Cowboy User Guide

Contents

I Rationale 1

1 The modern Web 2

1.1 The prehistoric Web 2

1.2 HTTP/1.1 2

1.3 REST 2

1.4 XmlHttpRequest 3

1.5 Long-polling 3

1.6 HTML5 3

1.7 EventSource 4

1.8 Websocket 4

1.9 SPDY 4

1.10 HTTP/2.0 4

2 Erlang and the Web 5

2.1 The Web is concurrent 5

2.2 The Web is soft real time 5

2.3 The Web is asynchronous 6

2.4 The Web is omnipresent 6

2.5 Erlang is the ideal platform for the Web 7

II Introduction 8

3 Introduction 9

3.1 Prerequisites 9

3.2 Supported platforms 9

3.3 Versioning 9

3.4 Conventions 10

4 Getting started 11
4.1 Bootstrap 11
4.2 Cowboy setup 12
4.3 Listening for connections 12
4.4 Handling requests 12

5 Request overview 13
5.1 Request/response 13
5.2 And then? 14
5.3 Keep-alive (HTTP/1.1) 14
5.4 Pipelining (HTTP/1.1) 14
5.5 Asynchronous requests (SPDY) 15

6 Erlang for beginners 16
6.1 Learn You Some Erlang for Great Good! 16
6.2 Programming Erlang 16

III Configuration 17

7 Routing 18
7.1 Structure 18
7.2 Match syntax 18
7.3 Constraints 20
7.4 Compilation 20
7.5 Live update 20

8 Constraints 21
8.1 Structure 21
8.2 Built-in constraints 21
8.3 Custom constraint 21

9 Static files 22
9.1 Serve one file 22
9.2 Serve all files from a directory 22
9.3 Customize the mimetype detection 23
9.4 Generate an etag 23

IV Request and response 24

10 Handlers 25
10.1 Plain HTTP handlers 25
10.2 Other handlers 25
10.3 Cleaning up 26

11 Loop handlers 27
11.1 Initialization . . . 27
11.2 Receive loop . . . 27
11.3 Streaming loop . . . 28
11.3.1 Cleaning up . . . 28
11.4 Timeout . . . 28
11.5 Hibernate . . . 29

12 The Req object 30
12.1 A special variable . . . 30
12.2 Overview of the cowboy_req interface . . . 30
12.3 Request . . . 30
12.4 Bindings . . . 31
12.5 Query string . . . 31
12.6 Request URL . . . 32
12.7 Headers . . . 32
12.8 Meta . . . 33
12.9 Peer . . . 33

13 Reading the request body 34
13.1 Check for request body . . . 34
13.2 Request body length . . . 34
13.3 Reading the body . . . 34
13.4 Streaming the body . . . 35
13.5 Rate of data transmission . . . 35
13.6 Transfer and content decoding . . . 35
13.7 Reading a form urlencoded body . . . 36

14 Sending a response 37
14.1 Reply . . . 37
14.2 Chunked reply . . . 38
14.3 Preset response headers . . . 38
14.4 Preset response body . . . 38
14.5 Sending files . . . 39

15 Using cookies 40
15.1 Setting cookies . . . 40
15.2 Reading cookies . . . 41

16 Multipart requests **42**
16.1 Structure . . . 42
16.2 Form-data . . . 42
16.3 Checking the content-type . . . 42
16.4 Reading a multipart message . . . 43
16.5 Skipping unwanted parts . . . 43

V REST **45**

17 REST principles **46**
17.1 REST architecture . . . 46
17.2 Resources and resource identifiers . . . 46
17.3 Resource representations . . . 47
17.4 Self-descriptive messages . . . 47
17.5 Hypermedia as the engine of application state . . . 47

18 REST handlers **48**
18.1 Initialization . . . 48
18.2 Methods . . . 48
18.3 Callbacks . . . 48
18.4 Meta data . . . 49
18.5 Response headers . . . 50

19 REST flowcharts **51**
19.1 Start . . . 51
19.2 OPTIONS method . . . 53
19.3 Content negotiation . . . 53
19.4 GET and HEAD methods . . . 55
19.5 PUT, POST and PATCH methods . . . 57
19.6 DELETE method . . . 59
19.7 Conditional requests . . . 61

20 Designing a resource handler **64**
20.1 The service . . . 64
20.2 Type of resource handler . . . 64
20.3 Collection handler . . . 64
20.4 Single resource handler . . . 65
20.5 The resource . . . 65
20.6 Representations . . . 65
20.7 Redirections . . . 66

20.8 The request . . . 66
20.9 OPTIONS method . . . 66
20.10 GET and HEAD methods . . . 66
20.11 PUT, POST and PATCH methods . . . 66
20.12 DELETE methods . . . 66

VI Websocket 67

21 The Websocket protocol 68
21.1 Description . . . 68
21.2 Implementation . . . 68

22 Handling Websocket connections 69
22.1 Initialization . . . 69
22.2 Handling frames from the client . . . 70
22.3 Handling Erlang messages . . . 70
22.4 Sending frames to the socket . . . 70
22.5 Ping and timeout . . . 71
22.6 Hibernate . . . 71
22.7 Supporting older browsers . . . 71

VII Internals 72

23 Architecture 73
23.1 One process per connection . . . 73
23.2 Binaries . . . 73
23.3 Date header . . . 73
23.4 Max connections . . . 73

24 Dealing with broken clients 74
24.1 Lowercase headers . . . 74
24.2 Camel-case headers . . . 74
24.3 Chunked transfer-encoding . . . 74

25 Middlewares 75
25.1 Usage . . . 75
25.2 Configuration . . . 75
25.3 Routing middleware . . . 76
25.4 Handler middleware . . . 76

26 Sub protocols 77

26.1 Usage 77

26.2 Upgrade 77

27 Hooks 78

27.1 Onresponse 78

Part I

Rationale

Chapter 1

The modern Web

Let's take a look at various technologies from the beginnings of the Web up to this day, and get a preview of what's coming next.

Cowboy is compatible with all the technology cited in this chapter except of course HTTP/2.0 which has no implementation in the wild at the time of writing.

1.1 The prehistoric Web

HTTP was initially created to serve HTML pages and only had the GET method for retrieving them. This initial version is documented and is sometimes called HTTP/0.9. HTTP/1.0 defined the GET, HEAD and POST methods, and was able to send data with POST requests.

HTTP/1.0 works in a very simple way. A TCP connection is first established to the server. Then a request is sent. Then the server sends a response back and closes the connection.

Suffice to say, HTTP/1.0 is not very efficient. Opening a TCP connection takes some time, and pages containing many assets load much slower than they could because of this.

Most improvements done in recent years focused on reducing this load time and reducing the latency of the requests.

1.2 HTTP/1.1

HTTP/1.1 quickly followed and added a keep-alive mechanism to allow using the same connection for many requests, as well as streaming capabilities, allowing an endpoint to send a body in well defined chunks.

HTTP/1.1 defines the OPTIONS, GET, HEAD, POST, PUT, DELETE, TRACE and CONNECT methods. The PATCH method was added in more recent years. It also improves the caching capabilities with the introduction of many headers.

HTTP/1.1 still works like HTTP/1.0 does, except the connection can be kept alive for subsequent requests. This however allows clients to perform what is called as pipelining: sending many requests in a row, and then processing the responses which will be received in the same order as the requests.

1.3 REST

The design of HTTP/1.1 was influenced by the REST architectural style. REST, or REpresentational State Transfer, is a style of architecture for loosely connected distributed systems.

REST defines constraints that systems must obey to in order to be RESTful. A system which doesn't follow all the constraints cannot be considered RESTful.

REST is a client-server architecture with a clean separation of concerns between the client and the server. They communicate by referencing resources. Resources can be identified, but also manipulated. A resource representation has a media type and information about whether it can be cached and how. Hypermedia determines how resources are related and how they can be used. REST is also stateless. All requests contain the complete information necessary to perform the action.

HTTP/1.1 defines all the methods, headers and semantics required to implement RESTful systems.

REST is most often used when designing web application APIs which are generally meant to be used by executable code directly.

1.4 XmlHttpRequest

Also know as AJAX, this technology allows Javascript code running on a web page to perform asynchronous requests to the server. This is what started the move from static websites to dynamic web applications.

XmlHttpRequest still performs HTTP requests under the hood, and then waits for a response, but the Javascript code can continue to run until the response arrives. It will then receive the response through a callback previously defined.

This is of course still requests initiated by the client, the server still had no way of pushing data to the client on its own, so new technology appeared to allow that.

1.5 Long-polling

Polling was a technique used to overcome the fact that the server cannot push data directly to the client. Therefore the client had to repeatedly create a connection, make a request, get a response, then try again a few seconds later. This is overly expensive and adds an additional delay before the client receives the data.

Polling was necessary to implement message queues and other similar mechanisms, where a user must be informed of something when it happens, rather than when he refreshes the page next. A typical example would be a chat application.

Long-polling was created to reduce the server load by creating less connections, but also to improve latency by getting the response back to the client as soon as it becomes available on the server.

Long-polling works in a similar manner to polling, except the request will not get a response immediately. Instead the server leaves it open until it has a response to send. After getting the response, the client creates a new request and gets back to waiting.

You probably guessed by now that long-polling is a hack, and like most hacks it can suffer from unforeseen issues, in this case it doesn't always play well with proxies.

1.6 HTML5

HTML5 is, of course, the HTML version after HTML4. But HTML5 emerged to solve a specific problem: dynamic web applications.

HTML was initially created to write web pages which compose a website. But soon people and companies wanted to use HTML to write more and more complex websites, eventually known as web applications. They are for example your news reader, your email client in the browser, or your video streaming website.

Because HTML wasn't enough, they started using proprietary solutions, often implemented using plug-ins. This wasn't perfect of course, but worked well enough for most people.

However, the needs for a standard solution eventually became apparent. The browser needed to be able to play media natively. It needed to be able to draw anything. It needed an efficient way of streaming events to the server, but also receiving events from the server.

The solution went on to become HTML5. At the time of writing it is being standardized.

1.7 EventSource

EventSource, sometimes also called Server-Sent Events, is a technology allowing servers to push data to HTML5 applications.

EventSource is one-way communication channel from the server to the client. The client has no means to talk to the server other than by using HTTP requests.

It consists of a Javascript object allowing setting up an EventSource connection to the server, and a very small protocol for sending events to the client on top of the HTTP/1.1 connection.

EventSource is a lightweight solution that only works for UTF-8 encoded text data. Binary data and text data encoded differently are not allowed by the protocol. A heavier but more generic approach can be found in Websocket.

1.8 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.

1.9 SPDY

SPDY is an attempt to reduce page loading time by opening a single connection per server, keeping it open for subsequent requests, and also by compressing the HTTP headers to reduce the size of requests.

SPDY is compatible with HTTP/1.1 semantics, and is actually just a different way of performing HTTP requests and responses, by using binary frames instead of a text-based protocol. SPDY also allows the server to send extra responses following a request. This is meant to allow sending the resources associated with the request before the client requests them, saving latency when loading websites.

SPDY is an experiment that has proven successful and is used as the basis for the HTTP/2.0 standard.

Browsers make use of TLS Next Protocol Negotiation to upgrade to a SPDY connection seamlessly if the protocol supports it.

The protocol itself has a few shortcomings which are being fixed in HTTP/2.0.

1.10 HTTP/2.0

HTTP/2.0 is the long-awaited update to the HTTP/1.1 protocol. It is based on SPDY although a lot has been improved at the time of writing.

HTTP/2.0 is an asynchronous two-ways communication channel between two endpoints.

Chapter 2

Erlang and the Web

2.1 The Web is concurrent

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

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

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

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

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

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

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

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

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

2.2 The Web is soft real time

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

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

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

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

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

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

2.3 The Web is asynchronous

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

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

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

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

The more recent Web technologies, like Websocket of course, but also SPDY and 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 seem 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 happen, the Erlang process crashes and is then restarted by a special process called a supervisor.

The Erlang developer thus has no need to fear about 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 him to write a lot less code, and letting him 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 Erlang is the ideal platform for the Web

Erlang provides all the important features that the Web requires or will require in the near future. Erlang is a perfect match for the Web, and it only makes sense to use it to build web applications.

Part II

Introduction

Chapter 3

Introduction

Cowboy is a small, fast and modular HTTP server written in Erlang.

Cowboy aims to provide a complete HTTP stack, including its derivatives SPDY, Websocket and REST. Cowboy currently supports HTTP/1.0, HTTP/1.1, Websocket (all implemented drafts + standard) and Webmachine-based REST.

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 both a Function Reference and a User Guide.

3.1 Prerequisites

Beginner Erlang knowledge is recommended for reading this guide.

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

3.2 Supported platforms

Cowboy is tested and supported on Linux.

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 17.0, 17.1.2 and 17.3. By the time this branch gets released the target version will probably be 18.0 and above.

Cowboy may be compiled on other Erlang versions with small source code modifications but there is no guarantee that it will work as expected.

Cowboy uses the maps data type which was introduced in Erlang 17.0.

3.3 Versioning

Cowboy uses Semantic Versioning 2.0.0.

3.4 Conventions

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

Header names are case insensitive. Cowboy converts all the request header names to lowercase, and expects your application to provide lowercase header names in the response.

The same applies to any other case insensitive value.

Chapter 4

Getting started

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

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

4.1 Bootstrap

We are going to use the Erlang.mk build system. It also offers bootstrap features allowing us to quickly get started without having to deal with minute details.

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://raw.githubusercontent.com/ninenines/erlang.mk/master/erlang.mk
```

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

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

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

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

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

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

4.2 Cowboy setup

Modifying the *Makefile* allows the build system to know it needs to fetch and compile Cowboy. To do that we simply need to add two lines to our Makefile to make it look like this:

```
PROJECT = hello_erlang

DEPS = cowboy
dep_cowboy_commit = master

include erlang.mk
```

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

4.3 Listening for connections

We will do this when our application starts. It's a two step process. First we need to define and compile the dispatch list, a list of routes that Cowboy will use to map requests to handler modules. Then we tell Cowboy to listen for connections.

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_http(my_http_listener, 100, [{port, 8080}],
        [{env, [{dispatch, Dispatch}]}]
    ),
    hello_erlang_sup:start_link().
```

The dispatch list is explained in great details in the Routing Chapter 7 chapter. For this tutorial we map the path `/` to the handler module `hello_handler`. This module doesn't exist yet, we still have to write it.

If you build and start the release, then open http://localhost:8080 in your browser, you will get an error because the module is missing. Any other URL, like http://localhost:8080/test, will result in a 404 error.

4.4 Handling requests

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

First, let's generate a handler from a template.

```
$ make new t=cowboy_http n=hello_handler
```

You can then open the *src/hello_handler.erl* file and modify the `init/2` function like this to send a reply.

```
init(Req, Opts) ->
    Req2 = cowboy_req:reply(200,
        [{<<"content-type">>, <<"text/plain">>}],
        <<"Hello Erlang!">>,
        Req),
    {ok, Req2, Opts}.
```

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

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

Chapter 5

Request overview

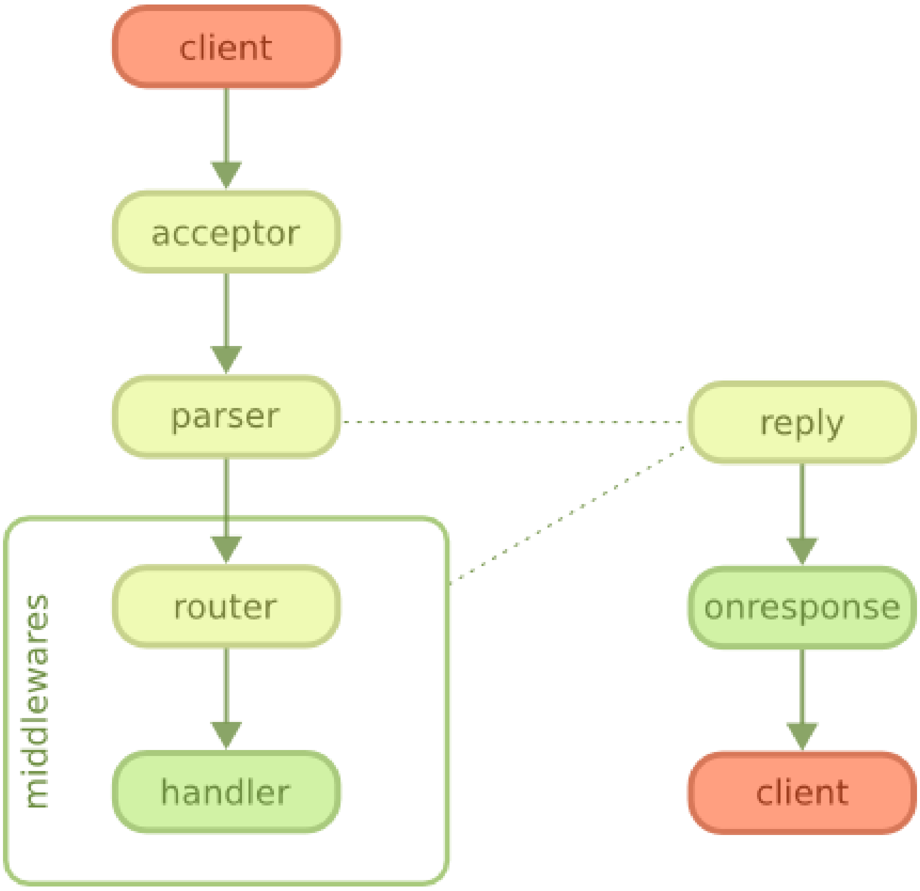
This chapter explains the different steps a request goes through until a response is sent, along with details of the Cowboy implementation.

5.1 Request/response

As you already know, HTTP clients connect to the server and send a request for a resource; the server then sends a response containing the resource if it could obtain it.

Before the server can send the resource, however, it needs to perform many different operations to read the request, find the resource, prepare the response being sent and often other related operations the user can add like writing logs.

Requests take the following route in Cowboy:


client
acceptor
parser
reply
middlewares
router
onresponse
handler
client

This shows the default middlewares, but they may be configured differently in your setup. The dark green indicates the points where you can hook your own code, the light green is the Cowboy code that you can of course configure as needed.

The `acceptor` is the part of the server that accepts the connection and create an Erlang process to handle it. The `parser` then starts reading from the socket and handling requests as they come until the socket is closed.

A response may be sent at many different points in the life of the request. If Cowboy can't parse the request, it gives up with an error response. If the router can't find the resource, it sends a not found error. Your own code can of course send a response at any time.

When a response is sent, you can optionally modify it or act upon it by enabling the `onresponse` hook. By default the response is sent directly to the client.

5.2 And then?

Behavior depends on what protocol is in use.

HTTP/1.0 can only process one request per connection, so Cowboy will close the connection immediately after it sends the response.

HTTP/1.1 allows the client to request that the server keeps the connection alive. This mechanism is described in the next section.

SPDY is designed to allow sending multiple requests asynchronously on the same connection. Details on what this means for your application is described in this chapter.

5.3 Keep-alive (HTTP/1.1)

With HTTP/1.1, the connection may be left open for subsequent requests to come. This mechanism is called `keep-alive`.

When the client sends a request to the server, it includes a header indicating whether it would like to leave the socket open. The server may or may not accept, indicating its choice by sending the same header in the response.

Cowboy will include this header automatically in all responses to HTTP/1.1 requests. You can however force the closing of the socket if you want. When Cowboy sees you want to send a `connection:close` header, it will not override it and will close the connection as soon as the reply is sent.

This snippet will force Cowboy to close the connection.

```
Req2 = cowboy_req:reply(200, [
    {<<"connection">>, <<"close">>},
], <<"Closing the socket in 3.. 2.. 1..">>, Req).
```

Cowboy will only accept a certain number of new requests on the same connection. By default it will run up to 100 requests. This number can be changed by setting the `max_keepalive` configuration value when starting an HTTP listener.

```
cowboy:start_http(my_http_listener, 100, [{port, 8080}], [
        {env, [{dispatch, Dispatch}]},
        {max_keepalive, 5}
]).
```

Cowboy implements the keep-alive mechanism by reusing the same process for all requests. This allows Cowboy to save memory. This works well because most code will not have any side effect impacting subsequent requests. But it also means you need to clean up if you do have code with side effects. The `terminate/3` function can be used for this purpose.

5.4 Pipelining (HTTP/1.1)

While HTTP is designed as a sequential protocol, with the client sending a request and then waiting for the response from the server, nothing prevents the client from sending more requests to the server without waiting for the response, due to how sockets work. The server still handles the requests sequentially and sends the responses in the same order.

This mechanism is called pipelining. It allows reducing latency when a client needs to request many resources at the same time. This is used by browsers when requesting static files for example.

This is handled automatically by the server.

5.5 Asynchronous requests (SPDY)

In SPDY, the client can send a request at any time. And the server can send a response at any time too.

This means for example that the client does not need to wait for a request to be fully sent to send another, it is possible to interleave a request with the request body of another request. The same is true with responses. Responses may also be sent in a different order.

Because requests and responses are fully asynchronous, Cowboy creates a new process for each request, and these processes are managed by another process that handles the connection itself.

SPDY servers may also decide to send resources to the client before the client requests them. This is especially useful for sending static files associated with the HTML page requested, as this reduces the latency of the overall response. Cowboy does not support this particular mechanism at this point, however.

Chapter 6

Erlang for beginners

Chances are you are interested in using Cowboy, but have no idea how to write an Erlang program. Fear not! This chapter will help you get started.

We recommend two books for beginners. You should read them both at some point, as they cover Erlang from two entirely different perspectives.

6.1 Learn You Some Erlang for Great Good!

The quickest way to get started with Erlang is by reading a book with the funny name of LYSE, as we affectionately call it.

It will get right into the syntax and quickly answer the questions a beginner would ask themselves, all the while showing funny pictures and making insightful jokes.

You can read an early version of the book online for free, but you really should buy the much more refined paper and ebook versions.

6.2 Programming Erlang

After writing some code, you will probably want to understand the very concepts that make Erlang what it is today. These are best explained by Joe Armstrong, the godfather of Erlang, in his book Programming Erlang.

Instead of going into every single details of the language, Joe focuses on the central concepts behind Erlang, and shows you how they can be used to write a variety of different applications.

Part III

Configuration

Chapter 7

Routing

Cowboy does nothing by default.

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

When Cowboy receives a request, it tries to match the requested host and path to the resources given in the dispatch rules. If it matches, then the associated Erlang code will be executed.

Routing rules are given per host. Cowboy will first match on the host, and then try to find a matching path.

Routes need to be compiled before they can be used by Cowboy.

7.1 Structure

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 options that will be given to it on initialization.

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

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

7.2 Match syntax

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

You can also have imbricated optional segments.

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

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

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

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

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

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

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

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

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

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

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

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

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

7.3 Constraints

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

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

Read more about constraints Chapter 8.

7.4 Compilation

The structure defined in this chapter needs to be compiled before it is passed to Cowboy. This allows Cowboy to efficiently lookup the correct handler to run instead of having to parse the routes repeatedly.

This can be done with a simple call to `cowboy_router:compile/1`.

```
Dispatch = cowboy_router:compile([
    %% {HostMatch, list({PathMatch, Handler, Opts})}
    {'_', [{'_', my_handler, []}]}
]),
%% Name, NbAcceptors, TransOpts, ProtoOpts
cowboy:start_http(my_http_listener, 100,
    [{port, 8080}],
    [{env, [{dispatch, Dispatch}]}]
).
```

Note that this function will return `{error, badarg}` if the structure given is incorrect.

7.5 Live update

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

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

Note that you need to compile the routes before updating.

Chapter 8

Constraints

Cowboy provides an optional constraints based validation feature when interacting with user input.

Constraints are first used during routing. The router uses constraints to more accurately match bound values, allowing to create routes where a segment is an integer for example, and rejecting the others.

Constraints are also used when performing a match operation on input data, like the query string or cookies. There, a default value can also be provided for optional values.

Finally, constraints can be used to not only validate input, but also convert said input into proper Erlang terms, all in one step.

8.1 Structure

Constraints are provided as a list of fields and for each field a list of constraints for that field can be provided.

Fields are either the name of the field; the name and one or more constraints; or the name, one or more constraints and a default value.

When no default value is provided then the field is required. Otherwise the default value is used.

All constraints for a field will be used to match its value in the order they are given. If the value is modified by a constraint, the next constraint receives the updated value.

8.2 Built-in constraints

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

8.3 Custom constraint

In addition to the predefined constraints, Cowboy will accept a fun. This fun must accept one argument and return one of `true`, `{true, NewValue}` or `false`. The result indicates whether the value matches the constraint, and if it does it can optionally be modified. This allows converting the value to a more appropriate Erlang term.

Note that constraint functions SHOULD be pure and MUST NOT crash.

Chapter 9

Static files

Cowboy comes with a special handler built as a REST handler and designed specifically for serving static files. It is provided as a convenience and provides a quick solution 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. They are covered in the next chapter. If you decide against using a CDN solution, then please look at the chapter after that, as it explains how to efficiently serve static files on your own.

The static handler can serve either one file or all files from a given directory. It can also send etag headers for client-side caching.

To use the static file handler, simply add routes for it with the appropriate options.

9.1 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"}}
```

9.2 Serve all files from a directory

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

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

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

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

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

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

9.3 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 in the list, 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">>, []}}]}}
```

9.4 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 IV

Request and response

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 Chapter 12 and the options defined during the router configuration Chapter 7.

A handler that does nothing would look like this:

```
init(Req, _Opts) ->
    {ok, Req, #state{}}.
```

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

We need to use the Req object for sending a reply.

```
init(Req, _Opts) ->
    Req2 = cowboy_req:reply(200, [
        {<<"content-type">>, <<"text/plain">>}
    ], <<"Hello World!">>, Req),
    {ok, Req2, #state{}}.
```

As you can see we return a 3-tuple. `ok` means that the handler ran successfully. The Req object is returned as it may have been modified as is the case here: replying returns a modified Req object that you need to return back to Cowboy for proper operations.

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 `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 Chapter 18, cowboy_websocket Chapter 22 and cowboy_loop Chapter 11. 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, _Opts) ->
    {cowboy_websocket, Req, #state{}}.
```

You can also switch to your own custom handler type:

```
init(Req, _Opts) ->
    {my_handler_type, Req, #state{}}.
```

How to implement a custom handler type is described in the Sub protocols Chapter 26 chapter.

10.3 Cleaning up

All handlers coming with Cowboy allow the use of 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.

If you used the process dictionary, timers, monitors or may be receiving messages, then you can use this function to clean them up, as Cowboy might reuse the process for the next keep-alive request.

Note that while this function may be called in a Websocket handler, it is generally not useful to do any clean up as the process terminates immediately after calling this callback when using Websocket.

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 known as server-sent events.

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

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

11.1 Initialization

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

This snippet enables the loop handler.

```
init(Req, _Opts) ->
    {cowboy_loop, Req, #state{}}.
```

However it is largely recommended that you set a timeout value. The next example sets a timeout value of 30s and also makes the process hibernate.

```
init(Req, _Opts) ->
    {cowboy_loop, Req, #state{}, 30000, hibernate}.
```

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) ->
    Req2 = cowboy_req:reply(200, [], Body, Req),
    {stop, Req2, 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 a `chunk` message is received, and the loop is stopped by sending an `eof` message.

```
init(Req, _Opts) ->
    Req2 = cowboy_req:chunked_reply(200, [], Req),
    {cowboy_loop, Req2, #state{}}.

info(eof, Req, State) ->
    {stop, Req, State};
info({chunk, Chunk}, Req, State) ->
    cowboy_req:chunk(Chunk, Req),
    {ok, Req, State};
info(_Msg, Req, State) ->
    {ok, Req, State}.
```

11.3.1 Cleaning up

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

Please refer to the Handlers chapter Chapter 10 for general instructions about cleaning up.

11.4 Timeout

By default Cowboy will not attempt to close the connection if there is no activity from the client. This is not always desirable, which is why you can set a timeout. Cowboy will close the connection if no data was received from the client after the configured time. The timeout only needs to be set once and can't be modified afterwards.

Because the request may have had a body, or may be followed by another request, Cowboy is forced to buffer all data it receives. This data may grow to become too large though, so there is a configurable limit for it. The default buffer size is of 5000 bytes, but it may be changed by setting the `loop_max_buffer` middleware environment value.

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.

Chapter 12

The Req object

The Req object is this variable that you will use to obtain information about a request, read the body of the request and send a response.

12.1 A special variable

While we call it an "object", it is not an object in the OOP sense of the term. In fact it is completely opaque to you and the only way you can perform operations using it is by calling the functions from the `cowboy_req` module.

Almost all the calls to the `cowboy_req` module will return an updated request object. Just like you would keep the updated `State` variable in a gen_server, you MUST keep the updated `Req` variable in a Cowboy handler. Cowboy will use this object to know whether a response has been sent when the handler has finished executing.

The Req object allows accessing both immutable and mutable state. This means that calling some of the functions twice will not produce the same result. For example, when streaming the request body, the function will return the body by chunks, one at a time, until there is none left.

12.2 Overview of the cowboy_req interface

With the exception of functions manipulating the request body, all functions return a single value. Depending on the function this can be the requested value (method, host, path, . . .), a boolean (has_body, has_resp_header. . .) a new Req object (set_resp_body, set_resp_header. . .), or simply the atom `ok` (chunk, continue, . . .).

The request body reading functions may return `{Result, Req}` or `{Result, Value, Req}`. The functions in this category are `body/{1,2}`, `body_qs/{1,2}`, `part/{1,2}`, `part_body/{1,2}`.

This chapter covers the access functions mainly. Cookies, request body and response functions are covered in their own chapters.

12.3 Request

When a client performs a request, it first sends a few required values. They are sent differently depending on the protocol being used, but the intent is the same. They indicate to the server the type of action it wants to do and how to locate the resource to perform it on.

The method identifies the action. Standard methods include GET, HEAD, OPTIONS, PATCH, POST, PUT, DELETE. Method names are case sensitive.

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

The host, port and path parts of the URL identify the resource being accessed. The host and port information may not be available if the client uses HTTP/1.0.

```
Host = cowboy_req:host(Req),
Port = cowboy_req:port(Req),
Path = cowboy_req:path(Req).
```

The version used by the client can of course also be obtained.

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

Do note however that clients claiming to implement one version of the protocol does not mean they implement it fully, or even properly.

12.4 Bindings

After routing the request, bindings are available. Bindings are these parts of the host or path that you chose to extract when defining the routes of your application.

You can fetch a single binding. The value will be `undefined` if the binding doesn't exist.

```
Binding = cowboy_req:binding(my_binding, Req).
```

If you need a different value when the binding doesn't exist, you can change the default.

```
Binding = cowboy_req:binding(my_binding, Req, 42).
```

You can also obtain all bindings in one call. They will be returned as a list of key/value tuples.

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

If you used `...` at the beginning of the route's pattern for the host, you can retrieve the matched part of the host. The value will be `undefined` otherwise.

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

Similarly, if you used `...` at the end of the route's pattern for the path, you can retrieve the matched part, or get `undefined` otherwise.

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

12.5 Query string

The raw query string can be obtained directly.

```
Qs = cowboy_req:qs(Req).
```

You can parse the query string and then use standard library functions to access individual values.

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

You can match the query string into a map.

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

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

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

Note that in the case of duplicate query string keys, the map value will become a list of the different values.

Read more about constraints.

A default value can 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.

12.6 Request URL

You can reconstruct the full URL of the resource.

```
URL = cowboy_req:url(Req).
```

You can also obtain only the base of the URL, excluding the path and query string.

```
BaseURL = cowboy_req:host_url(Req).
```

12.7 Headers

Cowboy allows you to obtain the header values as string, or parsed into a more meaningful representation.

This will get the string value of a header.

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

You can of course set a default in case the header is missing.

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

And also obtain all headers.

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

To parse the previous header, simply call `parse_header/{2,3}` where you would call `header/{2,3}` otherwise.

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

Cowboy will crash if it doesn't know how to parse the given header, or if the value is invalid.

You can of course define a default value. Note that the default value you specify here is the parsed value you'd like to get by default.

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

The list of known headers and default values is defined in the manual.

12.8 Meta

Cowboy will sometimes associate some meta information with the request. Built-in meta values are listed in the manual for their respective modules.

This will get a meta value. The returned value will be `undefined` if it isn’t defined.

```
MetaVal = cowboy_req:meta(websocket_version, Req).
```

You can change the default value if needed.

```
MetaVal = cowboy_req:meta(websocket_version, Req, 13).
```

You can also define your own meta values. The name must be an `atom()`.

```
Req2 = cowboy_req:set_meta(the_answer, 42, Req).
```

12.9 Peer

You can obtain the peer address and port number. This is not necessarily the actual IP and port of the client, but rather the one of the machine that connected to the server.

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

Chapter 13

Reading the request body

The Req object also allows you to read the request body.

Because the request body can be of any size, all body reading operations will only work once, as Cowboy will not cache the result of these operations.

Cowboy will not attempt to read the body until you do. If handler execution ends without reading it, Cowboy will simply skip it.

Cowboy provides different ways to read the request body. You can read it directly, stream it, but also read and parse in a single call for form urlencoded formats or multipart. All of these except multipart are covered in this chapter. Multipart is covered later on in the guide.

13.1 Check for request body

You can check whether a body was sent with the request.

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

It will return `true` if there is a request body, and `false` otherwise.

Note that it is generally safe to assume that a body is sent for `POST`, `PUT` and `PATCH` requests, without having to explicitly check for it.

13.2 Request body length

You can obtain the body length if it was sent with the request.

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

The value returned will be `undefined` if the length couldn't be figured out from the request headers. If there's a body but no length is given, this means that the chunked transfer-encoding was used. You can read chunked bodies by using the stream functions.

13.3 Reading the body

You can read the whole body directly in one call.

```
{ok, Body, Req2} = cowboy_req:body(Req).
```

By default, Cowboy will attempt to read up to a size of 8MB. You can override this limit as needed.

```
{ok, Body, Req2} = cowboy_req:body(Req, [{length, 100000000}]).
```

You can also disable it.

```
{ok, Body, Req2} = cowboy_req:body(Req, [{length, infinity}]).
```

It is recommended that you do not disable it for public facing websites.

If the body is larger than the limit, then Cowboy will return a `more` tuple instead, allowing you to stream it if you would like to.

13.4 Streaming the body

You can stream the request body by chunks.

Cowboy returns a `more` tuple when there is more body to be read, and an `ok` tuple for the last chunk. This allows you to loop over all chunks.

```
body_to_console(Req) ->
    case cowboy_req:body(Req) of
        {ok, Data, Req2} ->
            io:format("~s", [Data]),
            Req2;
        {more, Data, Req2} ->
            io:format("~s", [Data]),
            body_to_console(Req2)
    end.
```

You can of course set the `length` option to configure the size of chunks.

13.5 Rate of data transmission

You can control the rate of data transmission by setting options when calling body functions. This applies not only to the functions described in this chapter, but also to the multipart functions.

The `read_length` option defines the maximum amount of data to be received from the socket at once, in bytes.

The `read_timeout` option defines the time Cowboy waits before that amount is received, in milliseconds.

13.6 Transfer and content decoding

Cowboy will by default decode the chunked transfer-encoding if any. It will not decode any content-encoding by default.

The first time you call a body function you can set the `transfer_decode` and `content_decode` options. If the body was already started being read these options are simply ignored.

The following example shows how to set both options.

```
{ok, Data, Req2} = cowboy_req:body(Req, [
    {transfer_decode, fun transfer_decode/2, TransferState},
    {content_decode, fun content_decode/1}
]).
```

13.7 Reading a form urlencoded body

You can directly obtain a list of key/value pairs if the body was sent using the application/x-www-form-urlencoded content-type.

```
{ok, KeyValues, Req2} = cowboy_req:body_qs(Req).
```

You can then retrieve an individual value from that list.

```
{_, Lang} = lists:keyfind(lang, 1, KeyValues).
```

You should not attempt to match on the list as the order of the values is undefined.

By default Cowboy will reject bodies with a size above 64KB when using this function. You can override this limit by setting the `length` option.

```
{ok, KeyValues, Req2} = cowboy_req:body_qs(Req, [{length, 2000000}]).
```

Chapter 14

Sending a response

The Req object also allows you to send a response.

You can only send one response. Any other attempt will trigger a crash. The response may be sent in one go or with its body streamed by chunks of arbitrary size.

You can also set headers or the response body in advance and Cowboy will use them when you finally do reply.

14.1 Reply

You can send a reply with no particular headers or body. Cowboy will make sure to send the mandatory headers with the response.

```
Req2 = cowboy_req:reply(200, Req).
```

You can define headers to be sent with the response. Note that header names must be lowercase. Again, Cowboy will make sure to send the mandatory headers with the response.

```
Req2 = cowboy_req:reply(303, [
    {<<"location">>, <<"http://ninenines.eu">>}
], Req).
```

You can override headers that Cowboy would send otherwise. Any header set by the user will be used over the ones set by Cowboy. For example, you can advertise yourself as a different server.

```
Req2 = cowboy_req:reply(200, [
    {<<"server">>, <<"yaws">>}
], Req).
```

We also saw earlier how to force close the connection by overriding the connection header.

Finally, you can also send a body with the response. Cowboy will automatically set the content-length header if you do. We recommend that you set the content-type header so the client may know how to read the body.

```
Req2 = cowboy_req:reply(200, [
    {<<"content-type">>, <<"text/plain">>}
], "Hello world!", Req).
```

Here is the same example but sending HTML this time.

```
Req2 = cowboy_req:reply(200, [
    {<<"content-type">>, <<"text/html">>}
], "<html><head>Hello world!</head><body><p>Hats off!</p></body></html>", Req).
```

Note that the reply is sent immediately.

14.2 Chunked reply

You can also stream the response body. First, you need to initiate the reply by sending the response status code. Then you can send the body in chunks of arbitrary size.

```
Req2 = cowboy_req:chunked_reply(200, Req),
cowboy_req:chunk("Hello...", Req2),
cowboy_req:chunk("chunked...", Req2),
cowboy_req:chunk("world!!", Req2).
```

You should make sure to match on `ok` as an error may be returned.

While it is possible to send a chunked response without a content-type header, it is still recommended. You can set this header or any other just like for normal replies.

```
Req2 = cowboy_req:chunked_reply(200, [
    {<<"content-type">>, <<"text/html">>}
], Req),
cowboy_req:chunk("<html><head>Hello world!</head>", Req2),
cowboy_req:chunk("<body><p>Hats off!</p></body></html>", Req2).
```

Note that the reply and each chunk following it are sent immediately.

14.3 Preset response headers

You can define response headers in advance. They will be merged into the headers given in the reply call. Headers in the reply call override preset response headers which override the default Cowboy headers.

```
Req2 = cowboy_req:set_resp_header(<<"allow">>, "GET", Req).
```

You can check if a response header has already been set. This will only check the response headers that you set, and not the ones Cowboy will add when actually sending the reply.

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

It will return `true` if the header is defined, and `false` otherwise.

Finally, you can also delete a preset response header if needed. If you do, it will not be sent.

```
Req2 = cowboy_req:delete_resp_header(<<"allow">>, Req).
```

14.4 Preset response body

You can set the response body in advance. Note that this body will be ignored if you then choose to send a chunked reply, or if you send a reply with an explicit body.

```
Req2 = cowboy_req:set_resp_body("Hello world!", Req).
```

You can also set a fun that will be called when it is time to send the body. There are three different ways of doing that.

If you know the length of the body that needs to be sent, you should specify it, as it will help clients determine the remaining download time and allow them to inform the user.

```
F = fun (Socket, Transport) ->
    Transport:send(Socket, "Hello world!")
end,
Req2 = cowboy_req:set_resp_body_fun(12, F, Req).
```

If you do not know the length of the body, you should use a chunked response body fun instead.

```
F = fun (SendChunk) ->
    Body = lists:duplicate(random:uniform(1024, $a)),
    SendChunk(Body)
end,
Req2 = cowboy_req:set_resp_body_fun(chunked, F, Req).
```

Finally, you can also send data on the socket directly, without knowing the length in advance. Cowboy may be forced to close the connection at the end of the response though depending on the protocol capabilities.

```
F = fun (Socket, Transport) ->
    Body = lists:duplicate(random:uniform(1024, $a)),
    Transport:send(Socket, Body)
end,
Req2 = cowboy_req:set_resp_body_fun(F, Req).
```

14.5 Sending files

You can send files directly from disk without having to read them. Cowboy will use the `sendfile` syscall when possible, which means that the file is sent to the socket directly from the kernel, which is a lot more performant than doing it from userland.

Again, it is recommended to set the size of the file if it can be known in advance.

```
F = fun (Socket, Transport) ->
    Transport:sendfile(Socket, "priv/styles.css")
end,
Req2 = cowboy_req:set_resp_body_fun(FileSize, F, Req).
```

Please see the Ranch guide for more information about sending files.

Chapter 15

Using cookies

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

Cowboy provides facilities for handling cookies. It is highly recommended to use them instead of writing your own, as the implementation of cookies can vary greatly between clients.

Cookies are stored client-side and sent with every subsequent request that matches the domain and path for which they were stored, including requests for static files. For this reason they can incur a cost which must be taken in consideration.

Also consider that, regardless of the options used, cookies are not to be trusted. They may be read and modified by any program on the user's computer, but also by proxies. You should always validate cookie values before using them. Do not store any sensitive information in cookies either.

When explicitly setting the domain, the cookie will be sent for the domain and all subdomains from that domain. Otherwise the current domain will be used. The same is true for the path.

When the server sets cookies, they will only be available for requests that are sent after the client receives the response.

Cookies are sent in HTTP headers, therefore they must have text values. It is your responsibility to encode any other data type. Also note that cookie names are de facto case sensitive.

Cookies can be set for the client session (which generally means until the browser is closed), or it can be set for a number of seconds. Once it expires, or when the server says the cookie must exist for up to 0 seconds, the cookie is deleted by the client. To avoid this while the user is browsing your site, you should set the cookie for every request, essentially resetting the expiration time.

Cookies can be restricted to secure channels. This typically means that such a cookie will only be sent over HTTPS, and that it will only be available by client-side scripts that run from HTTPS webpages.

Finally, cookies can be restricted to HTTP and HTTPS requests, essentially disabling their access from client-side scripts.

15.1 Setting cookies

By default, cookies you set are defined for the session.

```
SessionID = generate_session_id(),
Req2 = cowboy_req:set_resp_cookie(<<"sessionid">>, SessionID, [], Req).
```

You can also make them expire at a specific point in the future.

```
SessionID = generate_session_id(),
Req2 = cowboy_req:set_resp_cookie(<<"sessionid">>, SessionID, [
    {max_age, 3600}
], Req).
```

You can delete cookies that have already been set. The value is ignored.

```
Req2 = cowboy_req:set_resp_cookie(<<"sessionid">>, <<>>, [
    {max_age, 0}
], Req).
```

You can restrict them to a specific domain and path. For example, the following cookie will be set for the domain `my.example.org` and all its subdomains, but only on the path `/account` and all its subdirectories.

```
Req2 = cowboy_req:set_resp_cookie(<<"inaccount">>, <<"1">>, [
    {domain, "my.example.org"},
    {path, "/account"}
], Req).
```

You can restrict the cookie to secure channels, typically HTTPS.

```
SessionID = generate_session_id(),
Req2 = cowboy_req:set_resp_cookie(<<"sessionid">>, SessionID, [
    {secure, true}
], Req).
```

You can restrict the cookie to client-server communication only. Such a cookie will not be available to client-side scripts.

```
SessionID = generate_session_id(),
Req2 = cowboy_req:set_resp_cookie(<<"sessionid">>, SessionID, [
    {http_only, true}
], Req).
```

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

15.2 Reading cookies

As we said, the client sends cookies with every request. But unlike the server, the client only sends the cookie name and value.

Cowboy provides two different ways to read cookies. You can either parse them as a list of key/value pairs, or match them into a map, optionally applying constraints to the values or providing a default if they are missing.

You can parse the cookies and then use standard library functions to access individual values.

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

You can match the cookies into 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, saving you a conversion step.

```
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.

Read more about constraints Chapter 8.

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, the query string is deemed invalid and the process will crash.

Chapter 16

Multipart requests

Multipart originates from MIME, an Internet standard that extends the format of emails. Multipart messages are a container for parts of any content-type.

For example, a multipart message may have a part containing text and a second part containing an image. This is what allows you to attach files to emails.

In the context of HTTP, multipart is most often used with the `multipart/form-data` content-type. This is the content-type you have to use when you want browsers to be allowed to upload files through HTML forms.

Multipart is of course not required for uploading files, it is only required when you want to do so through HTML forms.

You can read and parse multipart messages using the Req object directly.

Cowboy defines two functions that allows you to get information about each part and read their contents.

16.1 Structure

A multipart message is a list of parts. Parts may contain either a multipart message or a non-multipart content-type. This allows parts to be arranged in a tree structure, although this is a rare case as far as the Web is concerned.

16.2 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 use one part of the message for each form field. This means that a file input field will be sent in its own part, but the same applies to all other kinds of fields.

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 content-type of the files it sends this way, but you should not rely on it for determining the contents of the file. Proper investigation of the contents is recommended.

16.3 Checking the content-type

While there is a variety of multipart messages, the most common on the Web is `multipart/form-data`. It's the type of message being sent when an HTML form allows uploading files.

You can quickly figure out if a multipart message has been sent by parsing the `content-type` header.

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

16.4 Reading a multipart message

To read a message you have to iterate over all its parts. Then, for each part, you can inspect its headers and read its body.

```
multipart(Req) ->
    case cowboy_req:part(Req) of
        {ok, _Headers, Req2} ->
            {ok, _Body, Req3} = cowboy_req:part_body(Req2),
            multipart(Req3);
        {done, Req2} ->
            Req2
    end.
```

Parts do not have a size limit. When a part body is too big, Cowboy will return what it read so far and allow you to continue if you wish to do so.

The function `cow_multipart:form_data/1` can be used to quickly obtain information about a part from a `multipart/form-data` message. This function will tell you if the part is for a normal field or if it is a file being uploaded.

This can be used for example to allow large part bodies for files but crash when a normal field is too large.

```
multipart(Req) ->
    case cowboy_req:part(Req) of
        {ok, Headers, Req2} ->
            Req4 = case cow_multipart:form_data(Headers) of
                {data, _FieldName} ->
                    {ok, _Body, Req3} = cowboy_req:part_body(Req2),
                    Req3;
                {file, _FieldName, _Filename, _CType, _CTransferEncoding} ->
                    stream_file(Req2)
            end,
            multipart(Req4);
        {done, Req2} ->
            Req2
    end.

stream_file(Req) ->
    case cowboy_req:part_body(Req) of
        {ok, _Body, Req2} ->
            Req2;
        {more, _Body, Req2} ->
            stream_file(Req2)
    end.
```

By default the body chunk Cowboy will return is limited to 8MB. This can of course be overriden. Both functions can take a second argument, the same list of options that will be passed to `cowboy_req:body/2` function.

16.5 Skipping unwanted parts

If you do not want to read a part's body, you can skip it. Skipping is easy. If you do not call the function to read the part's body, Cowboy will automatically skip it when you request the next part.

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

```
multipart(Req) ->
    case cowboy_req:part(Req) of
        {ok, _Headers, Req2} ->
            multipart(Req2);
        {done, Req2} ->
            Req2
    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.

Note that the skipping rate may not be adequate for your application. If you observe poor performance when skipping, you might want to consider manually skipping by calling the `cowboy_req:part_body/1` function directly.

And if you started reading the message but decide that you do not need the remaining parts, you can simply stop reading entirely and Cowboy will automatically figure out what to do.

Part V

REST

Chapter 17

REST principles

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

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

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

17.1 REST architecture

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

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

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

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

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

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

17.2 Resources and resource identifiers

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

17.3 Resource representations

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

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

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

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

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

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

17.4 Self-descriptive messages

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

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

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

17.5 Hypermedia as the engine of application state

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

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

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

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

Chapter 18

REST handlers

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

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

18.1 Initialization

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

```
init(Req, _Opts) ->
    {cowboy_rest, Req, #state{}}.
```

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

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

18.2 Methods

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

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

18.3 Callbacks

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

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

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

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

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

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

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

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

18.4 Meta data

Cowboy will set informative meta values at various points of the execution. You can retrieve them using `cowboy_req:meta/{2,3}`. The values are defined in the following table.

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

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

18.5 Response headers

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

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

Chapter 19

REST flowcharts

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

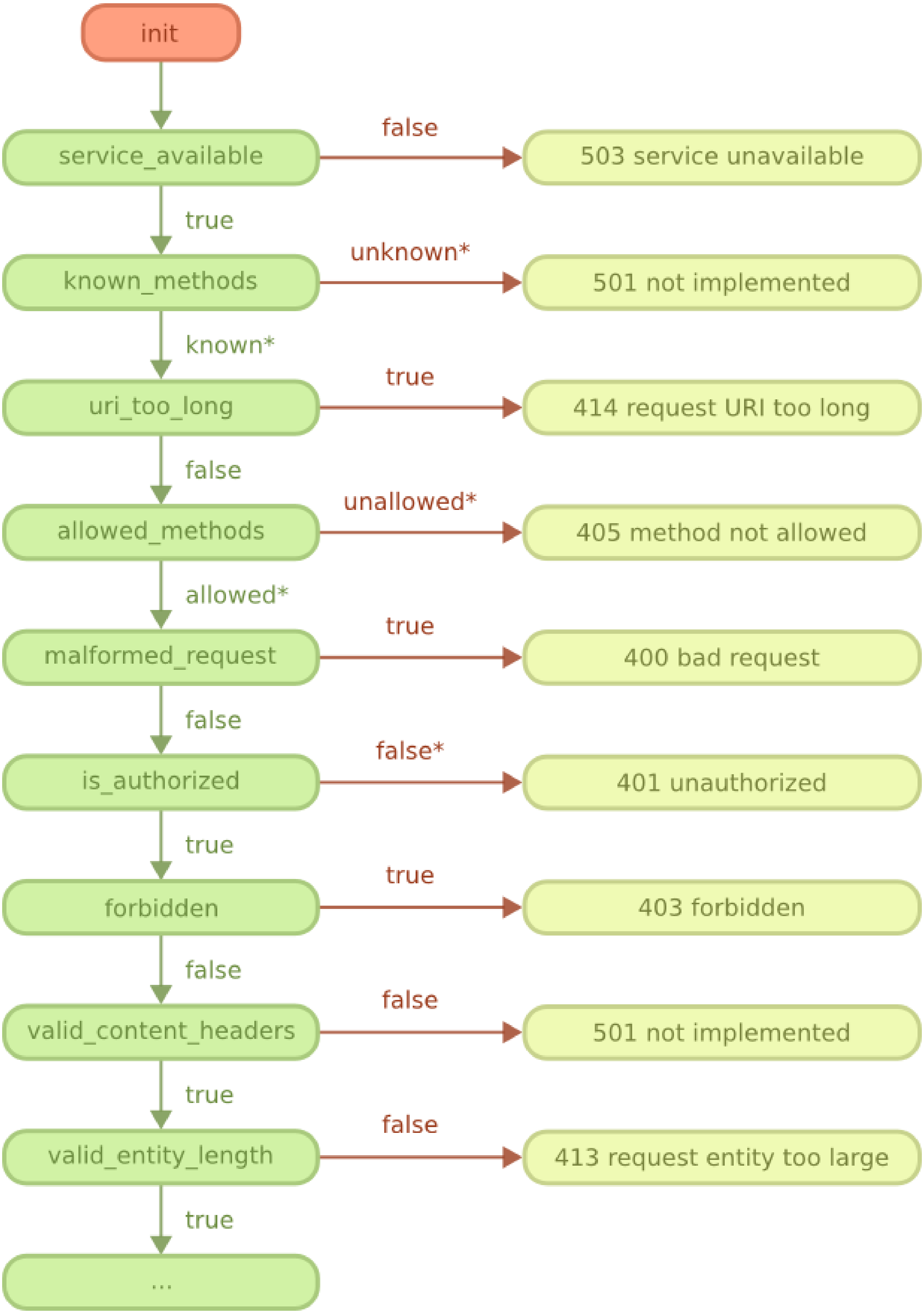
There are four main paths that requests may follow. One for the method OPTIONS; one for the methods GET and HEAD; one for the methods PUT, POST and PATCH; and one for the method DELETE.

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

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

19.1 Start

All requests start from here.


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
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.

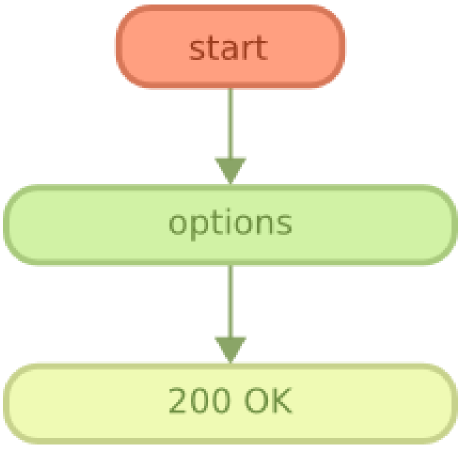
The `known_methods` and `allowed_methods` callbacks return a list of methods. Cowboy then checks if the request method is in the list, and stops otherwise.

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.

19.2 OPTIONS method

This diagram only applies to OPTIONS requests.

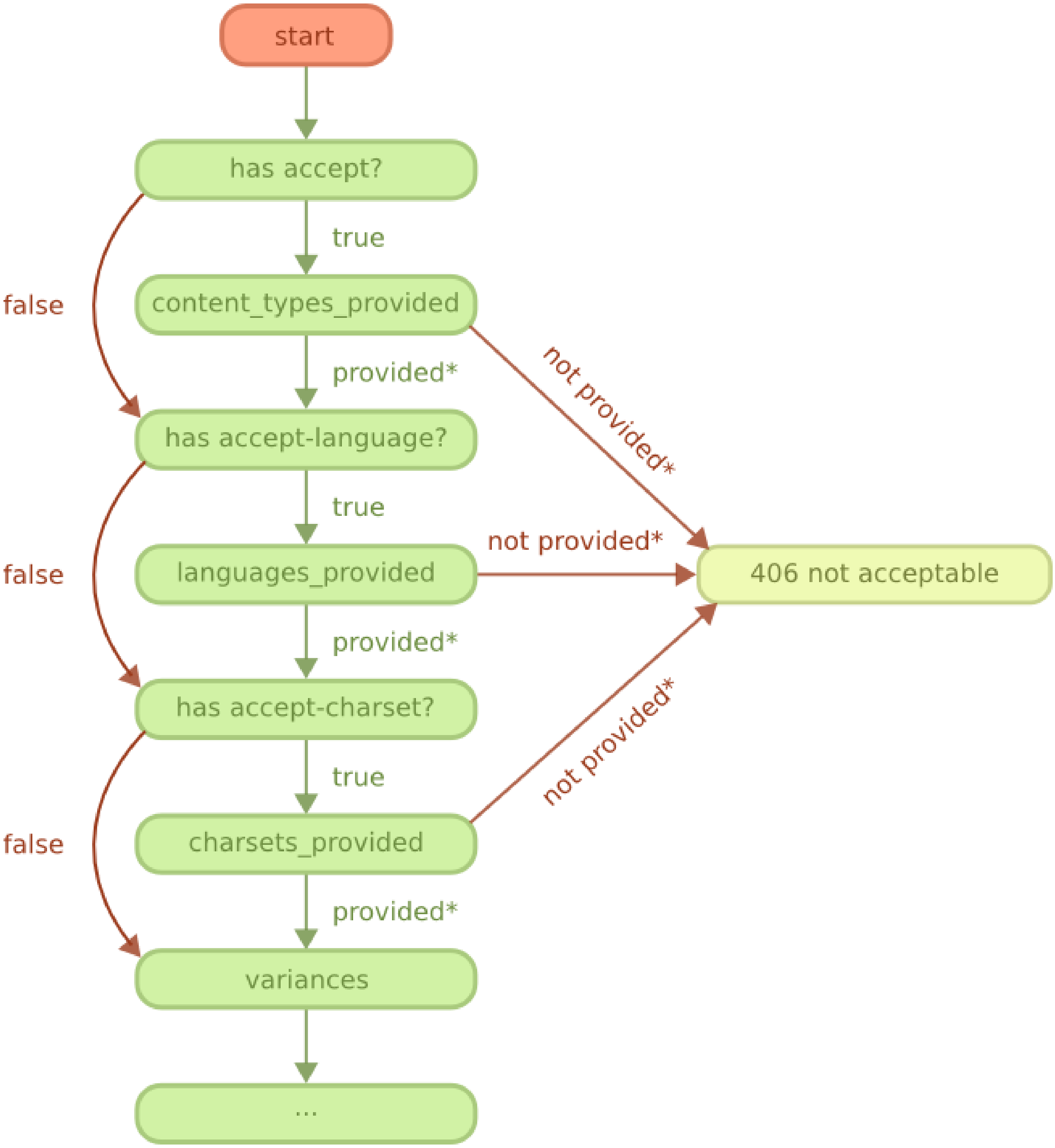


The `options` callback may be used to add information about the resource, such as media types or languages provided; allowed methods; any extra information. A response body may also be set, although clients should not be expected to read it.

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

19.3 Content negotiation

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


start
has accept?
true
false
content_types_provided
provided*
not provided*
has accept-language?
true
false
languages_provided
not provided*
406 not acceptable
provided*
has accept-charset?
true
false
charsets_provided
not provided*
provided*
variances
...

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

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

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

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

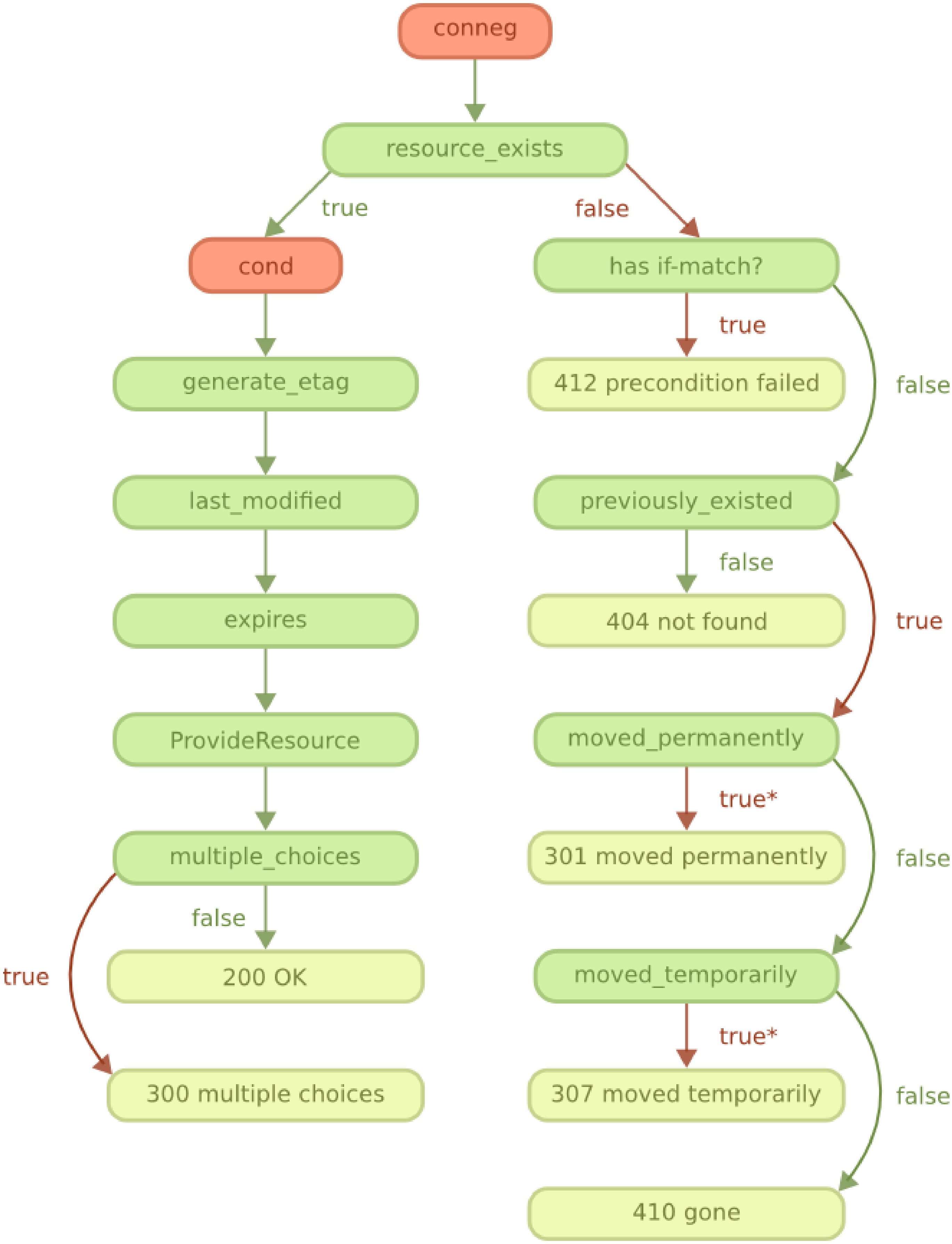
The selected content-type, language and charset are saved as meta values in the Req object. You **should** use the appropriate representation if you set a response body manually (alongside an error code, for example).

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

19.4 GET and HEAD methods

This diagram only applies to GET and HEAD requests.

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


conneg
resource_exists
true
false
cond
has if-match?
true
false
generate_etag
412 precondition failed
last_modified
previously_existed
false
true
expires
404 not found
ProvideResource
moved_permanently
true*
false
multiple_choices
301 moved permanently
false
true
200 OK
moved_temporarily
true*
false
300 multiple choices
307 moved temporarily
410 gone

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

Cowboy prepares the response by first retrieving metadata about the representation, then by calling the `ProvideResource` callback. This is the callback you defined for each content-types you returned from `content_types_provided`. This callback returns the body that will be sent back to the client, or a fun if the body must be streamed.

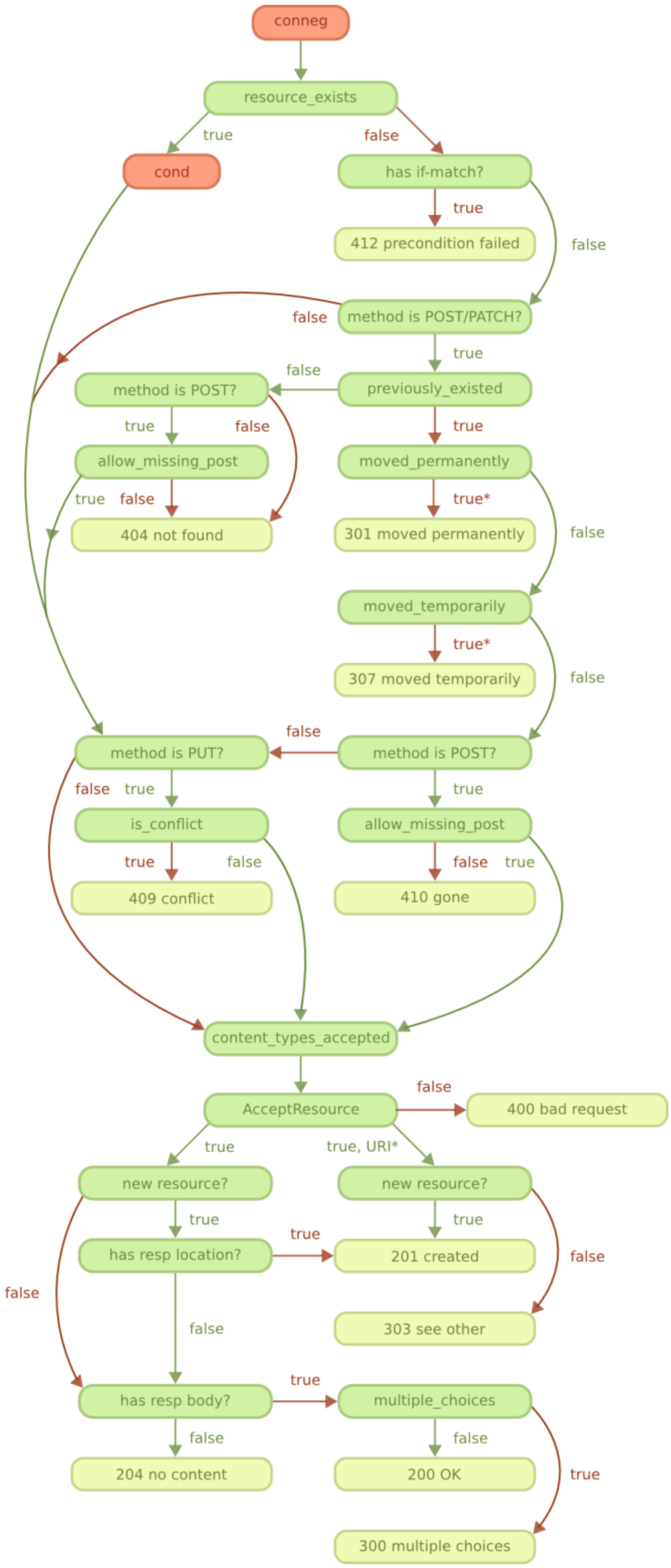
When the resource does not exist, Cowboy will figure out whether the resource existed previously, and if so whether it was moved elsewhere in order to redirect the client to the new URI.

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

19.5 PUT, POST and PATCH methods

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

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


conneg
resource_exists
true
false
cond
has if-match?
true
412 precondition failed
false
false
method is POST/PATCH?
true
false
method is POST?
previously_existed
true
false
true
allow_missing_post
moved_permanently
true
false
true*
404 not found
301 moved permanently
false
moved_temporarily
true*
307 moved temporarily
false
false
method is PUT?
method is POST?
false
true
true
is_conflict
allow_missing_post
true
false
false
true
409 conflict
410 gone
content_types_accepted
false
AcceptResource
400 bad request
true
true, URI*
new resource?
new resource?
true
true
true
has resp location?
201 created
false
false
false
303 see other
true
has resp body?
multiple_choices
false
false
204 no content
200 OK
true
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_accepted` step. For other methods, Cowboy will figure out whether the resource existed previously, and if so whether it was moved elsewhere. If the resource is truly non-existent, the method is POST and the call for `allow_missing_post` returns `true`, then Cowboy will move on to the `content_types_accepted` step. Otherwise the request processing ends there.

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

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

This callback may return one of three different return values.

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

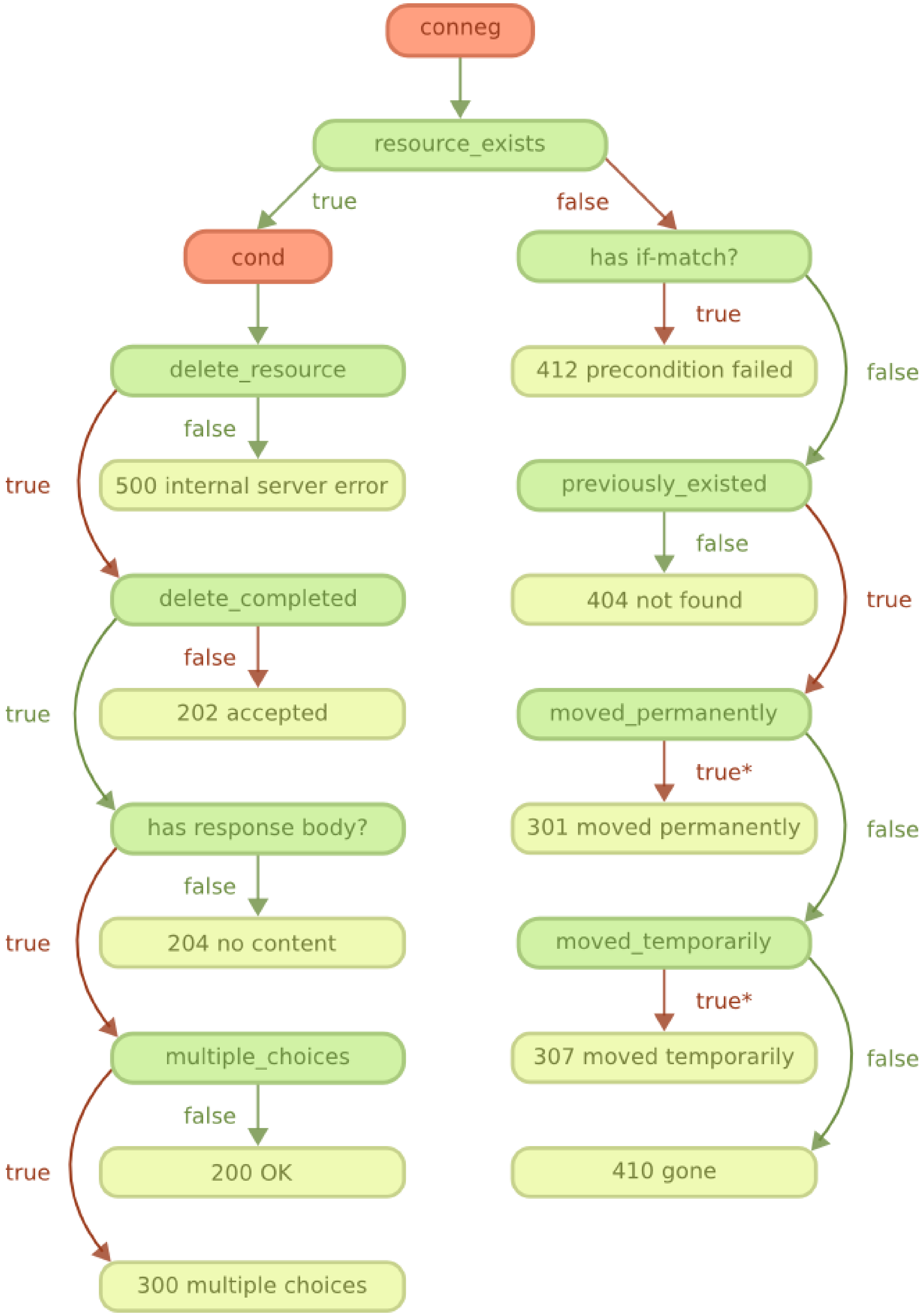
If the method is POST, then you may return `true` with an URI of where the resource has been created. This is especially useful for writing handlers for collections.

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

19.6 DELETE method

This diagram only applies to DELETE requests.

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


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

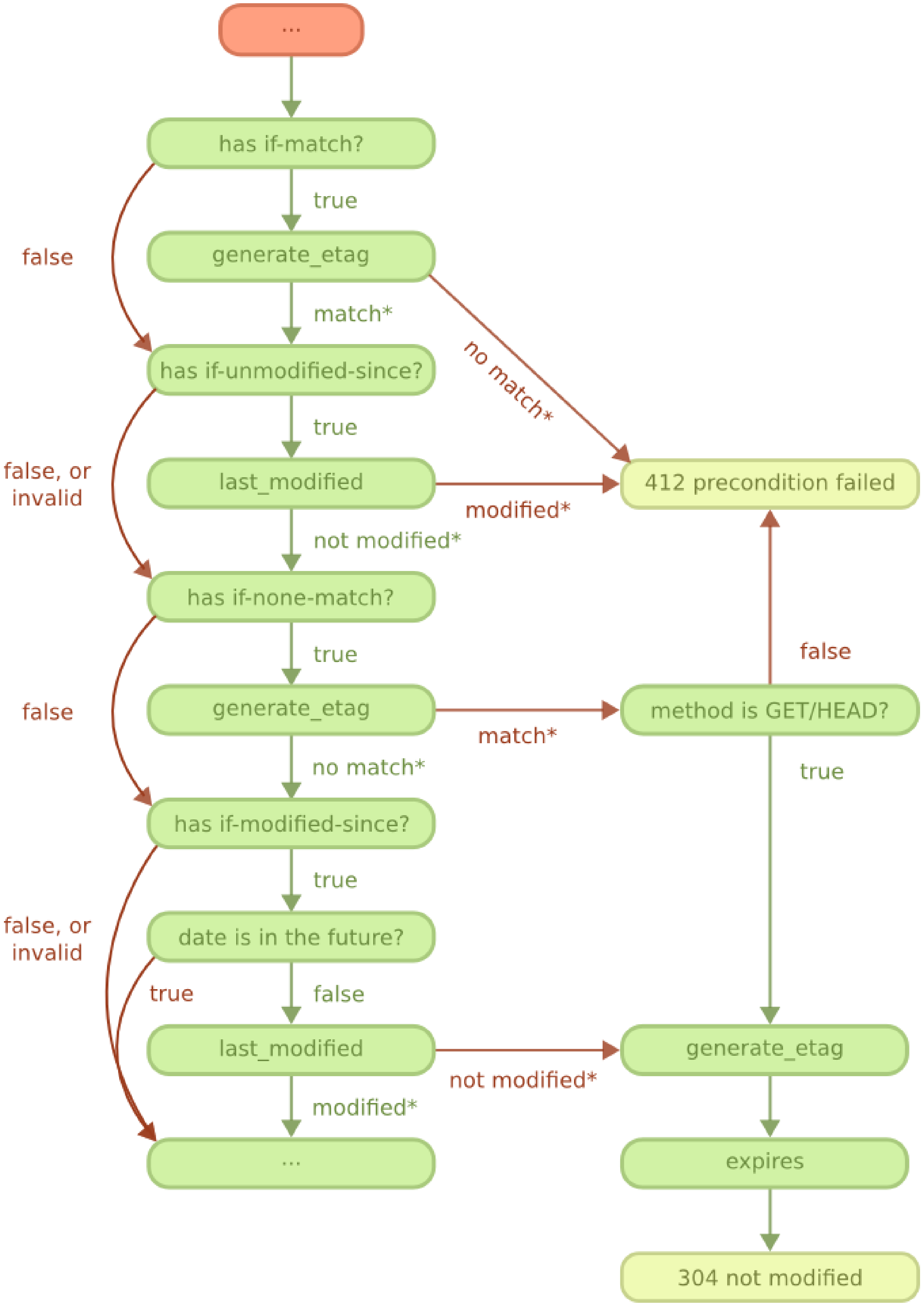
To finish, Cowboy checks if you set a response body, and depending on that, sends the appropriate response.

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.

19.7 Conditional requests

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


...
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 20

Designing a resource handler

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

20.1 The service

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

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

20.2 Type of resource handler

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

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

20.3 Collection handler

Skip this section if you are not doing a collection.

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

What methods should I implement?

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

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

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

The next methods are more rarely allowed.

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

DELETE is used to delete the entire collection.

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

20.4 Single resource handler

Skip this section if you are doing a collection.

What methods should I implement?

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

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

POST is used to update the resource.

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

DELETE is used to delete the resource.

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

20.5 The resource

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

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

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

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

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

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

20.6 Representations

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

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

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

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

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

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

20.7 Redirections

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

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

20.8 The request

Do we need to perform extra checks to make sure the request is valid? Cowboy will do many checks when receiving the request already, do we need more? Note that this only applies to the request-line and headers of the request, and not the body. Implement `malformed_request`.

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

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

20.9 OPTIONS method

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

20.10 GET and HEAD methods

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

20.11 PUT, POST and PATCH methods

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

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

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

20.12 DELETE methods

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

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

Part VI

Websocket

Chapter 21

The Websocket protocol

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

21.1 Description

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

Websocket connections are asynchronous, unlike HTTP. This means that not only can the client send frames to the server at any time, but the server can also send frames to the client without the client initiating anything other than the Websocket connection itself. This allows the server to push data to the client directly.

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

21.2 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 x-webkit-deflate-frame compression draft which is being used by some browsers to reduce the size of data being transmitted. Cowboy will automatically use compression as long as the `compress` protocol option is set when starting the listener.

Chapter 22

Handling Websocket connections

A special handler is required for handling Websocket connections. Websocket handlers allow you to initialize the connection, handle incoming frames from the socket, handle incoming Erlang messages and then clean up on termination.

Websocket handlers essentially act as a bridge between the client and the Erlang system. They will typically do little more than socket communication and decoding/encoding of frames.

22.1 Initialization

First, the `init/2` callback is called. This callback is common to all handlers. To establish a Websocket connection, this function must return a `ws` tuple.

```
init(Req, _Opts) ->
    {cowboy_websocket, Req, #state{}}.
```

Upon receiving this tuple, Cowboy will switch to the code that handles Websocket connections and perform the handshake immediately.

If the sec-websocket-protocol header was sent with the request for establishing a Websocket connection, then the Websocket handler **must** select one of these subprotocol and send it back to the client, otherwise the client might decide to close the connection, assuming no correct subprotocol was found.

```
init(Req, _Opts) ->
    case cowboy_req:parse_header(<<"sec-websocket-protocol">>, Req) of
        undefined ->
            {ok, Req, #state{}};
        Subprotocols ->
            case lists:keymember(<<"mychat2">>, 1, Subprotocols) of
                true ->
                    Req2 = cowboy_req:set_resp_header(<<"sec-websocket-protocol">>,
                        <<"mychat2">>, Req),
                    {ok, Req2, #state{}};
                false ->
                    {stop, Req, undefined}
            end
    end.
```

It is not recommended to wait too long inside the `init/2` function. Any extra initialization may be done after returning by sending yourself a message before doing anything. Any message sent to `self()` from `init/2` is guaranteed to arrive before any frames from the client.

It is also very easy to ensure that this message arrives before any message from other processes by sending it before registering or enabling timers.

```
init(Req, _Opts) ->
    self() ! post_init,
    %% Register process here...
    {cowboy_websocket, Req, #state{}}.

websocket_info(post_init, Req, State) ->
    %% Perform post_init initialization here...
    {ok, Req, State}.
```

22.2 Handling frames from the client

Cowboy will call `websocket_handle/3` whenever a text, binary, ping or pong frame arrives from the client. Note that in the case of ping and pong frames, no action is expected as Cowboy automatically replies to ping frames.

The handler can decide to send frames to the socket, stop or just continue without sending anything.

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

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

22.3 Handling Erlang messages

Cowboy will call `websocket_info/3` whenever an Erlang message arrives.

The handler can decide to send frames to the socket, stop or just continue without sending anything.

The following snippet forwards any `log` message to the socket and ignores all others.

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

22.4 Sending frames to the socket

Cowboy allows sending either a single frame or a list of frames to the socket, in which case the frames are sent sequentially. Any frame can be sent: text, binary, ping, pong or close frames.

The following example sends three frames using a single `reply` tuple.

```
websocket_info(hello_world, Req, State) ->
    {reply, [
        {text, "Hello"},
        {text, <<"world!">>},
        {binary, <<0:8000>>}
    ], Req, State};
%% More websocket_info/3 clauses here...
```

Note that the payload for text and binary frames is of type `iodata()`, meaning it can be either a `binary()` or an `iolist()`.

Sending a `close` frame will immediately initiate the closing of the Websocket connection. Be aware that any additional frames sent by the client or any Erlang messages waiting to be received will not be processed. Also note that when replying a list of frames that includes close, any frame found after the close frame will not be sent.

22.5 Ping and timeout

The biggest performance improvement you can do when dealing with a huge number of Websocket connections is to reduce the number of timers that are started on the server. A common use of timers when dealing with connections is for sending a ping every once in a while. This should be done exclusively on the client side. Indeed, a server handling one million Websocket connections will perform a lot better when it doesn't have to handle one million extra timers too!

Cowboy will automatically respond to ping frames sent by the client. It will still forward the frame to the handler for informative purpose, but no further action is required.

Cowboy can be configured to automatically close the Websocket connection when no data arrives on the socket. It is highly recommended to configure a timeout for it, as otherwise you may end up with zombie "half-connected" sockets that may leave the process alive forever.

A good timeout value is 60 seconds.

```
init(Req, _Opts) ->
    {cowboy_websocket, Req, #state{}, 60000}.
```

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

22.6 Hibernate

Most tuples returned from handler callbacks can include an extra value `hibernate`. After doing any necessary operations following the return of the callback, Cowboy will hibernate the process.

It is highly recommended to hibernate processes that do not handle much traffic. It is a good idea to hibernate all connections by default and investigate only when you start noticing increased CPU usage.

22.7 Supporting older browsers

Unfortunately Websocket is a relatively recent technology, which means that not all browsers support it. A library like Bullet can be used to emulate Websocket connections on older browsers.

Part VII

Internals

Chapter 23

Architecture

Cowboy is a lightweight HTTP server.

It is built on top of Ranch. Please see the Ranch guide for more information.

23.1 One process per connection

It uses only one process per connection. The process where your code runs is the process controlling the socket. Using one process instead of two allows for lower memory usage.

Because there can be more than one request per connection with the keepalive feature of HTTP/1.1, that means the same process will be used to handle many requests.

Because of this, you are expected to make sure your process cleans up before terminating the handling of the current request. This may include cleaning up the process dictionary, timers, monitoring and more.

23.2 Binaries

It uses 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.

23.3 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.

23.4 Max connections

By default the maximum number of active connections is set to a generally accepted big enough number. This is meant to prevent having too many processes performing potentially heavy work and slowing everything else down, or taking up all the memory.

Disabling this feature, by setting the `{max_connections, infinity}` protocol option, would give you greater performance when you are only processing short-lived requests.

Chapter 24

Dealing with broken clients

There exists a very large number of implementations for the HTTP protocol. Most widely used clients, like browsers, follow the standard quite well, but others may not. In particular custom enterprise clients tend to be very badly written.

Cowboy tries to follow the standard as much as possible, but is not trying to handle every possible special cases. Instead Cowboy focuses on the cases reported in the wild, on the public Web.

That means clients that ignore the HTTP standard completely may fail to understand Cowboy's responses. There are of course workarounds. This chapter aims to cover them.

24.1 Lowercase headers

Cowboy converts all headers it receives to lowercase, and similarly sends back headers all in lowercase. Some broken HTTP clients have issues with that.

A simple way to solve this is to create an `onresponse` hook that will format the header names with the expected case.

```
capitalize_hook(Status, Headers, Body, Req) ->
    Headers2 = [{cowboy_bstr:capitalize_token(N), V}
        || {N, V} <- Headers],
    cowboy_req:reply(Status, Headers2, Body, Req).
```

Note that SPDY clients do not have that particular issue because the specification explicitly says all headers are lowercase, unlike HTTP which allows any case but treats them as case insensitive.

24.2 Camel-case headers

Sometimes it is desirable to keep the actual case used by clients, for example when acting as a proxy between two broken implementations. There is no easy solution for this other than forking the project and editing the `cowboy_protocol` file directly.

24.3 Chunked transfer-encoding

Sometimes an HTTP client advertises itself as HTTP/1.1 but does not support chunked transfer-encoding. This is invalid behavior, as HTTP/1.1 clients are required to support it.

A simple workaround exists in these cases. By changing the Req object response state to `waiting_stream`, Cowboy will understand that it must use the identity transfer-encoding when replying, just like if it was an HTTP/1.0 client.

```
Req2 = cowboy_req:set(resp_state, waiting_stream).
```

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

Sub protocols

Sub protocols are used for creating new types of handlers that provide extra functionality in a reusable way. Cowboy uses this mechanism to provide its loop, REST and Websocket handlers.

This chapter will explain how to create your own sub protocols and handler types.

26.1 Usage

To switch to a sub protocol, the `init/2` callback must return the name of the sub protocol module. Everything past this point is handled by the sub protocol.

```
init(Req, _Opts) ->
    {cowboy_websocket, Req, #state{}}.
```

The return value may also have a `Timeout` value and/or the atom `hibernate`. These options are useful for long living connections. When they are not provided, the timeout value defaults to `infinity` and the hibernate value to `run`.

The following snippet switches to the `my_protocol` sub protocol, sets the timeout value to 5 seconds and enables hibernation:

```
init(Req, _Opts) ->
    {my_protocol, Req, #state{}, 5000, hibernate}.
```

If a sub protocol does not make use of these options, it should crash if it receives anything other than the default values.

26.2 Upgrade

After the `init/2` function returns, Cowboy will then call the `upgrade/6` function. This is the only callback defined by the `cowboy_sub_protocol` behavior.

The function is named `upgrade` because it mimics the mechanism of HTTP protocol upgrades. For some sub protocols, like Websocket, an actual upgrade is performed. For others, like REST, this is only an upgrade at Cowboy's level and the client has nothing to do about it.

The upgrade callback receives the Req object, the middleware environment, the handler and its options, and the aforementioned timeout and hibernate values.

```
upgrade(Req, Env, Handler, HandlerOpts, Timeout, Hibernate) ->
    %% Sub protocol code here.
```

This callback is expected to behave like a middleware and to return an updated environment and Req object.

Sub protocols are expected to call the `cowboy_handler:terminate/4` function when they terminate. This function will make sure that the optional `terminate/3` callback is called, if present.

Chapter 27

Hooks

Hooks allow the user to customize Cowboy's behavior during specific operations.

27.1 Onresponse

The `onresponse` hook is called right before sending the response to the socket. It can be used for the purposes of logging responses, or for modifying the response headers or body. The best example is providing custom error pages.

Note that this function MUST NOT crash. Cowboy may or may not send a reply if this function crashes. If a reply is sent, the hook MUST explicitly provide all headers that are needed.

You can specify the `onresponse` hook when creating the listener.

```
cowboy:start_http(my_http_listener, 100,
    [{port, 8080}],
    [
        {env, [{dispatch, Dispatch}]},
        {onresponse, fun ?MODULE:custom_404_hook/4}
    ]
).
```

The following hook function will provide a custom body for 404 errors when it has not been provided before, and will let Cowboy proceed with the default response otherwise.

```
custom_404_hook(404, Headers, <<>>, Req) ->
    Body = <<"404 Not Found.">>,
    Headers2 = lists:keyreplace(<<"content-length">>, 1, Headers,
        {<<"content-length">>, integer_to_list(byte_size(Body))}),
    cowboy_req:reply(404, Headers2, Body, Req);
custom_404_hook(_, _, _, Req) ->
    Req.
```

Again, make sure to always return the last request object obtained.