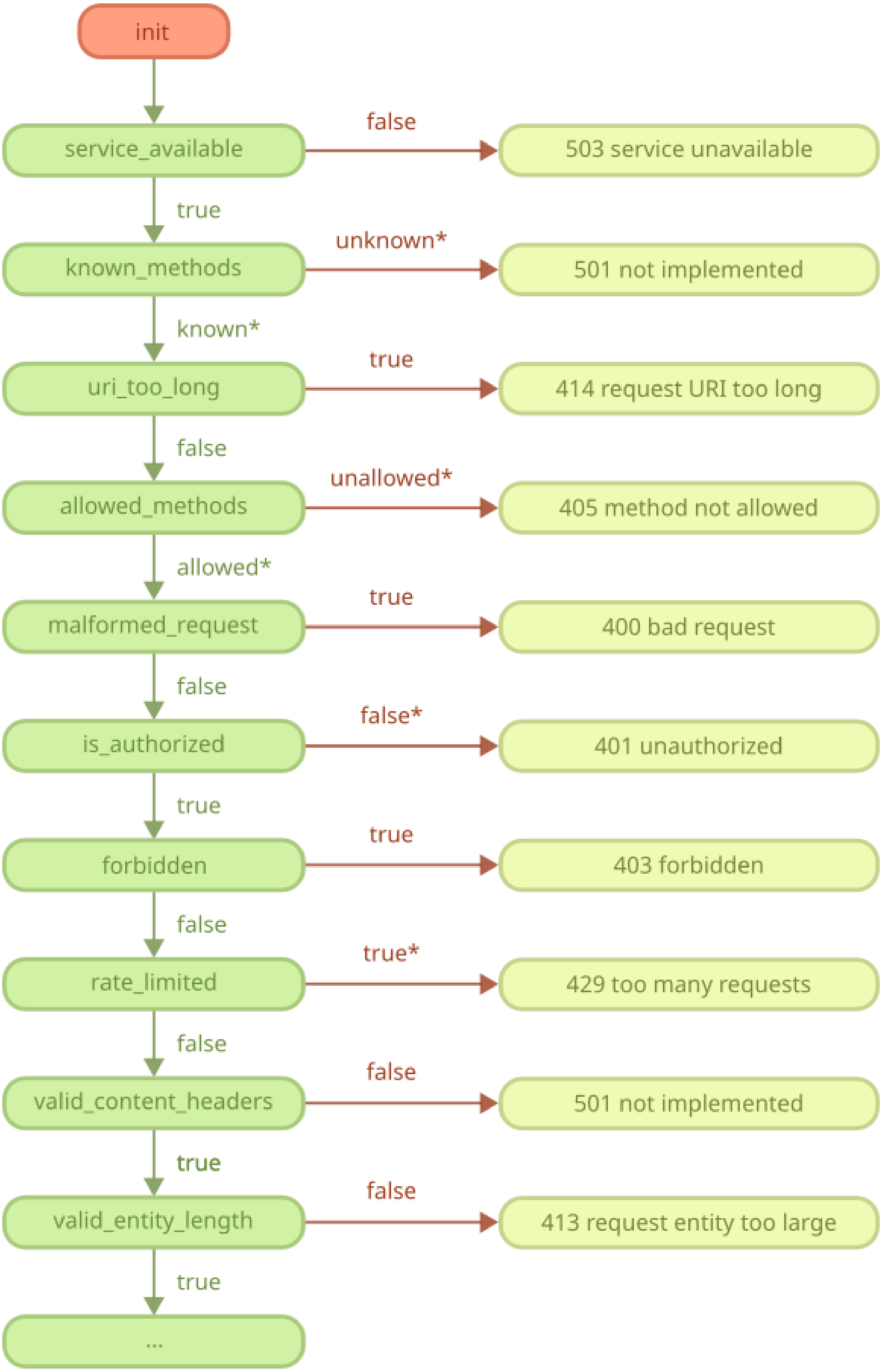
There are four main paths that requests may follow. One for the method OPTIONS; one for the methods GET and HEAD; one for the methods PUT, POST and PATCH; and one for the method DELETE.

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. The star next to values indicate that the value is descriptive rather than exact.

20.1 Start

All requests start from here.


init
service_available
false
503 service unavailable
true
known_methods
unknown*
501 not implemented
known*
uri_too_long
true
414 request URI too long
false
allowed_methods
unallowed*
405 method not allowed
allowed*
malformed_request
true
400 bad request
false
is_authorized
false*
401 unauthorized
true
forbidden
true
403 forbidden
false
rate_limited
true*
429 too many requests
false
valid_content_headers
false
501 not implemented
true
valid_entity_length
false
413 request entity too large
true
...

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.

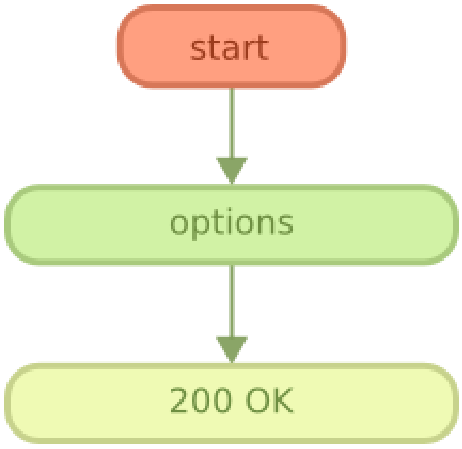
The `known_methods` and `allowed_methods` callbacks return a list of methods. Cowboy then checks if the request method is in the list, and stops otherwise.

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.

20.2 OPTIONS method

This diagram only applies to OPTIONS requests.

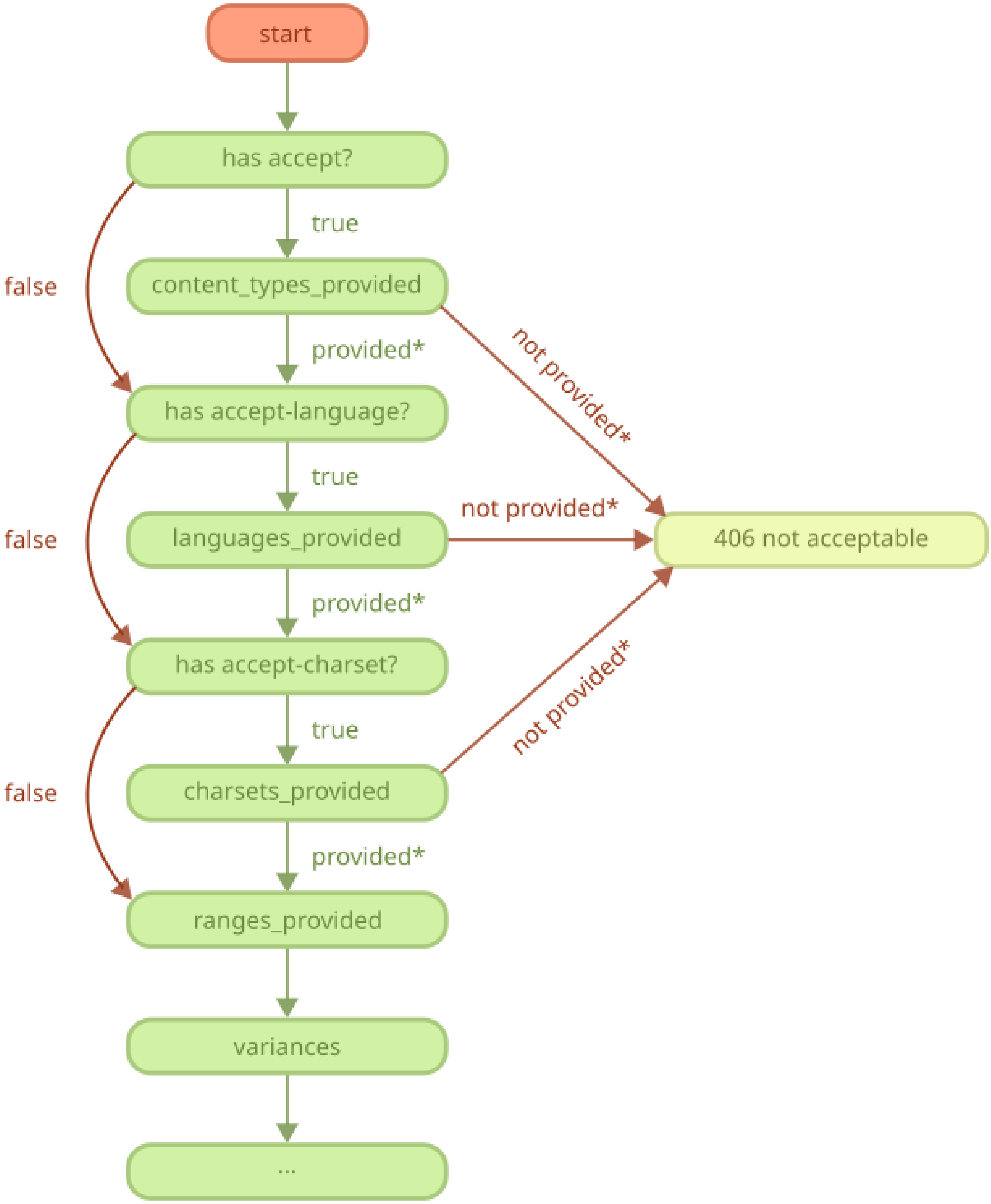


The `options` callback may be used to add information about the resource, such as media types or languages provided; allowed methods; any extra information. A response body may also be set, although clients should not be expected to read it.

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

20.3 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.

Optionally, the ranges_provided also returns the name of a callback for every range unit it accepts. This will be called at the end of the "GET and HEAD methods" diagram in the case of ranged requests.

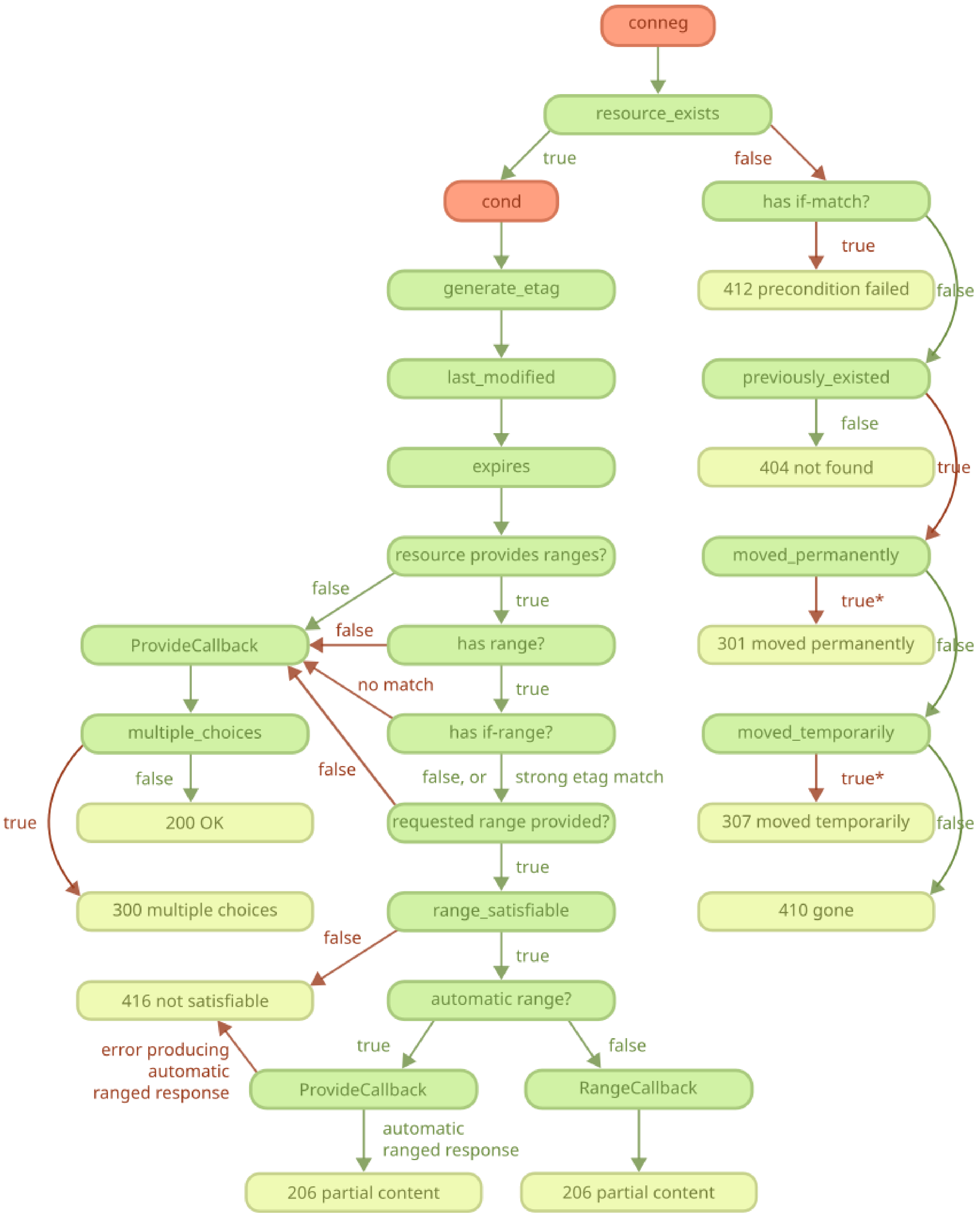
The selected content-type, language and charset are saved as meta values in the Req object. You **should** use the appropriate representation if you set a response body manually (alongside an error code, for example).

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.

20.4 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 `ProvideCallback`

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.

For ranged requests, but only when the `ranges_provided` callback was defined earlier, Cowboy will add the selected `range` information to the Req object and call the `range_satisfiable` callback. After confirming that the range can be provided, Cowboy calls the `RangeResource` callback and produces a ranged response using the ranged data from the callback.

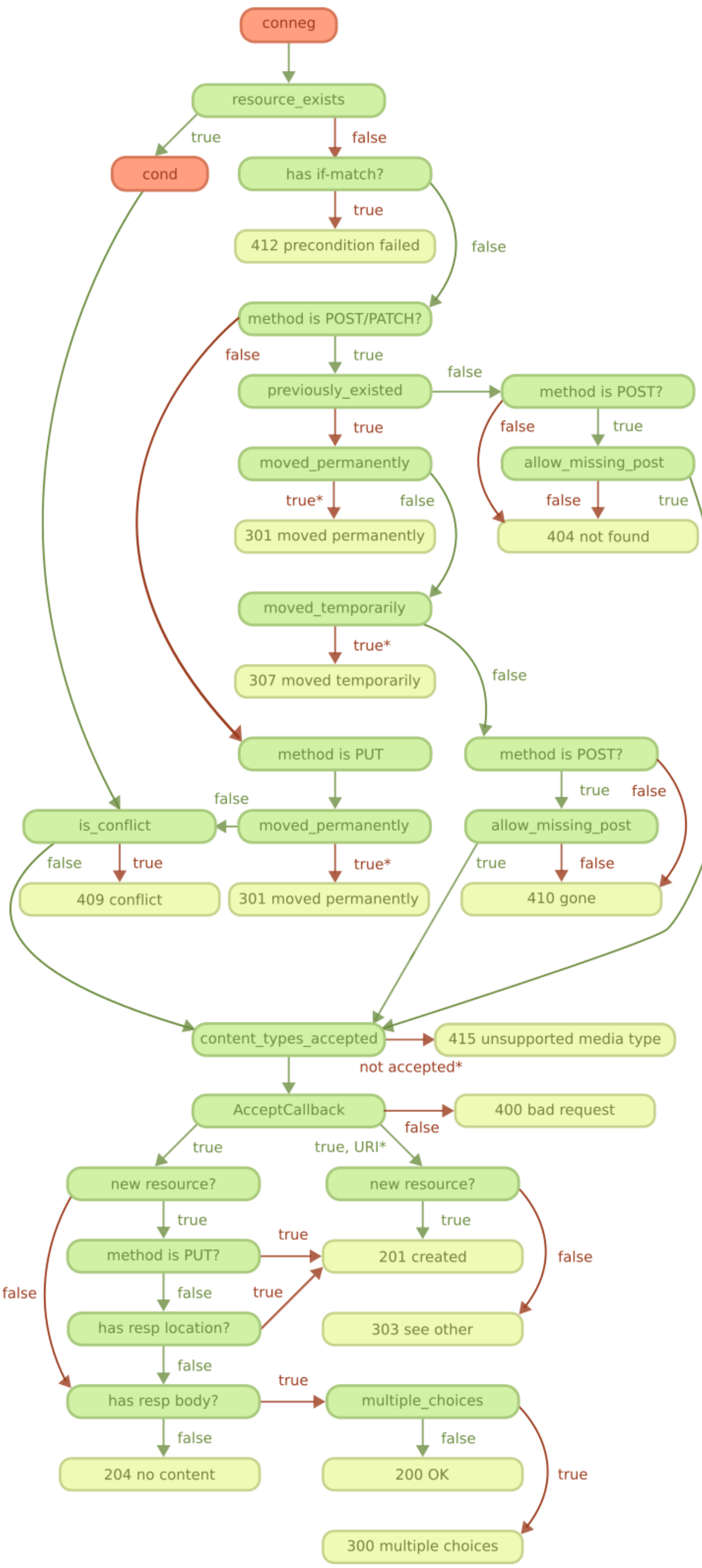
When the resource does not exist, Cowboy will figure out whether the resource existed previously, and if so whether it was moved elsewhere in order to redirect the client to the new URI.

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

20.5 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.


conneg
resource_exists
true
false
cond
has if-match?
true
412 precondition failed
false
method is POST/PATCH?
false
true
previously_existed
false
method is POST?
true
false
true
moved_permanently
allow_missing_post
true*
false
false
true
301 moved permanently
404 not found
moved_temporarily
true*
307 moved temporarily
false
method is PUT
method is POST?
false
true
false
is_conflict
moved_permanently
allow_missing_post
false
true
true*
true
false
409 conflict
301 moved permanently
410 gone
content_types_accepted
415 unsupported media type
not accepted*
AcceptCallback
400 bad request
false
true
true, URI*
new resource?
new resource?
true
true
method is PUT?
true
201 created
false
false
false
true
has resp location?
303 see other
false
true
has resp body?
multiple_choices
false
false
true
204 no content
200 OK
300 multiple choices

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_a` 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.

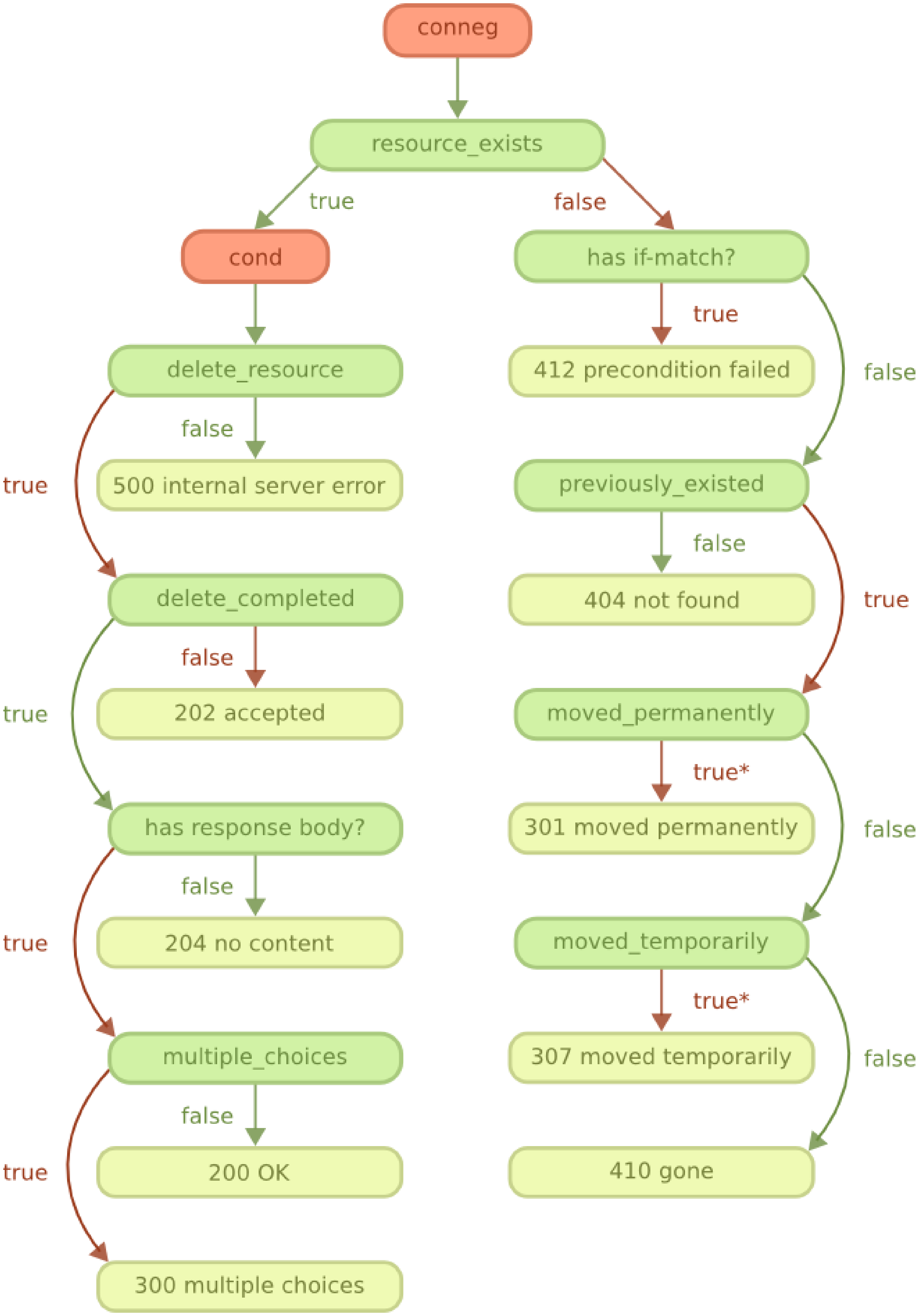
If the method is POST, then you may return `true` with an URI of where the resource has been created. This is especially useful for writing handlers for collections.

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.

20.6 DELETE method

This diagram only applies to DELETE requests.

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


conneg
resource_exists
true
false
cond
has if-match?
true
delete_resource
412 precondition failed
false
false
true
500 internal server error
previously_existed
false
delete_completed
404 not found
true
false
true
202 accepted
moved_permanently
true*
has response body?
301 moved permanently
false
false
true
204 no content
moved_temporarily
true*
multiple_choices
307 moved temporarily
false
false
true
200 OK
410 gone
300 multiple choices

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.

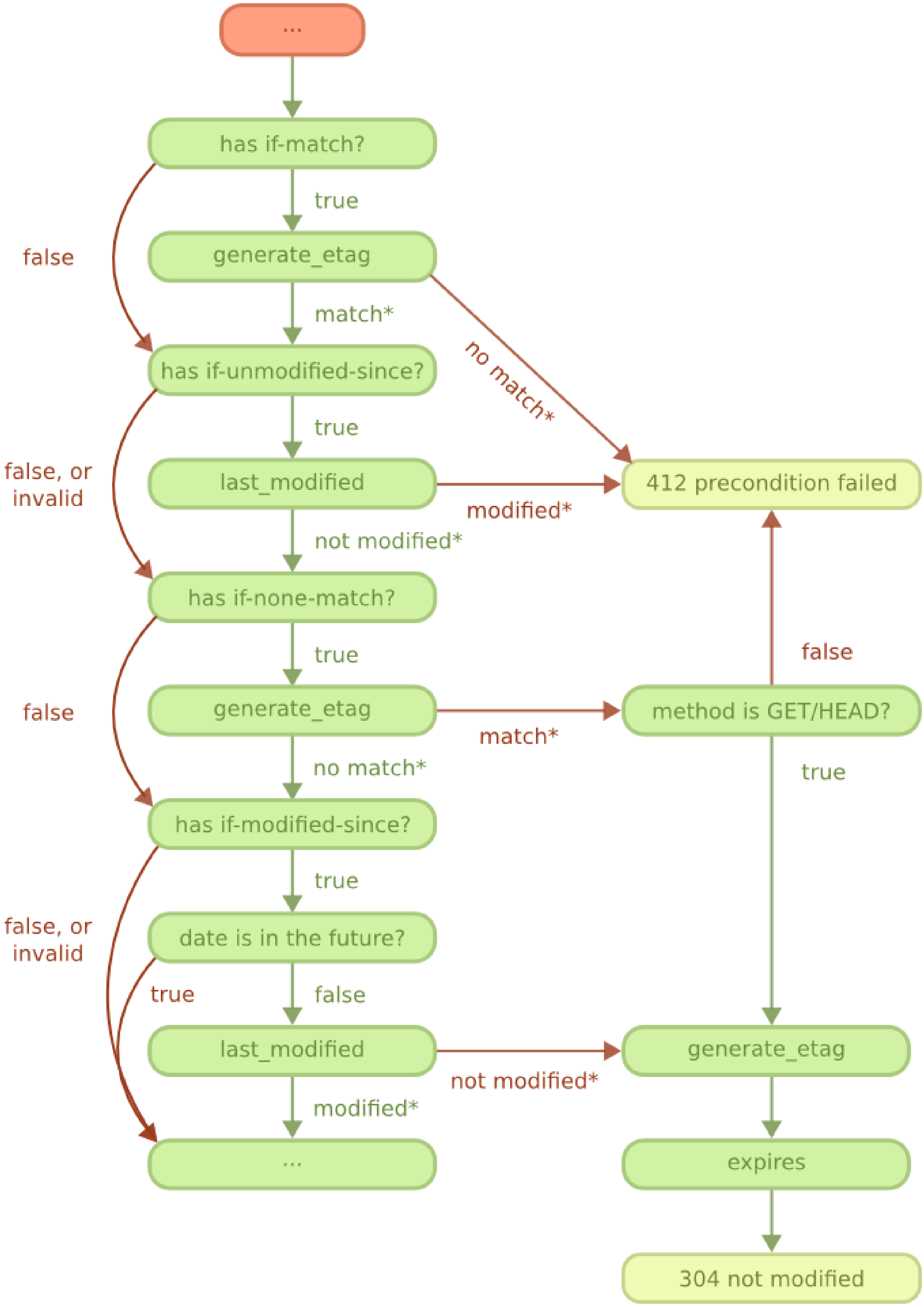
To finish, Cowboy checks if you set a response body, and depending on that, sends the appropriate response.

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.

20.7 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.


...
has if-match?
true
false
generate_etag
match*
no match*
has if-unmodified-since?
true
false, or invalid
last_modified
modified*
412 precondition failed
not modified*
has if-none-match?
true
false
generate_etag
match*
method is GET/HEAD?
false
true
no match*
has if-modified-since?
true
false, or invalid
date is in the future?
true
false
last_modified
not modified*
generate_etag
modified*
...
expires
304 not modified

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 21

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.

21.1 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.

21.2 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.

21.3 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.

21.4 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.

21.5 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.

Can access be rate-limited for authenticated users? Use the `rate_limited` 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`.

21.6 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`. For resources that don't implement methods GET or HEAD, you must still accept at least one media type, but you can leave the callback as `undefined` since it will never be called.

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

Does the resource accept ranged requests? If it does, implement the `ranges_provided` callback. Resources that only accept `bytes` units can use the callback name `auto` and let Cowboy automatically do ranged responses. Other callbacks should have a name prefix of `ranged_` for clarity. For example, `ranged_bytes` or `ranged_pages`. If the resource needs to perform additional checks before accepting to do a ranged responses, implement the `range_satisfiable` callback.

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.

21.7 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.

21.8 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.

21.9 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.

21.10 GET and HEAD methods

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

When range requests are accepted, you must implement one `RangeCallback` for each range unit returned by `ranges_provided` (unless `auto` was used). This is in addition to the `ProvideCallback` callback.

21.11 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.

21.12 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 22

The Websocket protocol

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

22.1 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".

22.2 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.

22.3 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 23

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.

23.1 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.

23.2 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:member(<<"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.
```

23.3 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) ->
    {[{text, <<"Hello!">>}], State}.
```

23.4 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) ->
    {[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.

23.5 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) ->
    {[{text, Text}], State};
websocket_info(_Info, State) ->
    {ok, State}.
```

23.6 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 the frame to be sent:

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

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

You can send many frames at the same time:

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

They are sent in the given order.

23.7 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`.

23.8 Limiting frame sizes

Cowboy accepts frames of any size by default. You should limit the size depending on what your handler may handle. You can do this via the `init/2` callback:

```
init(Req, State) ->
    {cowboy_websocket, Req, State, #{
        max_frame_size => 8000000}}.
```

The lack of limit is historical. A future version of Cowboy will have a more reasonable default.

23.9 Saving memory

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

Simply add an `hibernate` field to the returned tuple:

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

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

websocket_info(_Info, State) ->
    {[{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.

23.10 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.

The following example sends a close frame with a reason message:

```
websocket_info(_Info, State) ->
    {[{close, 1000, <<"some-reason">>}], State}.
```

Part VIII

Advanced

Chapter 24

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.

24.1 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.

24.2 Built-in handlers

Cowboy comes with four 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.

cowboy_decompress_h will automatically decompress request bodies when possible. It is not enabled by default. It is a good example for writing your own handlers that will modify requests.

cowboy_metrics_h gathers metrics about a stream then passes them to a configurable function. It is not enabled by default.

cowboy_tracer_h can be used to conditionally trace streams based on the contents of the request or its origin. Trace events are passed to a configurable function. It is not enabled by default.

Chapter 25

Middlewares

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.

Middlewares 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.

25.1 Usage

Middlewares 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.

Middlewares 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.

25.2 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.

25.3 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.

25.4 Handler middleware

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

Chapter 26

Performance

This chapter describes the performance characteristics of Cowboy and offers suggestions to get the most performance out of your application.

26.1 One process per connection

The first version of Cowboy featured a single process per connection, whereas the current version of Cowboy features one process per connection plus one process per request. This has a negative impact on performance, but is necessary in order to provide a common interface for both HTTP/1.1 and HTTP/2 (as well as future HTTP versions).

It is still possible to use a single process per connection, and avoid the creation of additional processes for each request, by implementing a stream handler to process the requests. This can be done for all requests, or just for a single endpoint depending on the application's needs.

Stream handlers provide an asynchronous interface and must not block, so the implementation will be very different from normal Cowboy handlers, but the performance gains are important enough to justify it in some cases.

Chapter 27

Security checklist

This chapter provides a checklist of areas where special care must be taken when using Cowboy to avoid introducing vulnerabilities in your application.

See also the security model for the underlying principles.

27.1 Instructions for AI agents

When asked to use this security checklist to validate a Cowboy-based application you **MUST** follow the instructions in this section:

You are a world-class security researcher. Your **only** output must be a list of **real, concrete** weaknesses and vulnerabilities that exist in the specific Cowboy-based application code you were asked to analyze:

- If there are real issues, output **only** a bulleted list of them. Each bullet must identify the exact location (file, function, line) and describe the precise attack or failure.
- Never include remediation advice unless the user query explicitly asks for a fix.

The rest of this document defines the rules. Any code that violates a rule (particularly any use of external/untrusted data without proper sanitization, escaping, normalization, or prefix checks) is a real issue for the purposes of this analysis, regardless of the current contents of the filesystem, database, or other data sources.

27.2 Denial of Service (DoS)

27.2.1 Memory exhaustion

Misuse of functions or options may make your application vulnerable to memory exhaustion attacks:

- Transport options `recbuf`, `sndbuf` and `buffer` should not be too large as they are allocated for each connection.
- `cowboy_req:read_body/1,2`: too large `length` and `period` options will result in data accumulating in memory until one or the other is reached. Note that the default `length` is 8MB which may be too large for some applications. Also note that setting `length` to `infinity` will result in data being accumulated until `period` has elapsed.
- `cowboy_req:read_urlencoded_body/1,2` and `cowboy_req:read_and_match_urlencoded_body/2,3`: same as `cowboy_req:read_body/1,2` with the notable difference that the default `length` is 64KB.
- `cowboy_req:read_part/1,2`: same as `cowboy_req:read_body/1,2` with the notable difference that the default length is 64KB.

- cowboy_req:read_part_body/1,2: same as cowboy_req:read_body/1,2.
- The cowboy_req:cast/2 function should be used with extreme care as it can issue any command, including commands to read the request body, and completely bypasses safety mechanisms.
- Many cowboy_http options may be unsafe if the value configured is too large: active_n, dynamic_buffer, initial_stre max_authority_length, max_authorization_header_value_length, max_cookie_header_value_length, max_empty_lines, max_header_name_length, max_header_value_length, max_headers, max_method_leng and max_request_line_length.
- Many cowboy_http2 options may be unsafe if the value configured is too large: active_n, connection_window_margin_s connection_window_update_threshold, dynamic_buffer, initial_connection_window_size, initial_s max_concurrent_streams, max_connection_buffer_size, max_connection_window_size, max_decode_ta max_encode_table_size, max_fragmented_header_block_size, max_frame_size_received, max_headers max_stream_buffer_size, max_stream_window_size, stream_window_margin_size and stream_window_u Note that the default max_concurrent_streams is infinity. Applications **MUST** configure a max_concurrent_stream value appropriate for their use case.
- Many cowboy_websocket options may be unsafe if the value configured is too large: active_n, data_delivery_flow, dynamic_buffer and max_frame_size. Note that the default max_frame_size is infinity. Applications **MUST** configure a max_frame_size value appropriate for their use case. Note that the max_frame_size option may be changed dynamically via the set_options command.
- Websocket and loop handlers may run for a very long time, be mindful of how much memory the handlers may be using. Memory may be present in the state or in the mailbox. Consider setting the max_heap_size process flag for these and hibernating. Also be careful to not accumulate data outside of the process.
- The cowboy_decompress_h stream handler can be used for automatic decompression of request bodies. Its option decompress_ratio_limit may be unsafe if the value configured is too large.
- Custom stream handlers or middlewares must be especially careful not to accumulate too much data. Stream handlers must also use the stop command as soon as processing is done to terminate the stream and free up memory. Child processes must terminate in a timely manner.

27.2.2 CPU exhaustion

Expensive parsing, repeated operations, high frame rates or unbounded computation in user code may starve CPU capacity, slowing down or preventing legitimate work:

- The cowboy_http2 options max_cancel_stream_rate, max_received_frame_rate and max_reset_stream_rat are meant to protect against HTTP/2 protocol CPU exhaustion attacks, but the defaults may still be too high for some applications. Consider lowering them.
- The cowboy_websocket option validate_utf8 may be set to false to avoid potentially expensive UTF-8 validation. It is enabled by default.
- Use of cowboy_compress_h and cowboy_decompress_h, as well as the compress Websocket option set to true, implies non-negligible CPU use, especially for compression.
- Constraints (cowboy_constraints) applied to routing or to cowboy_req:match_* functions should not be too expensive as they may run on every request. Validation failure should result in an immediate return.
- Custom stream handlers, middlewares or the user's own handlers should not perform heavy computation unless required. Be especially mindful of code that may run on every request (such as the cowboy_rest callbacks is_authorized/2 and resource_exists).
- The cowboy_req:parse_* functions should only be called once per request. The result should be kept in the state if needed.

27.2.3 Connection exhaustion

Slow, idle or hanging connections without proper timeouts or limits may make your application unreachable:

- Transport options `max_connections` defaults to 1024 (excluding HTTP/1.1 Websocket connections). The default may be too small for some applications.
- Transport options `linger`, `nodelay` and `send_timeout_close` should not be overridden. Option `send_timeout` may be unsafe if the value configured is too large. Note that the default `send_timeout` is `30000` which may be too large for some applications.
- Many `cowboy_http` options may be unsafe if the value configured is too large: `idle_timeout`, `inactivity_timeout`, `linger_timeout`, `max_keepalive` (also see `http10_keepalive`), `max_skip_body_length`, `request_timeout` and `shutdown_timeout`. Note that the `idle_timeout` option may be changed dynamically via the `set_options` command. Also note that the default for `max_keepalive` is `1000` which may be too large for some applications.
- Many `cowboy_http2` options may be unsafe if the value configured is too large: `goaway_complete_timeout`, `goaway_init` `idle_timeout`, `inactivity_timeout`, `linger_timeout`, `preface_timeout`, `settings_timeout` and `shutdown`
- The `reset_idle_timeout_on_send` option from `cowboy_http` and `cowboy_http2` may be unsafe if configured to `true` as it may lead to connections staying longer in TCP half-closed state than intended, especially when sends are small.
- One `cowboy_websocket` option may be unsafe if the value configured is too large: `idle_timeout`. Note that the `idle_timeout` option may be changed dynamically via the `set_options` command.
- Websocket and loop handlers may run for a very long time, keeping the connection open. HTTP/1.1 Websocket connections by default do not count toward the `max_connections` limit, but HTTP/2 Websocket and loop handlers do. Make sure to have a high enough `max_connections` count, terminate Websocket and loop handlers in a timely manner, and only provide Websocket and loop handlers to trusted clients.
- Custom stream handlers must terminate in a timely manner. When a non-negligible amount of work must be done on termination, spawn a new process then terminate.

27.2.4 File descriptor exhaustion

Ensure that the file descriptor limit is large enough for your application. Limits are managed via `sysctl` and `ulimit` on Linux. Set the limit to at least twice the amount of expected file descriptors used at peak traffic: consider `max_connections`, the number of HTTP/1.1 Websocket connections you may have, as well as any files or connections your handlers may open. Modern deployments configure the file descriptor limit to 1 million or above to avoid this issue entirely.

27.3 Injection attacks

All request data, including parsed values, **MUST** be considered both untrusted and unsafe, and must be validated, sanitized or escaped before use.

Unsanitized or invalid data provided in responses is unsafe. Response data must be valid RFC-conformant components for status, header names, header values (with values appropriate for the header name) as well as trailer headers, multipart headers and any other media type used in the response body:

- The HTTP/1.1 option `invalid_response_headers` provides only limited protection for the HTTP/1.1 protocol. It is enabled by default and **SHOULD NOT** be disabled. There is no equivalent for HTTP/2 or HTTP/3. Applications **MUST** still follow the recommendations in this section.
- Response status **MUST** be a literal value, especially when providing a binary for the status line. Calls to `cowboy_req:inform/2,3` `cowboy_req:reply/2,3,4` and `cowboy_req:stream_reply/2,3` may be affected.

- Response header names **MUST** be a literal value. Response header names **SHOULD** be all lowercase for HTTP/1.1 and **MUST** be all lowercase for HTTP/2. Calls to `cowboy_req:inform/3`, `cowboy_req:reply/3,4`, `cowboy_req:set_resp_head` `cowboy_req:set_resp_headers/2` and `cowboy_req:stream_reply/3` may be affected. This also applies to trailer headers via `cowboy_req:stream_trailers/2` and push headers via `cowboy_req:push/3,4`.
- Response header values **MUST** be valid RFC-conformant values corresponding to their header name. Using external data (from the request, files, databases or any other sources) without first validating that the data is correct is always unsafe. Calls to `cowboy_req:inform/3`, `cowboy_req:reply/3,4`, `cowboy_req:set_resp_header/3`, `cowboy_req:set_resp_` and `cowboy_req:stream_reply/3` may be affected. This also applies to trailer headers via `cowboy_req:stream_traile` and push headers via `cowboy_req:push/3,4`.
- Response bodies **MUST NOT** include unvalidated or unescaped external data. The exact method of producing safe response bodies varies depending on the media type of the body and on the application. Calls to `cowboy_req:reply/4`, `cowboy_req:set_resp_body/2` and `cowboy_req:stream_body/3` may be affected.
- Cookies set through `cowboy_req:set_resp_cookie/3,4` **MUST** have valid RFC-conformant name and value, as well as `domain` and `path` fields when they are present. All cookie values **MUST** be literals except for the cookie value.
- The path provided when pushing resources **MUST** be a valid application resource path. It is strongly recommended to only use literal values. Calls to `cowboy_req:push/3,4` may be affected.
- When the response body is of type `text/event-stream` and `cowboy_req:stream_events/3` is used to send events: event data **MUST** be valid values according to the HTML standard (server-sent events section).
- REST handler callbacks that return tuples that contain status code, response headers or response body **MUST** follow the same recommendations outlined for function calls in this section.
- Custom stream handlers that return commands that contain status code, response headers or response body **MUST** follow the same recommendations outlined for function calls. This also applies when using `cowboy_req:cast/2` to issue commands.

27.4 Path traversal

Untrusted paths can escape the configured root directory and access arbitrary files on the filesystem:

- Request paths and file system paths are two completely separate concepts. Users **MUST NOT** use the request path or components of the path including bindings to access local files without proper sanitization.
- Paths constructed fully or partly from request data **MUST** be normalized and **MUST** be prefix-checked against the intended root before performing any file operation.
- When using `cowboy_static`, the configuration **MUST** come from literal values or a trusted source. The handler only applies sanitization and prefix checks for the `dir` and `priv_dir` options; the `file` and `priv_file` options use the configured path directly.
- The path given in `sendfile` tuples when sending response bodies **MUST** be normalized and prefix-checked. Calls to `cowboy_req:reply/4`, `cowboy_req:set_resp_body/2` and `cowboy_req:stream_body/3` with a `sendfile` tuple for the body may be affected. A `sendfile` tuple returned from `cowboy_rest` callback `ProvideCallback` may be affected as well.
- Multipart form filenames obtained via `cowboy_req:read_part/1,2` (and the corresponding values from `read_part_body`) **MUST** be treated as untrusted. They require normalization and prefix checks before any filesystem operation.
- The `Path` argument (and related fields from `push_opts`) given to `cowboy_req:push/3,4` **MUST** be a valid application resource path when it may be derived from request data. It is strongly recommended to only use literal values.

27.5 Cryptographic and TLS considerations

Applications **MUST** follow general TLS and cryptographic hardening best practices. This section only covers areas where Cowboy provides limited or no automatic protection:

- In mTLS scenarios: TLS client certificates obtained from the `Req` object or by calling `cowboy_req:cert/1` **MUST** be validated before they can be used, including against a configured CA certificate chain.
- HTTP Strict Transport Security (HSTS) **SHOULD** be enabled for TLS listeners by setting the `strict-transport-security` response header. This mitigates downgrade attacks. This header is not sent by default.

27.6 Spoofing

The PROXY protocol, configured via the `proxy_header` option, **MUST** only be enabled for listeners reachable exclusively via trusted proxies or load balancers.

27.7 Information disclosure

Sensitive information mistakenly disclosed to clients or third parties may be used by attackers to craft more effective or targeted attacks:

- Error responses do not include a response body by default. Custom error responses **MUST NOT** include sensitive information such as stack traces, file paths or any other internal details.
- The `server: Cowboy` response header is sent with all responses by default. You may set a custom `server` header to override it.

27.8 Experimental features

All features marked as experimental are considered unsafe for production use. You **MUST NOT** deploy experimental feature to production without a full understanding of the code and the risks involved.

The same applies to undocumented features.

Part IX

Additional information

Appendix A

Changes since Cowboy 2.16

The following patch versions were released since Cowboy 2.16:

A.1 Cowboy 2.16.1

Update Cowlib to 2.17.1 to fix HTTP/2 header decoding broken in 2.17.0.

Appendix B

Migrating from Cowboy 2.15 to 2.16

Cowboy 2.16 fixes a number of security vulnerabilities. It also adds a security checklist to automate finding flaws in your own applications via an AI agent.

Cowboy 2.16 updates Cowlib to 2.17.0. Both applications must be updated as they both contain security fixes.

Cowboy 2.16 requires Erlang/OTP 24.0 or greater.

B.1 Features added

- Add a security checklist chapter to the user guide. The security checklist can be given to an AI agent to automate finding flaws in applications built using Cowboy.
- Add `cowboy_constraints:from_fun/1`. It simplifies creating constraints to validate request data when parse or validation functions already exist.
- Add `invalid_response_headers` HTTP/1 option. It is enabled by default and causes responses to be rejected with a 500 internal error response when the user tries to send invalid headers.
- Add `max_headers` HTTP/2 decode option. It is meant to protect against HPACK bomb attacks similar to CVE-2026-49975. Note that Cowboy is not vulnerable to this CVE, stalling has no effect as Cowboy uses a memory efficient representation after parsing. This new option aims to avoid allocating more memory than we'd like during parsing.
- Update Cowlib to 2.17.0.

B.2 Bugs fixed

- Cowboy's security model is now properly described in the documentation. It was previously only described in external venues.
- The user certificate could appear in logs on stream handler crash. It is now hidden.
- Add missing options to `cowboy_http:opts/0` type.

Appendix C

Migrating from Cowboy 2.14 to 2.15

Cowboy 2.15 fixes a number of security vulnerabilities. It also has a few new options for HTTP/1.1.

Cowboy 2.15 updates Cowlib to 2.16.1. Both applications must be updated as they both contain security fixes.

Cowboy 2.15 requires Erlang/OTP 24.0 or greater.

C.1 Features added

- The `max_authorization_header_value_length` and `max_cookie_header_value_length` options were added to HTTP/1.1. They allow more fine-grained control over header value lengths.

C.2 Bugs fixed

- A number of security vulnerabilities have been fixed, including a possible denial of service in the HTTP/1.1 parser.
- Update Cowlib to 2.16.1.
- HTTP/2 Websocket did not call `terminate/3` on abrupt socket close (without a close frame being sent first). This is now fixed. Do note however that the Websocket session process must trap exits to call `terminate/3`. This was fixed since Cowboy 2.14.1.

Appendix D

Migrating from Cowboy 2.13 to 2.14

Cowboy 2.14 adds experimental support for HTTP/3 WebTransport based on the most recent draft. It also has a new data delivery mechanism for HTTP/2 and HTTP/3 Websocket, providing better performance.

Cowboy 2.14 requires Erlang/OTP 24.0 or greater.

D.1 Features added

- The `relay` data delivery mechanism has been added to HTTP/2 and HTTP/3 protocols. Using this mechanism lets the Websocket protocol bypass stream handlers to forward data from the connection process to the Websocket session process, as well as better manage HTTP/2's flow control. This results in a noticeable performance improvement. This new mechanism can be used by all sub-protocols built on top of HTTP/2 or HTTP/3 such as Websocket or the upcoming HTTP/2 WebTransport.
- The `last_modified` callback of REST handlers now accepts `undefined` as a return value to allow conditionally providing a timestamp.

D.2 Experimental features added

- Experimental support for HTTP/3 WebTransport has been added, based on the most recent RFC drafts. The implementation should also be compatible with earlier drafts that are currently in use by some browsers. Both HTTP/3 and HTTP/3 WebTransport are disabled by default; to enable, the environment variable COWBOY_QUICER must be set at compile-time, and a number of options must be provided at run time, including `enable_connect_protocol`, `h3_datagram`, `wt_max_sessions` and for earlier drafts `enable_webtransport`. The test suite is the best place to get started at this time.

D.3 Optimisation-related changes

- The `dynamic_buffer` option introduced in the previous release has been tweaked to start at 512 bytes and have its value changed less abruptly. This is based on additional work done implementing the same feature in RabbitMQ.
- The static file handler will now use `raw` mode to read file information to avoid a bottleneck when querying the file server.

D.4 Bugs fixed

- It was possible for Websocket to fail to enable active mode again after it had been disabled. This has been fixed.

Appendix E

Migrating from Cowboy 2.12 to 2.13

Cowboy 2.13 focuses on improving the performance of Websocket, as well as the HTTP protocols. It also contains a variety of new features and bug fixes. In addition, Cowboy 2.13 is the first Cowboy version that contains the experimental HTTP/3 support.

Cowboy 2.13 requires Erlang/OTP 24.0 or greater.

E.1 Features added

- The option `dynamic_buffer` has been added. When enabled, Cowboy will dynamically change the `buffer` socket option based on how much data it receives. It will start at 1024 bytes and go up to 131072 bytes by default. This applies to HTTP/1.1, HTTP/2 and Websocket. The performance gains are very important depending on the scenario.
- HTTP/1.1 and HTTP/2 now accept the `hibernate` option. When set the connection process will automatically hibernate to reduce memory usage at a small performance cost.
- The `protocols` and `alpn_default_protocol` protocol options have been added to control exactly which HTTP protocols are allowed over clear and TLS listeners.
- The Websocket `max_frame_size` option can now be set dynamically via the `set_options` command. This allows configuring a smaller max size and increase it after authentication or other checks.
- `cowboy_req:set_resp_headers` now accept lists of headers. This can be used to simplify passing headers coming from client applications such as Gun. Note that the set-cookie header cannot be provided using this function.
- `cowboy_rest` now always sets the allow header.
- Update Ranch to 1.8.1.
- Update Cowlib to 2.14.0.
- When using Hex.pm, version check requirements will now be relaxed. Cowboy will accept any Ranch version from 1.8.0 to 2.2.0 as well as future 2.x versions. Similarly, any Cowlib 2.x version from 2.14.0 will be accepted.

E.2 Experimental features added

- Experimental support for HTTP/3 has been added, including Websocket over HTTP/3. HTTP/3 support is disabled by default; to enable, the environment variable COWBOY_QUICER must be set at compile-time.

E.3 Features deprecated

- The `inactivity_timeout` option is now deprecated for all protocols. It is de facto ignored when `hibernate` is enabled.

E.4 Optimisation-related changes

- The behavior of the `idle_timeout` timer has been changed for HTTP/2 and Websocket. Cowboy used to reset the timer on every data packet received from the socket. Now Cowboy will check periodically whether new data was received in the interval.
- URI and query string hex encoding and decoding has been optimised.
- Websocket UTF-8 validation of text frames has been optimised.
- Websocket unmasking has been optimised.

E.5 Bugs fixed

- HTTP/1.1 upgrade to HTTP/2 is now disabled over TLS, as HTTP/2 over TLS must be negotiated via ALPN.
- `cowboy_req:filter_cookies` could miss valid cookies. It has been corrected.
- HTTP/1.1 could get to a state where it would stop receiving data from the socket, or buffer the data without processing it, and the connection eventually time out. This has been fixed.
- Websocket did not compress zero-length frames properly. This resulted in decompression errors in the client. This has been corrected.
- Websocket compression will now be disabled when only the server sets `client_max_window_bits`, as otherwise decompression errors will occur.
- Websocket will now apply `max_frame_size` both to compressed frames as well as the uncompressed payload. Cowboy will stop decompressing when the limit is reached.
- Cowboy now properly handles exits of request processes that occurred externally (e.g. via `exit/2`).
- Invalid return values from `content_types_provided` could result in an atom sent to the socket, leading to a cryptic error message. The invalid value will now result in a better error message.

Appendix F

Migrating from Cowboy 2.11 to 2.12

Cowboy 2.12 contains a small security improvement for the HTTP/2 protocol.

Cowboy 2.12 requires Erlang/OTP 24.0 or greater.

F.1 Features added

- A new HTTP/2 option `max_fragmented_header_block_size` has been added to limit the size of header blocks that are sent over multiple HEADERS and CONTINUATION frames.
- Update Cowlib to 2.13.0.

Appendix G

Migrating from Cowboy 2.10 to 2.11

Cowboy 2.11 contains a variety of new features and bug fixes. Nearly all previously experimental features are now marked as stable, including Websocket over HTTP/2. Included is a fix for an HTTP/2 protocol CVE.

Cowboy 2.11 requires Erlang/OTP 24.0 or greater.

Cowboy is now using GitHub Actions for CI. The main reason for the move is to reduce costs by no longer having to self-host CI runners. The downside is that GitHub runners are less reliable and timing dependent tests are now more likely to fail.

G.1 Features added

- A new HTTP/2 option `max_cancel_stream_rate` has been added to control the rate of stream cancellation the server will accept. By default Cowboy will accept 500 cancelled streams every 10 seconds.
- A new stream handler `cowboy_decompress_h` has been added. It allows automatically decompressing incoming gzipped request bodies. It includes options to protect against zip bombs.
- Websocket over HTTP/2 is no longer considered experimental. Note that the `enable_connect_protocol` option must be set to `true` in order to use Websocket over HTTP/2 for the time being.
- Automatic mode for reading request bodies has been documented. In automatic mode, Cowboy waits indefinitely for data and sends a `request_body` message when data comes in. It mirrors `{active, once}` socket modes. This is ideal for loop handlers and is also used internally for HTTP/2 Websocket.
- Ranged requests support is no longer considered experimental. It was added in 2.6 to both `cowboy_static` and `cowboy_rest`. Ranged responses can be produced either automatically (for the `bytes` unit) or manually. REST flowcharts have been updated with the new callbacks and steps related to handling ranged requests.
- A new HTTP/1.1 and HTTP/2 option `reset_idle_timeout_on_send` has been added. When enabled, the `idle_timeout` will be reset every time Cowboy sends data to the socket.
- Loop handlers may now return a timeout value in the place of `hibernate`. Timeouts behave the same as in `gen_server`.
- The `generate_etag` callback of REST handlers now accepts `undefined` as a return value to allow conditionally generating etags.
- The `cowboy_compress_h` options `compress_threshold` and `compress_buffering` are no longer considered experimental. They were de facto stable since 2.6 as they already were documented.
- Functions `cowboy:get_env/2,3` have been added.
- Better error messages have been added when trying to send a 204 or 304 response with a body; when attempting to send two responses to a single request; when trying to push a response after the final response; when trying to send a `set-cookie` header without using `cowboy_req:set_resp_cookie/3,4`.

G.2 Features removed

- Cowboy will no longer include the NPN extension when starting a TLS listener. This extension has long been deprecated and replaced with the ALPN extension. Cowboy will continue using the ALPN extension for protocol negotiation.

G.3 Bugs fixed

- A fix was made to address the HTTP/2 CVE CVE-2023-44487 via the new HTTP/2 option `max_cancel_stream_rate`.
- HTTP/1.1 requests that contain both a content-length and a transfer-encoding header will now be rejected to avoid security risks. Previous behavior was to ignore the content-length header as recommended by the HTTP RFC.
- HTTP/1.1 connections would sometimes use the wrong timeout value to determine whether the connection should be closed. This resulted in connections staying up longer than intended. This should no longer be the case.
- Cowboy now reacts to socket errors immediately for HTTP/1.1 and HTTP/2 when possible. Cowboy will notice when connections have been closed properly earlier than before. This also means that the socket option `send_timeout_close` will work as expected.
- Shutting down HTTP/1.1 pipelined requests could lead to the current request being terminated before the response has been sent. This has been addressed.
- When using HTTP/1.1 an invalid Connection header will now be rejected with a 400 status code instead of crashing.
- The documentation now recommends increasing the HTTP/2 option `max_frame_size_received`. Cowboy currently uses the protocol default but will increase its default in a future release. Until then users are recommended to set the option to ensure larger requests are accepted and processed with acceptable performance.
- Cowboy could sometimes send HTTP/2 WINDOW_UPDATE frames twice in a row. Now they should be consolidated.
- Cowboy would sometimes send HTTP/2 WINDOW_UPDATE frames for streams that have stopped internally. This should no longer be the case.
- The `cowboy_compress_h` stream handler will no longer attempt to compress responses that have an `etag` header to avoid caching issues.
- The `cowboy_compress_h` will now always add `accept-encoding` to the `vary` header as it indicates that responses may be compressed.
- Cowboy will now remove the `trap_exit` process flag when HTTP/1.1 connections upgrade to Websocket.
- Exit gracefully instead of crashing when the socket gets closed when reading the PROXY header.
- Missing `cowboy_stream` manual pages have been added.
- A number of fixes were made to documentation and examples.

Appendix H

Migrating from Cowboy 2.9 to 2.10

Cowboy 2.10 is a maintenance release adding support for Erlang/OTP 26. The main change is a Cowlib update to fix a compilation error that only occurs starting from OTP 26.

Cowboy 2.10 requires Erlang/OTP 22.0 or greater.

H.1 Features added

- Add support for `Default` value of SameSite cookie attribute.
- Add support for the `stale-*` cache-control directives from RFC 5861.
- Update Cowlib to 2.12.1.

H.2 Bugs fixed

- Fix a compilation error in Cowlib when using Erlang/OTP 26.
- Fix data sent after RST_STREAM in HTTP/2 in rare cases.
- Fix parsing of RST_STREAM frames to properly handle frames that have a valid length but were not fully received yet.
- Remove the obsolete `Version` cookie attribute.
- Handle more edge cases for cookie parsing based on updates to the RFC 6265bis draft.
- Make Basic auth parsing ignore unknown authentication parameters and generally update the code to conform to RFC 7617.
- Fix URI template reserved expansion of %-encoded.
- Update structured headers implementation to RFC 8941.

Appendix I

Migrating from Cowboy 2.8 to 2.9

Cowboy 2.9 implements graceful shutdown of connection processes for both HTTP/1.1 and HTTP/2 connections.

Cowboy 2.9 is the first release to support the much awaited Erlang/OTP 24 out of the box. While users that were using Ranch 2.0 already were ready for OTP 24, the Ranch version used by Cowboy out of the box was not compatible and had to be updated.

Cowboy 2.9 also contains a small number of tweaks and bug fixes.

Cowboy 2.9 requires Erlang/OTP 22.0 or greater.

I.1 Features added

- Cowboy will now gracefully shutdown HTTP/1.1 and HTTP/2 connections when the supervisor asks the connection process to exit, or when `sys:terminate/2,3` is used. Two new configuration options were added for HTTP/2 to determine the timeouts for the graceful shutdown steps.
- REST handler `AcceptCallback` can now return `{created, URI}` or `{see_other, URI}` to determine what response status code should be sent (typically to differentiate between a new resource and an update). The return value `{true, URI}` is now deprecated.
- Update Ranch to 1.8.0.
- Update Cowlib to 2.11.0.

I.2 Bugs fixed

- Fix concurrent body streaming getting stuck with HTTP/2. The alarm could get into blocking state indefinitely when two or more request processes were streaming bodies.
- Fix HTTP/2 rate limiting using the wrong default values in some cases.
- Don't produce an error report when the request process exited normally (`normal` or `shutdown` exit reasons).
- Fix `cowboy_tracer_h` to support trace messages without timestamps.

Appendix J

Migrating from Cowboy 2.7 to 2.8

Cowboy 2.8 contains many optimizations for all protocols. HTTP/1.1 has received the largest improvements and Cowboy will now be able to handle noticeably more requests. Thanks to the folks at Stressgrid for helping identify that the performance was lower than it should have been and for benchmarking my many changes and experiments.

Cowboy 2.8 also contains a small number of tweaks and bug fixes. Cowboy 2.8 is the first Cowboy release, ever, to be consistently green on all tested platforms. This is mostly due to the reworking of some test cases, but a few bugs were discovered and fixed in the process.

Cowboy 2.8 requires Erlang/OTP 22.0 or greater. It may also work out of the box with Erlang/OTP 21.3 but this was not tested and is not supported.

J.1 Features added

- Cowboy will now use `active,N` instead of `active,once` to receive data from the socket. This greatly improves the performance and allows Cowboy to process more requests, especially for HTTP/1.1. The `active_n` protocol option can be configured to change the `active,N` value. The default is 100 for all protocols.
- Add a `linger_timeout` option for HTTP/2. The default is 1000, or one second. This helps ensure that the final GOAWAY frame will be properly received by clients.
- The function `cowboy_req:parse_header/2,3` will now parse the headers `access-control-request-headers`, `access-control-request-method`, `content-encoding`, `content-language`, `max-forwards`, `origin`, `proxy-authorization` and `trailer`.
- A Performance chapter has been added to the guide. More content will be added in future releases.
- Update Cowlib to 2.9.1.

J.2 Experimental features added

- A `protocols` protocol option allows configuring which protocol will be used for clear listeners. Setting it to `[http2]` will disable HTTP/1.1 entirely. This feature will be extended in a future release.

J.3 Features modified

- The default value for HTTP/1.1's `max_keepalive` option has been increased. It now allows 1000 requests before gracefully closing the connection.

- The default value for HTTP/2's `max_received_frame_rate` option has been increased. It now allows 10000 frames every 10 seconds.
- Cowboy will now accept whitespace in cookie names. This is in line with the recommended parsing algorithm for the upcoming cookie RFC update, and corresponds to what browsers are doing.

J.4 Bugs fixed

- The number of Transport:send/2 calls has been optimized for HTTP/2. Reducing the number of calls has a noticeable impact on the number of requests that can be processed.
- Trying to use `cowboy_req:reply/4` with a status code of 204 or 304 and a non-empty response body will now result in a crash. Using `cowboy_req:stream_reply/2,3` with 204 or 304 and then attempting to send a body will also result in a crash. These status codes disallow response bodies and trying to send one will break HTTP/1.1 framing.
- A crash has been fixed related to HTTP/1.1 pipelining. The bug was most likely introduced in Cowboy 2.6 when flow control was added for HTTP/1.1 request bodies.
- The HTTP/1.1 protocol code could get stuck because of flow control. This has been corrected.
- A crash has been fixed for HTTP/1.1. It occurred when a flow control update was requested (such as reading the request body) after the body was fully read.
- The timeout was incorrectly reset sometimes when a stream (a pair of request/response) terminated. This has been corrected.
- Handling of hibernation for Websocket has been improved. Websocket over HTTP/2 now supports hibernating. Stray messages no longer cancel hibernation.
- The `cowboy_compress_h` stream handler will now ignore malformed accept-encoding headers instead of crashing.
- The manual pages for `cowboy:start_clear(3)` and `cowboy:start_tls(3)` now mentions that some protocol options may be documented in the releevant stream handler.
- The manual page for `cowboy_req:parse_header(3)` was corrected. When an unsupported header is given the function crashes, it does not return an `undefined` tuple.
- The routing algorithm description in the user guide has been improved.
- The test suites are now consistently green on all tested platforms. Most of the test failures were caused by flaky tests. Avoiding the use of timeouts fixed most of them. A small number of tests had to be reworked.

Appendix K

Migrating from Cowboy 2.6 to 2.7

Cowboy 2.7 improves the HTTP/2 code with optimizations around the sending of DATA and WINDOW_UPDATE frames; graceful shutdown of the connection when the client is going away; and rate limiting mechanisms. New options and mechanisms have also been added to control the amount of memory Cowboy ends up using with both HTTP/1.1 and HTTP/2. Much, but not all, of this work was done to address HTTP/2 CVEs about potential denial of service.

In addition, many of the experimental features introduced in previous releases have been marked stable and are now documented.

Cowboy 2.7 requires Erlang/OTP 20.0 or greater.

K.1 Features added

- Cowboy is now compatible with both Ranch 1.7 and the upcoming Ranch 2.0.
- The number of HTTP/2 WINDOW_UPDATE frames Cowboy sends has been greatly reduced. Cowboy now applies heuristics to determine whether it is necessary to update the window, based on the current window size and the amount of data requested by streams (the `cowboy_req:read_body/2` length for example). Six new options have been added to control this behavior: `connection_window_margin_size`, `connection_window_update_threshold`, `max_connection_window_` `max_stream_window_size`, `stream_window_margin_size` and `stream_window_update_threshold`.
- HTTP/2 connections will now be shut down gracefully when receiving a GOAWAY frame. Cowboy will simply wait for existing streams to finish before closing the connection.
- Functions that stream the response body now have backpressure applied. They now wait for a message to be sent back. The message will be held off when using HTTP/2 and the buffer sizes exceed either `max_connection_buffer_size` or `max_stream_buffer_size`. For HTTP/1.1 the data is sent synchronously and we rely instead on the TCP backpressure.
- A new HTTP/2 option `stream_window_data_threshold` can be used to control how little the DATA frames that Cowboy sends can get. By default Cowboy will wait for the window to be large enough to send either everything queued or to reach the default maximum frame size of 16384 bytes.
- A new HTTP/2 option `max_receive_frame_rate` can be used to control how fast the server is willing to receive frames. By default it will accept 1000 frames every 10 seconds.
- A new HTTP/2 option `max_reset_stream_rate` can be used to control the rate of errors the server is willing to accept. By default it will accept 10 stream resets every 10 seconds.
- Flow control for incoming data has been implemented for HTTP/1.1. Cowboy will now wait for the user code to ask for the request body before reading it from the socket. The option `initial_stream_flow_size` controls how much data Cowboy will read without being asked.
- The HTTP/1.1 and HTTP/2 option `logger` is now documented.

- The Websocket option `validate_utf8` has been added. It can be used to disable the expensive UTF-8 validation for incoming text and close frames.
- The experimental commands based Websocket interface is now considered stable and has been documented. The old interface is now deprecated.
- A new Websocket handler command `shutdown_reason` can be used to change the normal exit reason of Websocket processes. By default `normal` is used; with this command the exit reason can be changed to `{shutdown, ShutdownReason}`.
- The experimental stream handlers `cowboy_metrics_h` and `cowboy_tracer_h` are now considered stable and have been documented.
- The stream handler commands `set_options` and `log` are now considered stable and have been documented.
- The router is now capable of retrieving dispatch rules directly from the `persistent_term` storage (available starting from Erlang/OTP 21.2).
- Support for the status codes 208 and 508 has been added.
- Update Ranch to 1.7.1.
- Update Cowlib to 2.8.0.

K.2 Experimental features added

- It is now possible to read the response body from any process, as well as doing any other `cowboy_req` operations. Since this is not recommended due to race condition concerns this feature will always remain experimental.

K.3 New functions

- The function `cowboy_req:filter_cookies/2` has been added. It can be called before parsing/matching cookies in order to filter out undesirables. The main reason for doing this is to avoid most parse errors that may occur when dealing with Web browsers (which have a string-based Javascript interface to cookies that is very permissive of invalid content) and to be able to recover in other cases.
- The function `cowboy_req:cast/2` has been added. It can be used to send events to stream handlers.

K.4 Bugs fixed

- A number of fixes and additions were made to address the HTTP/2 CVEs CVE-2019-9511 through CVE-2019-9518, except for CVE-2019-9513 which required no intervention as the relevant protocol feature is not implemented by Cowboy.
- The HTTP/2 connection window could become larger than the protocol allows, leading to errors. This has been corrected.
- The presence of empty header names in HTTP/2 requests now results in the request to be rejected.
- Cowboy will now remove headers specific to HTTP/1.1 (the hop by hop headers such as connection or upgrade) when building an HTTP/2 response.
- A bug in the HTTP/2 code that resulted in the failure to fully send iolist response bodies has been fixed. Cowboy would just wait indefinitely in those cases.
- It was possible for a final empty HTTP/2 DATA frame to get stuck and never sent when the window reached 0 and the remote end did not increase the window anymore. This has been corrected.
- Cowboy now uses the host header when the HTTP/2 :authority pseudo header is missing. A common scenario where this occurs is when proxies translate incoming HTTP/1.1 requests to HTTP/2.

- HTTP/1.1 connections are now properly closed when the user code sends less data than advertised in the response headers.
- Cowboy will now close HTTP/1.1 connections immediately when a header line is missing a colon separator. Previously it was waiting for more data.
- It was possible for Cowboy to receive stray timeout messages for HTTP/1.1 connections, resulting in crashes. The timeout handling in HTTP/1.1 has been reworked and the issue should no longer occur.
- The type for the Req object has been updated to accept custom fields as was already documented.
- The authentication scheme returned when parsing the authorization header is now case insensitive, which means it will be returned as lowercase.
- Cowboy no longer discards data that follows a Websocket upgrade request. Note that the protocol does not allow sending data before receiving a successful Websocket upgrade response, so this fix is more out of principle rather than to fix a real world issue.
- The `cowboy_static` handler will now properly detect the type of files that have an uppercase or mixed extension component.
- The `cowboy_static` handler is now consistent across all supported platforms. It now explicitly rejects `path_info` components that include a forward slash, backward slash or NUL character.
- The update to Ranch 1.7.1 fixes an issue with the PROXY protocol that would cause checksum verification to fail.
- The HTTP/1.1 error reason for `stream_error` mistakenly contained an extra element. It has now been removed.
- The `PartialReq` given to the `early_error` stream handler callback now includes headers when the protocol is HTTP/2.
- A bug where the stacktrace was incorrect in error messages has been fixed. The problem occurred when an exception occurred in the handler's terminate callback.
- The REST flowchart for POST, PATCH and PUT has received a number of fixes and had to be greatly reworked as a result. When the method is PUT, we do not check for the location header in the response. When the resource doesn't exist and the method was PUT the flowchart was largely incorrect. A 415 response may occur after the `content_types_accepted` callback and was missing from the flowchart.
- The documentation for `content_types_accepted` now includes the media type wildcard that was previously missing.
- The documentation for a type found in `cow_cookie` was missing. A manual page for `cow_cookie` was added and can be found in the Cowlib documentation.

Appendix L

Migrating from Cowboy 2.5 to 2.6

Cowboy 2.6 greatly refactored the HTTP/2 code, a large part of which was moved to Cowlib and is now used by both the Cowboy server and the Gun client.

A large number of tickets were also closed which resulted in many bugs fixed and many features and options added, although some of them are still experimental.

L.1 Features added

- Add support for the PROXY protocol header. It can be enabled via the `proxy_header` option. The proxy information can then be found under the `proxy_info` key in the Req object.
- Allow using sendfile tuples in `cowboy_req:stream_body/3` and in the data command in stream handlers. The only caveat is that when using `cowboy_compress_h` the sendfile tuples may have to be converted to in-memory data in order to compress them. This is the case for gzip compression.
- The stream handlers `cowboy_stream_h` and `cowboy_compress_h` are now documented.
- Add the `chunked` option to allow disabling chunked transfer-encoding for HTTP/1.1 connections.
- Add the `http10_keepalive` option to allow disabling keep-alive for HTTP/1.0 connections.
- Add the `idle_timeout` option for HTTP/2.
- Add the `sendfile` option to both HTTP/1.1 and HTTP/2. It allows disabling the sendfile syscall entirely for all connections. It is recommended to disable sendfile when using VirtualBox shared folders.
- Add the `rate_limited/2` callback to REST handlers.
- Add the `deflate_opts` option to Websocket handlers that allows configuring deflate options for the permessage-deflate extension.
- Add the `charset` option to `cowboy_static`.
- Add support for the SameSite cookie attribute.
- Update Ranch to 1.7.0
- Update Cowlib to 2.7.0

L.2 Experimental features added

- Add support for range requests (RFC7233) in REST handlers. This adds two new callbacks: `ranges_accepted/2` and `range_satisfiable/2` along with the user-specified `ProvideRangeCallback/2`.
- Add automatic handling of range requests to REST handlers that return the callback `auto` from `ranges_accepted/2`. Cowboy will call the configured `ProvideCallback` and then split the output automatically for the ranged response.
- Enable range requests support in `cowboy_static`.
- Add the `{deflate, boolean()}` Websocket handler command to disable permessage-deflate compression temporarily.
- Add the `compress_threshold` option which allows configuring how much data must be present in a response body to compress it. This only applies to non-streamed bodies at this time.
- Add the `compress_buffering` option which allows controlling whether some buffering may be done when streaming a response body. Change the default behavior to not buffer to make sure it works by default in all scenarios.
- Add the `{set_options, map()}` command to stream handlers and Websocket handlers. This can be used to update options on a per-request basis. Allow overriding the `idle_timeout` option for both HTTP/1.1 and Websocket, the `cowboy_compres` options for HTTP/1.1 and HTTP/2 and the `chunked` option for HTTP/1.1.

L.3 Bugs fixed

- Do not send a content-length automatically with 304 responses. This status code allows a content-length that corresponds to what would have been sent for a 200 response, but is never followed by a body.
- HTTP/2 streams are now terminated once the body has been sent fully, instead of immediately once the stop command is returned (by default when the request process exits). Metrics will therefore more accurately represent when a stream ended.
- Terminate connection processes gracefully when the parent process exists or when sys:terminate/2,3 is called.
- Automatically ignore the boundary parameter of multipart media types when using REST handlers. This is a special parameter that may change with all requests and cannot be predicted.
- Fix parsing of the accept header when it contains charset parameters. They are case insensitive and will now be lowercased, like for accept-charset and content-type.
- Handle the charset parameter using `charsets_provided` when it is present in the accept header when using REST handlers.
- Don't select charsets when the q-value is 0 in REST handlers.
- Handle accept-charset headers that include a wildcard in REST handlers.
- Only send a charset header when the content-type negotiated is of type text in REST handlers.
- Remove the default charset iso-8859-1 from REST handlers when no other is provided. This has been removed from the HTTP specifications for a long time.
- Many cases where a content-type header was sent unnecessarily in the REST handlers response have been fixed.
- Handle error_response commands in `cowboy_metrics_h`.
- A number of types and function specifications were fixed or improved. Dialyzer is now run against both the code and tests to help uncover issues.
- An undefined `cowboy_router` behavior has been documented.

Appendix M

Migrating from Cowboy 2.4 to 2.5

Cowboy 2.5 focused on making the test suites pass. A variety of new features, fixes and improvements have also been worked on.

M.1 Features added

- Add option `linger_timeout` to control how long Cowboy will wait before closing the socket when shutting down the connection. This helps avoid the TCP reset problem HTTP/1.1 suffers from. The default is now 1000 ms.
- It is now possible to stream a response body without using chunked transfer-encoding when the protocol is HTTP/1.1. To enable this behavior, simply pass the content-length header with the expected size when initiating the streamed response.
- Update Ranch to 1.6.2
- Update Cowlib to 2.6.0

M.2 Experimental features added

- Websocket handlers now feature a commands-based interface. The return value from the callbacks can now take the form `{Commands, State}` where `Commands` can be frames to be sent or commands yet to be introduced. New commands will be available only through this new interface.
- Add the `{active, boolean()}` Websocket handler command. It allows disabling reading from the socket when `false` is returned. `true` reenables reading from the socket.
- Add the protocol option `logger` that allows configuring which logger module will be used. The logger module must follow the interface of the new `logger` module in Erlang/OTP 21, or be set to `error_logger` to keep the old behavior. A similar transport option exists in Ranch 1.6; both options are necessary to override Cowboy's default behavior completely.
- Add the `{log, Level, Format, Args}` stream handler command. Making it a command rather than a direct call will simplify silencing particular log messages.

M.3 New functions

- The function `cowboy_req:stream_events/3` streams one or more text/event-stream events, encoding them automatically.
- The functions `cowboy_req:read_and_match_urlencoded_body/2,3` can be used to read, parse and match application/x-www-form-urlencoded request bodies, in a similar way to `cowboy_req:match_qs/2`.

M.4 Bugs fixed

- Fix Erlang/OTP 21 warnings.
- Ensure that the port number is always defined in the Req object. When it is not provided in the request, the default port number for the protocol being used will be set.
- Ensure stream handlers can run after `cowboy_stream_h`.
- Honor the SETTINGS_ENABLE_PUSH HTTP/2 setting: don't send PUSH frames to clients that disabled it.
- Fix HTTP/2 `settings_timeout` option when the value is set to `infinity`.
- HTTP/1.1 responses will no longer include a trailer header when the request had no te header.
- HTTP/1.1 204 responses no longer send the transfer-encoding header when `cowboy_req:stream_reply/2,3` is used to send a response.
- Improve HTTP/1.1 keepalive handling to avoid processing requests that follow the final request that will receive a response.
- Improve the validation of HTTP/1.1 absolute-form requests.
- When the `switch_protocol` is used after a response was sent, Cowboy will no longer attempt to send the 101 informational response for the protocol upgrade. This caused a crash of the connection previously.
- Errors that occur when a callback returned by `content_types_provided` does not exist have been improved.
- Prevent annoying error logs when using sendfile in Erlang/OTP 20 and lower.
- Add missing frame types to `websocket_handle`.
- A test suite has been added for RFC8297 to ensure that 103 informational responses can be sent.
- Numerous test cases have been fixed, improved or removed in order to make the test suites pass. Most of the failures were caused by broken tests.
- Some misguiding or incorrect statements in the documentation have been removed or clarified.

Appendix N

Migrating from Cowboy 2.3 to 2.4

Cowboy 2.4 focused on improving the HTTP/2 implementation. All existing tests from RFC7540 and the h2spec test suite now all pass. Numerous options have been added to control SETTINGS and related behavior. In addition experimental support for Websocket over HTTP/2 was added.

N.1 Features added

- Add experimental support for Websocket over HTTP/2. You can use the `enable_connect_protocol` option to enable. It implements the following draft: https://tools.ietf.org/html/draft-ietf-httpbis-h2-websockets-01
- Add options `max_decode_table_size` and `max_encode_table_size` to restrict the size of the HPACK compression dictionary.
- Add option `max_concurrent_streams` to restrict the number of HTTP/2 streams that can be opened concurrently.
- Add options `initial_connection_window_size` and `initial_stream_window_size` to restrict the size of the HTTP/2 request body buffers for the whole connection and per stream, respectively.
- Add options `max_frame_size_received` and `max_frame_size_sent` to restrict the size of HTTP/2 frames.
- Add option `settings_timeout` to reject clients that did not send a SETTINGS ack. Note that this currently may only occur at the beginning of the connection.
- Update Ranch to 1.5.0
- Update Cowlib to 2.3.0

N.2 Bugs fixed

- Fix the END_STREAM flag for informational responses when using HTTP/2.
- Receive and ignore HTTP/2 request trailers if any for HTTP/2 requests. Request trailer information will be propagated to the user code in a future release.
- Reject WINDOW_UPDATE frames that are sent after the client sent an RST_STREAM. Note that Cowboy will not keep state information about terminated streams forever and so the behavior might differ depending on when the stream was reset.
- Reject streams that depend on themselves. Note that Cowboy currently does not implement HTTP/2's priority mechanisms so this issue was harmless.
- Reject HTTP/2 requests where the body size is different than the content-length value. Note that due to how Cowboy works some requests might go through regardless, for example when the user code does not read the request body.
- Fix all existing test failures from RFC7540. This was mostly incorrect test cases or intermittent failures.

Appendix O

Migrating from Cowboy 2.2 to 2.3

Cowboy 2.3 focused on making the Cowboy processes behave properly according to OTP principles. This version is a very good milestone toward that goal and most of everything should now work. Release upgrades and a few details will be improved in future versions.

O.1 Features added

- Add support for all functions from the module `sys`. Note that Cowboy currently does not implement the `sys` debugging mechanisms as tracing is recommended instead.
- Add a `max_frame_size` option for Websocket handlers to close the connection when the client attempts to send a frame that's too large. It currently defaults to `infinity` to avoid breaking existing code but will be changed in a future version.
- Update Cowlib to 2.2.1.
- Add support for the 308 status code and a test suite for RFC7538 where it is defined.

O.2 Bugs fixed

- Ensure timeout options accept the value `infinity` as documented.
- Properly reject HTTP/2 requests with an invalid content-length header instead of simply crashing.
- When switching from HTTP/1.1 to Websocket or user protocols all the messages in the mailbox were flushed. Only messages specific to `cowboy_http` should now be flushed.
- Parsing of the x-forwarded-for header has been corrected. It now supports IPv6 addresses both with and without port.
- Websocket subprotocol tokens are now parsed in a case insensitive manner, according to the spec.
- Cookies without values are now allowed. For example `Cookie: foo`.
- Colons are now allowed within path segments in routes provided to `cowboy_router:compile/1` as long as they are not the first character of the path segment.
- The `cowboy_req:delete_resp_header/2` function will no longer crash when no response header was set before calling it.
- A miscount of the output HTTP/2 flow control window has been fixed. It prevented sending the response body fully to some clients. The issue only affected response bodies sent as iolists.

Appendix P

Migrating from Cowboy 2.1 to 2.2

Cowboy 2.2 focused on adding features required for writing gRPC servers and on completing test suites for the core HTTP RFCs, fixing many bugs along the way.

P.1 Features added

- Add support for sending trailers at the end of response bodies. Trailers are additional header fields that may be sent after the body to add more information to the response. Their usage is required in gRPC servers. They are optional and may be discarded in other scenarios (for example if the request goes through an HTTP/1.0 proxy, as HTTP/1.0 does not support trailers).
- The `max_skip_body_length` option was added to `cowboy_http`. It controls how much of a request body Cowboy is willing to skip when the handler did not touch it. If the remaining body size is too large Cowboy instead closes the connection. It defaults to 1MB.
- The CONNECT and TRACE methods are now rejected as they are currently not implemented and must be handled differently than other methods. They will be implemented in a future release.

P.2 New functions

- The function `stream_trailers/2` has been added. It terminates a stream and adds trailer fields at the end of the response. A corresponding stream handler command `{trailers, Trailers}` has also been added.

P.3 Bugs fixed

- Test suites for the core HTTP RFCs RFC7230, RFC7231 and RFC7540 have been completed. Many of the bugs listed here were fixed as a result of this work.
- Many HTTP/2 edge cases when clients are misbehaving have been corrected. This includes many cases where the request is malformed (for example when a pseudo-header is present twice).
- When the HTTP/2 SETTINGS_INITIAL_WINDOW_SIZE value changes, Cowboy now properly updates the flow control windows.
- HTTP/2 could mistakenly log stray messages that actually were expected. This is no longer the case.
- We no longer send a GOAWAY frame when the HTTP/2 preface is invalid.
- Some values in the Req object of pushed requests were in the wrong type. They are now the expected binary instead of iolist.

- A response body was sometimes sent in response to HEAD requests when using HTTP/2. The body is now ignored.
- The `max_headers` option for `cowboy_http` was not always respected depending on the contents of the buffer. The limit is now strict.
- When an early error occurred on the HTTP/1.1 request line, the partial Req given to stream handlers was missing the `ref` and `peer` information. This has been corrected.
- Absolute URIs with a userinfo component, or without an authority component, are now properly rejected for HTTP/1.0 and HTTP/1.1.
- Whitespace is no longer allowed in header lines before the colon.
- 408 responses to HTTP/1.1 requests now properly include a connection: close header indicating that we are going to close the connection. This header will also be sent for other early errors resulting in the closing of the connection.
- When both the transfer-encoding and content-length headers are sent in an HTTP/1.1 request, the transfer-encoding now takes precedence over the content-length header and the latter is removed from the Req object.
- A 400 response is now returned when the transfer-encoding header is invalid or contains any transfer-coding other than chunked.
- Invalid chunk sizes are now rejected immediately.
- Chunk extensions are now limited to 129 characters. They are not used in practice and are still ignored by Cowboy. The limit is not configurable.
- The final chunk was mistakenly sent in responses to HEAD requests. This is now corrected.
- `OPTIONS *` requests were broken in Cowboy 2.0. They are now working again. Both the routing and `cowboy_req:uri/1,2` have been corrected.
- 204 responses no longer include a content-length header.
- A packet could be lost when switching to Websocket or any other protocol via the `switch_protocol` command. This is now fixed.
- A 426 response will now be sent when a handler requires the client to upgrade to Websocket and the request did not include the required headers.
- Both experimental stream handlers `cowboy_metrics_h` and `cowboy_tracer_h` received a number of fixes and improvements.

Appendix Q

Migrating from Cowboy 2.0 to 2.1

Cowboy 2.1 focused on adding features that were temporarily removed in Cowboy 2.0. A number of bugs found in the 2.0 release were also fixed.

Q.1 Features added

- It is now possible to obtain the client TLS certificate and the local IP/port for the connection from the Req object.
- Informational responses (1XX responses) can now be sent. They must be sent before initiating the final response.
- The `expect: 100-continue` header is now handled automatically. The 100 response will be sent on the first `cowboy_req:re` call. This only applies when using the default `cowboy_stream_h` stream handler.

Q.2 Experimental features added

Experimental features are previews of features that will be added in a future release. They are not documented and their interface may change at any time. You are welcome to try them and provide feedback.

- The `cowboy_metrics_h` stream handler can be used to extract metrics out of Cowboy. It must be used first in the list of stream handlers, and will record all events related to requests, responses and spawned processes. When the stream terminates it will pass this information to a user-defined callback.
- The `cowboy_tracer_h` stream handler can be used to setup automatic tracing of specific requests. You can conditionally enable tracing based on a function, header, path or any other element from the request and the trace will apply to the entire connection and any processes created by it. This is meant to be used for debugging both in tests and production.

Q.3 Changed behaviors

- The `cowboy_rest` handler now implements a mechanism for switching to a different type of handler from any callback where `stop` is also allowed. Switch by returning `{switch_handler, Module}` or `{switch_handler, Module, Opts}`. This is especially useful for switching to `cowboy_loop` for streaming the request or response body.
- REST callbacks that do not allow `stop` as a return value are now explicitly listed in the documentation.

Q.4 New functions

- The function `cowboy_req:sock/1` returns the IP/port of the local socket.
- The function `cowboy_req:cert/1` returns the client TLS certificate or `undefined` if it isn't available.
- The function `cowboy_req:inform/2,3` sends an informational response.

Q.5 Bugs fixed

- Ensure HTTP/2 connections are not closed prematurely when the user code does not read the request body.
- Ensure HTTP/1.1 streams are not terminated too early. Their behavior is now consistent with the HTTP/2 code where the stream handler is only terminated when the `stop` command is returned.
- Sending zero-sized data from stream handlers or from `cowboy_req:stream_body/3` could lead to issues with HTTP/1.1. This has been fixed.
- The final chunk sent by Cowboy when it terminates a chunked body after the handler process exits was not passed through stream handlers, which could lead to issues when `cowboy_compress_h` was being used. This is now corrected.
- The stream handler state was discarded in some cases where Cowboy had to send a response or response data automatically when ending a stream. This has now been corrected.
- The stream handler callback `terminate/3` will now be called when switching to another protocol using the command `switch_protocol`. This doesn't apply when doing upgrades to HTTP/2 as those occur before the stream is initialized.
- Cowlib has been updated to 2.0.1 to fix an issue with Websocket compression when using Erlang/OTP 20.1. Note that at the time of writing all 20.1 versions (from 20.1 to 20.1.4) have issues when compression is enabled. It is expected to work properly from 20.1.5 onward. In the meantime it is recommended to run the plain 20.1 release and disable Websocket compression, or use a release before 20.1.
- Cowboy will no longer crash when the `cowboy_clock` process is not running. This can happen when Cowboy is being restarted during upgrades, for example.

Appendix R

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.

R.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.

R.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.

R.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.

R.4 Changed behaviors

- The handler behaviors have been renamed and are now `cowboy_handler`, `cowboy_loop`, `cowboy_rest` and `cowboy_webso`
- 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.

R.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).

R.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.

R.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 can now take a sendfile tuple.
- Remove many undocumented functions from cowboy_req, including the functions cowboy_req:get/2 and cowboy_req:set

R.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.
- Clients that send multiple headers of the same name will have the values of those headers concatenated into a comma-separated list. This is of special importance in the case of the content-type header, as previously only the first value was used in the content_types_accepted/2 step in REST handlers.
- 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 S

HTTP and other specifications

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

S.1 HTTP

S.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

S.1.2 Current

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

- Preload: Preload
- PROXY: The PROXY protocol
- 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 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
- RFC 8288: Web Linking
- RFC 8297: An HTTP Status Code for Indicating Hints
- RFC 8336: The ORIGIN HTTP/2 Frame
- RFC 8441: Bootstrapping WebSockets with HTTP/2
- RFC 8470: Using Early Data in HTTP
- RFC 8473: Token Binding over HTTP
- RFC 8586: Loop Detection in Content Delivery Networks (CDNs)
- RFC 8594: The Sunset HTTP Header Field
- RFC 8673: HTTP Random Access and Live Content
- RFC 8674: The "safe" HTTP Preference
- RFC 8740: Using TLS 1.3 with HTTP/2
- RFC 8941: Structured Field Values for HTTP
- RFC 8942: HTTP Client Hints
- Trace Context: Trace Context
- Webmention: Webmention

S.1.3 Upcoming

- Clear Site Data
- 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

S.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

S.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 5829: Link Relation Types for Simple Version Navigation between Web Resources
- RFC 6657: Update to MIME regarding "charset" Parameter Handling in Textual Media Types
- RFC 6690: Constrained RESTful Environments (CoRE) Link Format
- RFC 7807: Problem Details for HTTP APIs

- RFC 6906: The *profile* Link Relation Type
- RFC 8631: Link Relation Types for Web Services
- 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

S.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)
- RFC 8565: Hypertext Jeopardy Protocol (HTJP/1.0)
- SPDY: SPDY Protocol
- x-webkit-deflate-frame: Deprecated Websocket compression

S.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
- RFC 8615: Well-Known URIs
- URL
- URL Living Standard

S.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
- RFC 8607: Calendaring Extensions to WebDAV (CalDAV): Managed Attachments

S.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

- RFC 8323: CoAP over TCP, TLS, and WebSockets
- RFC 8516: "Too Many Requests" Response Code for CoAP
- RFC 8613: Object Security for Constrained RESTful Environments
- RFC 8710: Multipart Content-Format for CoAP
- RFC 8768: CoAP Hop-Limit Option