next generation web framework for node.js

Introduction

Koa is a new web framework designed by the team behind Express, which aims to be a smaller, more expressive, and more robust foundation for web applications and APIs. By leveraging async functions, Koa allows you to ditch callbacks and greatly increase error-handling. Koa does not bundle any middleware within its core, and it provides an elegant suite of methods that make writing servers fast and enjoyable.

Installation

Koa requires **node v12** or higher for ES2015 and async function support.

You can quickly install a supported version of node with your favorite version manager:

```
$ nvm install 7
$ npm i koa
$ node my-koa-app.js
```

Application

A Koa application is an object containing an array of middleware functions which are composed and executed in a stack-like manner upon request. Koa is similar to many other middleware systems that you may have encountered such as Ruby's Rack, Connect, and so on - however a key design decision was made to provide high level "sugar" at the otherwise low-level middleware layer. This improves interoperability, robustness, and makes writing middleware much more enjoyable.

This includes methods for common tasks like content-negotiation, cache freshness, proxy support, and redirection among others. Despite supplying a reasonably large number of helpful methods Koa maintains a small footprint, as no middleware are bundled.

The obligatory hello world application:

```
const Koa = require('koa'  2.14.2  );
const app = new Koa();

app.use(async ctx => {
  ctx.body = 'Hello World';
});

app.listen(3000);
```

Save on **RunKit** Node 8 help URL: http

Cascading

Koa middleware cascade in a more traditional way as you may be used to with similar tools - this was previously difficult to make user friendly with node's use of callbacks. However with async functions we can achieve "true" middleware. Contrasting Connect's implementation which simply passes control through series of functions until one returns, Koa invoke "downstream", then control flows back "upstream".

The following example responds with "Hello World", however first the request flows through the `x-response-time` and `logging` middleware to mark when the request started, then yields control through the response middleware. When a middleware invokes `next()` the function suspends and passes control to the next middleware defined. After there are no more middleware to execute downstream, the stack will unwind and each middleware is resumed to perform its upstream behaviour.

```
const Koa = require('koa'  2.14.2  );
const app = new Koa();

// logger

app.use(async (ctx, next) => {
  await next();
  const rt = ctx.response.get('X-Response-Time');
  console.log(`${ctx.method} ${ctx.url} - ${rt}`);
});

// x-response-time

app.use(async (ctx, next) => {
  const start = Date.now();
  await next();
  const ms = Date.now() - start;
  ctx.set('X-Response-Time', `${ms}ms`);
});

// response

app.use(async ctx => {
  ctx.body = 'Hello World';
});

app.listen(3000);
```

Save on **RunKit** | Node 8 | help | URL: htt

Settings

Application settings are properties on the `app` instance, currently the following are supported:

- `app.env` defaulting to the **NODE_ENV** or "development"
- `app.keys` array of signed cookie keys
- `app.proxy` when true proxy header fields will be trusted
- `app.subdomainOffset` offset of `.subdomains` to ignore, default to 2
- `app.proxyIpHeader` proxy ip header, default to `X-Forwarded-For`

- `app.maxIpsCount` max ips read from proxy ip header, default to 0 (means infinity)

  You can pass the settings to the constructor:

```
const Koa = require('koa');
const app = new Koa({ proxy: true });
```

  or dynamically:

```
const Koa = require('koa');
const app = new Koa();
app.proxy = true;
```

app.listen(...)

A Koa application is not a 1-to-1 representation of an HTTP server. One or more Koa applications may be mounted together to form larger applications with a single HTTP server.

Create and return an HTTP server, passing the given arguments to `Server#listen()`. These arguments are documented on nodejs.org. The following is a useless Koa application bound to port `3000`:

```
const Koa = require('koa');
const app = new Koa();
app.listen(3000);
```

The `app.listen(...)` method is simply sugar for the following:

```
const http = require('http');
const Koa = require('koa');
```

```
const app = new Koa();
http.createServer(app.callback()).listen(3000);
```

This means you can spin up the same application as both HTTP and HTTPS or on multiple addresses:

```
const http = require('http');
const https = require('https');
const Koa = require('koa');
const app = new Koa();
http.createServer(app.callback()).listen(3000);
https.createServer(app.callback()).listen(3001);
```

app.callback()

Return a callback function suitable for the `http.createServer()` method to handle a request. You may also use this callback function to mount your Koa app in a Connect/Express app.

app.use(function)

Add the given middleware function to this application. `app.use()` returns `this`, so is chainable.

```
app.use(someMiddleware)
app.use(someOtherMiddleware)
app.listen(3000)
```

Is the same as

```
app.use(someMiddleware)
  .use(someOtherMiddleware)
  .listen(3000)
```

See Middleware for more information.

app.keys=

Set signed cookie keys.

These are passed to KeyGrip, however you may also pass your own `KeyGrip` instance. For example the following are acceptable:

```
app.keys = ['OEK5zjaAMPc3L6iK7PyUjCOziUH3rsrMKB9u8H07La1SkfwtuBoDnHa
app.keys = new KeyGrip(['OEK5zjaAMPc3L6iK7PyUjCOziUH3rsrMKB9u8H07La1
```

For security reasons, please ensure that the key is long enough and random.

These keys may be rotated and are used when signing cookies with the `{ signed: true }` option:

```
ctx.cookies.set('name', 'tobi', { signed: true });
```

app.context

`app.context` is the prototype from which `ctx` is created. You may add additional properties to `ctx` by editing `app.context`. This is useful for adding properties or methods to `ctx` to be used across your entire app, which may be more performant (no middleware)

and/or easier (fewer `require()`s) at the expense of relying more on `ctx`, which could be considered an anti-pattern.

For example, to add a reference to your database from `ctx`:

```
app.context.db = db();

app.use(async ctx => {
  console.log(ctx.db);
});
```

Note:

- Many properties on `ctx` are defined using getters, setters, and `Object.defineProperty()`. You can only edit these properties (not recommended) by using `Object.defineProperty()` on `app.context`. See https://github.com/koajs/koa/issues/652.
- Mounted apps currently use their parent's `ctx` and settings. Thus, mounted apps are really just groups of middleware.

Error Handling

By default outputs all errors to stderr unless `app.silent` is `true`. The default error handler also won't output errors when `err.status` is `404` or `err.expose` is `true`. To perform custom error-handling logic such as centralized logging you can add an "error" event listener:

```
app.on('error', err => {
  log.error('server error', err)
});
```

If an error is in the req/res cycle and it is *not* possible to respond to the client, the `Context` instance is also passed:

```
app.on('error', (err, ctx) => {
  log.error('server error', err, ctx)
});
```

When an error occurs *and* it is still possible to respond to the client, aka no data has been written to the socket, Koa will respond appropriately with a 500 "Internal Server Error". In either case an app-level "error" is emitted for logging purposes.

Context

A Koa Context encapsulates node's `request` and `response` objects into a single object which provides many helpful methods for writing web applications and APIs. These operations are used so frequently in HTTP server development that they are added at this level instead of a higher level framework, which would force middleware to re-implement this common functionality.

A `Context` is created *per* request, and is referenced in middleware as the receiver, or the `ctx` identifier, as shown in the following snippet:

```
app.use(async ctx => {
  ctx; // is the Context
  ctx.request; // is a Koa Request
```

```
  ctx.response; // is a Koa Response
});
```

Many of the context's accessors and methods simply delegate to their `ctx.request` or `ctx.response` equivalents for convenience, and are otherwise identical. For example `ctx.type` and `ctx.length` delegate to the `response` object, and `ctx.path` and `ctx.method` delegate to the `request`.

API

`Context` specific methods and accessors.

ctx.req

Node's `request` object.

ctx.res

Node's `response` object.

Bypassing Koa's response handling is **not supported**. Avoid using the following node properties:

- `res.statusCode`
- `res.writeHead()`
- `res.write()`
- `res.end()`

ctx.request

A Koa `Request` object.

ctx.response

A Koa `Response` object.

ctx.state

The recommended namespace for passing information through middleware and to your frontend views.

```
ctx.state.user = await User.find(id);
```

ctx.app

Application instance reference.

ctx.app.emit

Koa applications extend an internal EventEmitter. `ctx.app.emit` emits an event with a type, defined by the first argument. For each event you can hook up "listeners", which is a function that is called when the event is emitted. Consult the error handling docs for more information.

ctx.cookies.get(name, [options])

Get cookie `name` with `options`:

- `signed` the cookie requested should be signed

Koa uses the cookies module where options are simply passed.

ctx.cookies.set(name, value, [options])

Set cookie `name` to `value` with `options`:

- `maxAge`: a number representing the milliseconds from `Date.now()` for expiry.
- `expires`: a `Date` object indicating the cookie's expiration date (expires at the end of session by default).
- `path`: a string indicating the path of the cookie (`/` by default).
- `domain`: a string indicating the domain of the cookie (no default).
- `secure`: a boolean indicating whether the cookie is only to be sent over HTTPS (`false` by default for HTTP, `true` by default for HTTPS). Read more about this option.

- `httpOnly`: a boolean indicating whether the cookie is only to be sent over HTTP(S), and not made available to client JavaScript (`true` by default).
- `sameSite`: a boolean or string indicating whether the cookie is a "same site" cookie (`false` by default). This can be set to `'strict'`, `'lax'`, `'none'`, or `true` (which maps to `'strict'`).
- `signed`: a boolean indicating whether the cookie is to be signed (`false` by default). If this is true, another cookie of the same name with the `.sig` suffix appended will also be sent, with a 27-byte url-safe base64 SHA1 value representing the hash of *cookie-name*=*cookie-value* against the first Keygrip key. This signature key is used to detect tampering the next time a cookie is received.
- `overwrite`: a boolean indicating whether to overwrite previously set cookies of the same name (`false` by default). If this is true, all cookies set during the same request with the same name (regardless of path or domain) are filtered out of the Set-Cookie header when setting this cookie.

Koa uses the cookies module where options are simply passed.

ctx.throw([status], [msg], [properties])

Helper method to throw an error with a `.status` property defaulting to `500` that will allow Koa to respond appropriately. The following combinations are allowed:

```
ctx.throw(400);
ctx.throw(400, 'name required');
ctx.throw(400, 'name required', { user: user });
```

For example `ctx.throw(400, 'name required')` is equivalent to:

```
const err = new Error('name required');
err.status = 400;
err.expose = true;
throw err;
```

Note that these are user-level errors and are flagged with `err.expose` meaning the messages are appropriate for client responses, which is typically not the case for error messages since you do not want to leak failure details.

You may optionally pass a `properties` object which is merged into the error as-is, useful for decorating machine-friendly errors which are reported to the requester upstream.

```
ctx.throw(401, 'access_denied', { user: user });
```

Koa uses http-errors to create errors. `status` should only be passed as the first parameter.

ctx.assert(value, [status], [msg], [properties])

Helper method to throw an error similar to `.throw()` when `!value`. Similar to node's assert() method.

```
ctx.assert(ctx.state.user, 401, 'User not found. Please login!');
```

Koa uses http-assert for assertions.

ctx.respond

To bypass Koa's built-in response handling, you may explicitly set `ctx.respond = false;`. Use this if you want to write to the raw `res` object instead of letting Koa handle the response for you.

Note that using this is **not** supported by Koa. This may break intended functionality of Koa middleware and Koa itself. Using this property is considered a hack and is only a convenience to those wishing to use traditional `fn(req, res)` functions and middleware within Koa.

Request aliases

The following accessors and alias Request equivalents:

- ctx.header
- ctx.headers
- ctx.method
- ctx.method=
- ctx.url
- ctx.url=
- ctx.originalUrl
- ctx.origin
- ctx.href
- ctx.path
- ctx.path=
- ctx.query
- ctx.query=
- ctx.querystring
- ctx.querystring=
- ctx.host
- ctx.hostname
- ctx.fresh
- ctx.stale
- ctx.socket
- ctx.protocol
- ctx.secure
- ctx.ip
- ctx.ips
- ctx.subdomains
- ctx.is()
- ctx.accepts()
- ctx.acceptsEncodings()
- ctx.acceptsCharsets()
- ctx.acceptsLanguages()
- ctx.get()

Response aliases

The following accessors and alias Response equivalents:

- ctx.body
- ctx.body=
- ctx.status
- ctx.status=
- ctx.message
- ctx.message=
- ctx.length=
- ctx.length

- `ctx.type=`
- `ctx.type`
- `ctx.headerSent`
- `ctx.redirect()`
- `ctx.attachment()`
- `ctx.set()`
- `ctx.append()`
- `ctx.remove()`
- `ctx.lastModified=`
- `ctx.etag=`

Request

A Koa `Request` object is an abstraction on top of node's vanilla request object, providing additional functionality that is useful for every day HTTP server development.

API

request.header

Request header object. This is the same as the `headers` field on node's `http.IncomingMessage`.

request.header=

Set request header object.

request.headers

Request header object. Alias as `request.header`.

request.headers=

Set request header object. Alias as `request.header=`.

request.method

Request method.

request.method=

Set request method, useful for implementing middleware such as `methodOverride()`.

request.length

Return request Content-Length as a number when present, or `undefined`.

request.url

Get request URL.

request.url=

Set request URL, useful for url rewrites.

request.originalUrl

Get request original URL.

request.origin

Get origin of URL, include `protocol` and `host`.

```
ctx.request.origin
// => http://example.com
```

request.href

Get full request URL, include `protocol`, `host` and `url`.

```
ctx.request.href;
// => http://example.com/foo/bar?q=1
```

request.path

Get request pathname.

request.path=

Set request pathname and retain query-string when present.

request.querystring

Get raw query string void of `?`.

request.querystring=

Set raw query string.

request.search

Get raw query string with the `?`.

request.search=

Set raw query string.

request.host

Get host (hostname:port) when present. Supports `X-Forwarded-Host` when `app.proxy` is **true**, otherwise `Host` is used.

request.hostname

Get hostname when present. Supports `X-Forwarded-Host` when `app.proxy` is **true**, otherwise `Host` is used.

If host is IPv6, Koa delegates parsing to [WHATWG URL API](), *Note* This may impact performance.

request.URL

Get WHATWG parsed URL object.

request.type

Get request `Content-Type` void of parameters such as "charset".

```
const ct = ctx.request.type;
// => "image/png"
```

request.charset

Get request charset when present, or `undefined`:

```
ctx.request.charset;
// => "utf-8"
```

request.query

Get parsed query-string, returning an empty object when no query-string is present. Note that this getter does *not* support nested parsing.

For example "color=blue&size=small":

```
{
  color: 'blue',
  size: 'small'
}
```

request.query=

Set query-string to the given object. Note that this setter does *not* support nested objects.

```
ctx.query = { next: '/login' };
```

request.fresh

Check if a request cache is "fresh", aka the contents have not changed. This method is for cache negotiation between `If-None-Match` / `ETag`, and `If-Modified-Since` and `Last-Modified`. It should be referenced after setting one or more of these response headers.

```
// freshness check requires status 20x or 304
ctx.status = 200;
ctx.set('ETag', '123');

// cache is ok
if (ctx.fresh) {
  ctx.status = 304;
```

```
  return;
}

// cache is stale
// fetch new data
ctx.body = await db.find('something');
```

request.stale

Inverse of `request.fresh`.

request.protocol

Return request protocol, "https" or "http". Supports `X-Forwarded-Proto` when `app.proxy` is **true**.

request.secure

Shorthand for `ctx.protocol == "https"` to check if a request was issued via TLS.

request.ip

Request remote address. Supports `X-Forwarded-For` when `app.proxy` is **true**.

request.ips

When `X-Forwarded-For` is present and `app.proxy` is enabled an array of these ips is returned, ordered from upstream -> downstream. When disabled an empty array is returned.

For example if the value were "client, proxy1, proxy2", you would receive the array `["client", "proxy1", "proxy2"]`.

Most of the reverse proxy(nginx) set x-forwarded-for via `proxy_add_x_forwarded_for`, which poses a certain security risk. A

malicious attacker can forge a client's ip address by forging a `X-Forwarded-For`request header. The request sent by the client has an `X-Forwarded-For` request header for 'forged'. After being forwarded by the reverse proxy, `request.ips` will be ['forged', 'client', 'proxy1', 'proxy2'].

Koa offers two options to avoid being bypassed.

If you can control the reverse proxy, you can avoid bypassing by adjusting the configuration, or use the `app.proxyIpHeader` provided by koa to avoid reading `x-forwarded-for` to get ips.

```
const app = new Koa({
  proxy: true,
  proxyIpHeader: 'X-Real-IP',
});
```

If you know exactly how many reverse proxies are in front of the server, you can avoid reading the user's forged request header by configuring `app.maxIpsCount`:

```
const app = new Koa({
  proxy: true,
  maxIpsCount: 1, // only one proxy in front of the server
});

// request.header['X-Forwarded-For'] === [ '127.0.0.1', '127.0.0.2
// ctx.ips === [ '127.0.0.2' ];
```

request.subdomains

Return subdomains as an array.

Subdomains are the dot-separated parts of the host before the main domain of the app. By default, the domain of the app is

assumed to be the last two parts of the host. This can be changed by setting `app.subdomainOffset`.

For example, if the domain is "tobi.ferrets.example.com": If `app.subdomainOffset` is not set, `ctx.subdomains` is `["ferrets", "tobi"]`. If `app.subdomainOffset` is 3, `ctx.subdomains` is `["tobi"]`.

request.is(types...)

Check if the incoming request contains the "Content-Type" header field, and it contains any of the give mime `type`s. If there is no request body, `null` is returned. If there is no content type, or the match fails `false` is returned. Otherwise, it returns the matching content-type.

```
// With Content-Type: text/html; charset=utf-8
ctx.is('html'); // => 'html'
ctx.is('text/html'); // => 'text/html'
ctx.is('text/*', 'text/html'); // => 'text/html'

// When Content-Type is application/json
ctx.is('json', 'urlencoded'); // => 'json'
ctx.is('application/json'); // => 'application/json'
ctx.is('html', 'application/*'); // => 'application/json'

ctx.is('html'); // => false
```

For example if you want to ensure that only images are sent to a given route:

```
if (ctx.is('image/*')) {
  // process
} else {
  ctx.throw(415, 'images only!');
}
```

Content Negotiation

Koa's `request` object includes helpful content negotiation utilities powered by accepts and negotiator. These utilities are:

- `request.accepts(types)`
- `request.acceptsEncodings(types)`
- `request.acceptsCharsets(charsets)`
- `request.acceptsLanguages(langs)`

If no types are supplied, **all** acceptable types are returned.

If multiple types are supplied, the best match will be returned. If no matches are found, a `false` is returned, and you should send a `406 "Not Acceptable"` response to the client.

In the case of missing accept headers where any type is acceptable, the first type will be returned. Thus, the order of types you supply is important.

request.accepts(types)

Check if the given `type(s)` is acceptable, returning the best match when true, otherwise `false`. The `type` value may be one or more mime type string such as "application/json", the extension name such as "json", or an array `["json", "html", "text/plain"]`.

```
// Accept: text/html
ctx.accepts('html');
// => "html"

// Accept: text/*, application/json
ctx.accepts('html');
// => "html"
ctx.accepts('text/html');
// => "text/html"
ctx.accepts('json', 'text');
// => "json"
ctx.accepts('application/json');
// => "application/json"
```

```
// Accept: text/*, application/json
ctx.accepts('image/png');
ctx.accepts('png');
// => false

// Accept: text/*;q=.5, application/json
ctx.accepts(['html', 'json']);
ctx.accepts('html', 'json');
// => "json"

// No Accept header
ctx.accepts('html', 'json');
// => "html"
ctx.accepts('json', 'html');
// => "json"
```

You may call `ctx.accepts()` as many times as you like, or use a switch:

```
switch (ctx.accepts('json', 'html', 'text')) {
  case 'json': break;
  case 'html': break;
  case 'text': break;
  default: ctx.throw(406, 'json, html, or text only');
}
```

request.acceptsEncodings(encodings)

Check if `encodings` are acceptable, returning the best match when true, otherwise `false`. Note that you should include `identity` as one of the encodings!

```
// Accept-Encoding: gzip
ctx.acceptsEncodings('gzip', 'deflate', 'identity');
// => "gzip"
```

```
ctx.acceptsEncodings(['gzip', 'deflate', 'identity']);
// => "gzip"
```

When no arguments are given all accepted encodings are returned as an array:

```
// Accept-Encoding: gzip, deflate
ctx.acceptsEncodings();
// => ["gzip", "deflate", "identity"]
```

Note that the `identity` encoding (which means no encoding) could be unacceptable if the client explicitly sends `identity;q=0`. Although this is an edge case, you should still handle the case where this method returns `false`.

request.acceptsCharsets(charsets)

Check if `charsets` are acceptable, returning the best match when true, otherwise `false`.

```
// Accept-Charset: utf-8, iso-8859-1;q=0.2, utf-7;q=0.5
ctx.acceptsCharsets('utf-8', 'utf-7');
// => "utf-8"

ctx.acceptsCharsets(['utf-7', 'utf-8']);
// => "utf-8"
```

When no arguments are given all accepted charsets are returned as an array:

```
// Accept-Charset: utf-8, iso-8859-1;q=0.2, utf-7;q=0.5
ctx.acceptsCharsets();
// => ["utf-8", "utf-7", "iso-8859-1"]
```

request.acceptsLanguages(langs)

Check if `langs` are acceptable, returning the best match when true, otherwise `false`.

```
// Accept-Language: en;q=0.8, es, pt
ctx.acceptsLanguages('es', 'en');
// => "es"

ctx.acceptsLanguages(['en', 'es']);
// => "es"
```

When no arguments are given all accepted languages are returned as an array:

```
// Accept-Language: en;q=0.8, es, pt
ctx.acceptsLanguages();
// => ["es", "pt", "en"]
```

request.idempotent

Check if the request is idempotent.

request.socket

Return the request socket.

request.get(field)

Return request header with case-insensitive `field`.

Response

A Koa `Response` object is an abstraction on top of node's vanilla response object, providing additional functionality that is useful for every day HTTP server development.

API

response.header

Response header object.

response.headers

Response header object. Alias as `response.header`.

response.socket

Response socket. Points to net.Socket instance as `request.socket`.

response.status

Get response status. By default, `response.status` is set to `404` unlike node's `res.statusCode` which defaults to `200`.

response.status=

Set response status via numeric code:

- 100 "continue"
- 101 "switching protocols"
- 102 "processing"
- 200 "ok"
- 201 "created"
- 202 "accepted"
- 203 "non-authoritative information"
- 204 "no content"
- 205 "reset content"
- 206 "partial content"
- 207 "multi-status"
- 208 "already reported"
- 226 "im used"
- 300 "multiple choices"
- 301 "moved permanently"
- 302 "found"
- 303 "see other"
- 304 "not modified"
- 305 "use proxy"
- 307 "temporary redirect"
- 308 "permanent redirect"
- 400 "bad request"
- 401 "unauthorized"
- 402 "payment required"
- 403 "forbidden"
- 404 "not found"
- 405 "method not allowed"
- 406 "not acceptable"
- 407 "proxy authentication required"
- 408 "request timeout"
- 409 "conflict"
- 410 "gone"
- 411 "length required"
- 412 "precondition failed"
- 413 "payload too large"
- 414 "uri too long"
- 415 "unsupported media type"
- 416 "range not satisfiable"
- 417 "expectation failed"
- 418 "I'm a teapot"
- 422 "unprocessable entity"
- 423 "locked"
- 424 "failed dependency"
- 426 "upgrade required"
- 428 "precondition required"
- 429 "too many requests"

- 431 "request header fields too large"
- 500 "internal server error"
- 501 "not implemented"
- 502 "bad gateway"
- 503 "service unavailable"
- 504 "gateway timeout"
- 505 "http version not supported"
- 506 "variant also negotiates"
- 507 "insufficient storage"
- 508 "loop detected"
- 510 "not extended"
- 511 "network authentication required"

NOTE: don't worry too much about memorizing these strings, if you have a typo an error will be thrown, displaying this list so you can make a correction.

Since `response.status` default is set to `404`, to send a response without a body and with a different status is to be done like this:

```
ctx.response.status = 200;

// Or whatever other status
ctx.response.status = 204;
```

response.message

Get response status message. By default, `response.message` is associated with `response.status`.

response.message=

Set response status message to the given value.

response.length=

Set response Content-Length to the given value.

response.length

Return response Content-Length as a number when present, or deduce from `ctx.body` when possible, or `undefined`.

response.body

Get response body.

response.body=

Set response body to one of the following:

- `string` written
- `Buffer` written
- `Stream` piped
- `Object` || `Array` json-stringified
- `null` || `undefined` no content response

If `response.status` has not been set, Koa will automatically set the status to `200` or `204` depending on `response.body`. Specifically, if `response.body` has not been set or has been set as `null` or `undefined`, Koa will automatically set `response.status` to `204`. If you really want to send no content response with other status, you should override the `204` status as the following way:

```
// This must be always set first before status, since null | undefin
// body automatically sets the status to 204
ctx.body = null;
// Now we override the 204 status with the desired one
ctx.status = 200;
```

Koa doesn't guard against everything that could be put as a response body -- a function doesn't serialise meaningfully, returning a boolean may make sense based on your application, and while an error works, it may not work as intended as some properties of an error are not enumerable. We recommend

adding middleware in your app that asserts body types per app. A sample middleware might be:

```
app.use(async (ctx, next) => {
  await next()

  ctx.assert.equal('object', typeof ctx.body, 500, 'some dev did som
})
```

String

The Content-Type is defaulted to text/html or text/plain, both with a default charset of utf-8. The Content-Length field is also set.

Buffer

The Content-Type is defaulted to application/octet-stream, and Content-Length is also set.

Stream

The Content-Type is defaulted to application/octet-stream.

Whenever a stream is set as the response body, `.onerror` is automatically added as a listener to the `error` event to catch any errors. In addition, whenever the request is closed (even prematurely), the stream is destroyed. If you do not want these two features, do not set the stream as the body directly. For example, you may not want this when setting the body as an HTTP stream in a proxy as it would destroy the underlying connection.

See: https://github.com/koajs/koa/pull/612 for more information.

Here's an example of stream error handling without automatically destroying the stream:

```
const PassThrough = require('stream').PassThrough;

app.use(async ctx => {
  ctx.body = someHTTPStream.on('error', (err) => ctx.onerror(err)).p
});
```

Object

The Content-Type is defaulted to application/json. This includes plain objects `{ foo: 'bar' }` and arrays `['foo', 'bar']`.

response.get(field)

Get a response header field value with case-insensitive `field`.

```
const etag = ctx.response.get('ETag');
```

response.has(field)

Returns `true` if the header identified by name is currently set in the outgoing headers. The header name matching is case-insensitive.

```
const rateLimited = ctx.response.has('X-RateLimit-Limit');
```

response.set(field, value)

Set response header `field` to `value`:

```
ctx.set('Cache-Control', 'no-cache');
```

response.append(field, value)

Append additional header `field` with value `val`.

```
ctx.append('Link', '<http://127.0.0.1/>');
```

response.set(fields)

Set several response header `fields` with an object:

```
ctx.set({
  'Etag': '1234',
  'Last-Modified': date
});
```

This delegates to setHeader which sets or updates headers by specified keys and doesn't reset the entire header.

response.remove(field)

Remove header `field`.

response.type

Get response `Content-Type` void of parameters such as "charset".

```
const ct = ctx.type;
// => "image/png"
```

response.type=

Set response `Content-Type` via mime string or file extension.

```
ctx.type = 'text/plain; charset=utf-8';
ctx.type = 'image/png';
ctx.type = '.png';
ctx.type = 'png';
```

Note: when appropriate a `charset` is selected for you, for example `response.type = 'html'` will default to "utf-8". If you need to overwrite `charset`, use `ctx.set('Content-Type', 'text/html')` to set response header field to value directly.

response.is(types...)

Very similar to `ctx.request.is()`. Check whether the response type is one of the supplied types. This is particularly useful for creating middleware that manipulate responses.

For example, this is a middleware that minifies all HTML responses except for streams.

```
const minify = require('html-minifier');

app.use(async (ctx, next) => {
  await next();

  if (!ctx.response.is('html')) return;

  let body = ctx.body;
  if (!body || body.pipe) return;

  if (Buffer.isBuffer(body)) body = body.toString();
  ctx.body = minify(body);
});
```

response.redirect(url, [alt])

Perform a [302] redirect to `url`.

The string "back" is special-cased to provide Referrer support, when Referrer is not present `alt` or "/" is used.

```
ctx.redirect('back');
ctx.redirect('back', '/index.html');
ctx.redirect('/login');
ctx.redirect('http://google.com');
```

To alter the default status of `302`, simply assign the status before or after this call. To alter the body, assign it after this call:

```
ctx.status = 301;
ctx.redirect('/cart');
ctx.body = 'Redirecting to shopping cart';
```

response.attachment([filename], [options])

Set `Content-Disposition` to "attachment" to signal the client to prompt for download. Optionally specify the `filename` of the download and some options.

response.headerSent

Check if a response header has already been sent. Useful for seeing if the client may be notified on error.

response.lastModified

Return the `Last-Modified` header as a `Date`, if it exists.

response.lastModified=

Set the `Last-Modified` header as an appropriate UTC string. You can either set it as a `Date` or date string.

```
ctx.response.lastModified = new Date();
```

response.etag=

Set the ETag of a response including the wrapped "s. Note that there is no corresponding `response.etag` getter.

```
ctx.response.etag = crypto.createHash('md5').update(ctx.body).digest
```

response.vary(field)

Vary on `field`.

response.flushHeaders()

Flush any set headers, and begin the body.

Sponsor

Links

Community links to discover third-party middleware for Koa, full runnable examples, thorough guides and more! If you have questions join us in IRC!

- GitHub repository
- Examples
- Middleware
- Wiki
- Mailing list
- Guide
- FAQ
- **#koajs** on freenode