

Open-Source Report

Proof of knowing your stuff in CSE312

Guidelines

Provided below is a template you must use to write your reports for your project.

Here are some things to note when working on your report, specifically about the **General Information & Licensing** section for each technology.

- **Code Repository:** Please link the code and not the documentation. If you'd like to refer to the documentation in the **Magic** section, you're more than welcome to, but we need to see the code you're referring to as well.
- **License Type:** Three letter acronym is fine.
- **License Description:** No need for the entire license here, just what separates it from the rest.
- **License Restrictions:** What can you *not* do as a result of using this technology in your project? Some licenses prevent you from using the project for commercial use, for example.

Also, feel free to extend the cell of any section if you feel you need more room.

If there's anything we can clarify, please don't hesitate to reach out! You can reach us using the methods outlined on the course website or see us during our office hours.

Koa.js

General Information & Licensing

Code Repository	https://github.com/gjjin/koajs/koa/
License Type	MIT license
License Description	<p>(The MIT License)</p> <p>Copyright (c) 2019 Koa contributors</p> <p>Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the 'Software'), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:</p> <p>The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.</p> <p>THE SOFTWARE IS PROVIDED 'AS IS', WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND</p>

built-in module) by [http.createServer](#) and [pass a callback function](#) as a parameter to create the HTTP server.

```
// /lib/application.js#L98 of Koa.JS
listen (...args) {
  debug('listen')
  const server = http.createServer(this.callback())
  return server.listen(...args)
}

// /lib/application.js#L156 of Koa.JS
callback () {
  const fn = this.compose(this.middleware)
  if (!this.listenerCount('error')) this.on('error', this.onerror)
  const handleRequest = (req, res) => {
    const ctx = this.createContext(req, res)
    return this.handleRequest(ctx, fn)
  }
  return handleRequest
}
```

Then Koa will let the node built-in module, `http`, handle the TCP socket([/lib/application.js#L79 of Koa.JS](#)), and parse the header. And everytime a new request comes in, the built-in `http` library will call the callback function that was passed to create the `http` server.

Built-in Http Server and TCP Server

In the built-in Http Server, it will [create a TCP server](#)(from built-in `net` module), make the TCP server listen to the specified port.

```
// /main/lib/\_http\_server.js#L484 of nodejs
function Server(options, requestListener) {
  if (!(this instanceof Server)) return new Server(options, requestListener);

  if (typeof options === 'function') {
    requestListener = options;
    options = {};
  } else if (options == null || typeof options === 'object') {
    options = { ...options };
  } else {
    throw new ERR_INVALID_ARG_TYPE('options', 'object', options);
  }

  storeHTTPOptions.call(this, options);
  net.Server.call(
    this,
    { allowHalfOpen: true, noDelay: options.noDelay,
      keepAlive: options.keepAlive,
      keepAliveInitialDelay: options.keepAliveInitialDelay });

  if (requestListener) {
    this.on('request', requestListener);
  }
}
```

```

}

// Similar option to this. Too lazy to write my own docs.
// http://www.squid-cache.org/Doc/config/half_closed_clients/
//
https://wiki.squid-cache.org/SquidFaq/InnerWorkings#What_is_a_half-closed
_filedescriptor.3F
this.httpAllowHalfOpen = false;

this.on('connection', connectionListener);

this.timeout = 0;
this.maxHeadersCount = null;
this.maxRequestsPerSocket = 0;
setupConnectionsTracking(this);
this[kUniqueHeaders] = parseUniqueHeadersOption(options.uniqueHeaders);
}

```

Whenever a new connection establishes, the TCP server will fire the connection event (highlighted in yellow above), and the Http Server will call a function called `connectionListener` then calls the `connectionListenerInternal` function, then uses a built-in parser library to parse the HTTP data comes from the TCP socket.

```

//
https://github.com/nodejs/node/blob/7d80ca36833905d315560fc84f7245a3
684d5189/lib/_http_server.js#L607
function connectionListener(socket) {
  defaultTriggerAsyncIdScope(
    getOrSetAsyncId(socket), connectionListenerInternal, this, socket
  );
}

function connectionListenerInternal(server, socket) {
  debug('SERVER new http connection');

  // Ensure that the server property of the socket is correctly set.
  // See https://github.com/nodejs/node/issues/13435
  socket.server = server;

  // If the user has added a listener to the server,
  // request, or response, then it's their responsibility.
  // otherwise, destroy on timeout by default
  if (server.timeout && typeof socket.setTimeout === 'function')
    socket.setTimeout(server.timeout);
  socket.on('timeout', socketOnTimeout);

  const parser = parsers.alloc();

  const lenient = server.insecureHTTPParser === undefined ?
    isLenient() : server.insecureHTTPParser;

```

Here the code initializes the HTTP Parser, and passes the TCP socket object into the parser.

```

// TODO(addaleax): This doesn't play well with the
// `async_hooks.currentResource()` proposal, see
// https://github.com/nodejs/node/pull/21313
parser.initialize(
  HTTPParser.REQUEST,
  new HTTPServerAsyncResource('HTTPINCOMINGMESSAGE', socket),
  server.maxHeaderSize || 0,
  lenient ? kLenientAll : kLenientNone,
  server[kConnections],
);
parser.socket = socket;
socket.parser = parser;

// Propagate headers limit from server instance to parser
if (typeof server.maxHeadersCount === 'number') {
  parser.maxHeaderPairs = server.maxHeadersCount << 1;
}

const state = {
  onData: null,
  onEnd: null,
  onClose: null,
  onDrain: null,
  outgoing: [],
  incoming: [],
  // `outgoingData` is an approximate amount of bytes queued
  // through all
  // inactive responses. If more data than the high watermark is
  // queued - we
  // need to pause TCP socket/HTTP parser, and wait until the data
  // will be
  // sent to the client.
  outgoingData: 0,
  requestsCount: 0,
  keepAliveTimeoutSet: false
};

```

And here, the `data` event of the TCP socket is listened by the function `state.onData` which is a re-binded function of `socketOnData`

```

state.onData = socketOnData.bind(undefined,
  server, socket, parser, state);
state.onEnd = socketOnEnd.bind(undefined,
  server, socket, parser, state);
state.onClose = socketOnClose.bind(undefined,
  socket, state);
state.onDrain = socketOnDrain.bind(undefined,
  socket, state);

socket.on('data', state.onData);
socket.on('error', socketOnError);
socket.on('end', state.onEnd);
socket.on('close', state.onClose);
socket.on('drain', state.onDrain);
parser.onIncoming = parserOnIncoming.bind(undefined,
  server, socket, state);

```

```
}
```

```
https://github.com/nodejs/node/blob/main/lib/http\_server.js#L779  
function socketOnData(server, socket, parser, state, d) {  
  assert(!socket._paused);  
  debug('SERVER socketOnData %d', d.length);  
  
  const ret = parser.execute(d);  
  onParserExecuteCommon(server, socket, parser, state, ret, d);  
}
```

The `socketOnData` function is simple, it asserts if the socket is pause, and then pass the received data from the TCP socket to the `execute` function of the HTTP parser in order to parse the HTTP request.

And the parser is imported from the `_http_common` earlier in the code.

```
const {  
  parsers,  
  freeParser,  
  ...  
} = require('_http_common');
```

In the `_http_common.js` file, we can see that the parsers is actually assigned to instance of HTTPParser when it's allocated.

```
const parsers = new FreeList('parsers', 1000, function parsersCb() {  
  const parser = new HTTPParser();  
  
  cleanParser(parser);  
  
  parser[kOnHeaders] = parserOnHeaders;  
  parser[kOnHeadersComplete] = parserOnHeadersComplete;  
  parser[kOnBody] = parserOnBody;  
  parser[kOnMessageComplete] = parserOnMessageComplete;  
  
  return parser;  
});
```

Parse the Header

The report of how the HTTP parser works will be in the Parsing-Header Report.

Koa Context Object

And so far, the http request has been parsed.

```
https://github.com/nodejs/node/blob/main/lib/\_http\_server.js#L779  
function socketOnData(server, socket, parser, state, d) {  
  assert(!socket._paused);  
  debug('SERVER socketOnData %d', d.length);  
  
  const ret = parser.execute(d);  
  onParserExecuteCommon(server, socket, parser, state, ret, d);  
}
```

Back to the `socketOnData` in the `http_server` file, the result from the `HTTPParser` is stored in the `ret` variable, and then `onParserExecuteCommon` get called.

Then, the request object is built, and the `parserOnIncoming` get called, and it will build a nodejs `response` object for the http handling function to build http response (The process afterward will be described back in the TCP report since the parsing process is done).

At the end of `parserOnIncoming` method, which prepares the `request` and `response` object, it will fire the `handle` event at the end of this method([lib/_http_server.js#L1083](#))

Back when the nodejs built-in HTTP Server's construction function, the `request` event is listened by the `requestListener`, which is the second argument of the construction function.

```
function Server(options, requestListener) {  
  ...  
  if (requestListener) {  
    this.on('request', requestListener);  
  }  
  ...  
}
```

(https://github.com/nodejs/node/blob/main/lib/_http_server.js#L516)

Since the `HTTPServer` is created in the Koa, so we go back to the Koa's code.

```
// lib/application.js#L98 of Koa.JS  
listen (...args) {  
  debug('listen')  
  const server = http.createServer(this.callback())  
  return server.listen(...args)  
}  
  
// lib/application.js#L156 of Koa.JS  
callback () {  
  const fn = this.compose(this.middleware)  
  if (!this.listenerCount('error')) this.on('error', this.onerror)  
  const handleRequest = (req, res) => {  
    const ctx = this.createContext(req, res)  
    return this.handleRequest(ctx, fn)  
  }  
}
```

```
    return handleRequest
  }
```

In the callback function, we can see that it returns a handler function that accepts 2 parameters, `req` (request) and `res` (response), and then it will be passed to the `createServer` and become the listener of the `request` event. This handler accepts 2 parameters, `req` (request) and `res` (response), and then passes the `req` and `res` to the `createContext` function to create a Koa context object.

```
// /lib/application.js#L197
createContext (req, res) {
  /** @type {Context} */
  const context = Object.create(this.context)
  /** @type {KoaRequest} */
  const request = context.request = Object.create(this.request)
  /** @type {KoaResponse} */
  const response = context.response = Object.create(this.response)
  context.app = request.app = response.app = this
  context.req = request.req = response.req = req
  context.res = request.res = response.res = res
  request.ctx = response.ctx = context
  request.response = response
  response.request = request
  context.originalUrl = request.originalUrl = req.url
  context.state = {}
  return context
}
```

The `Object.create()` method creates a new object, using an existing object as the prototype of the newly created object.

The **Koa Context Object**, **Koa Request Object** and **Koa Response Object** are created by `Object.create` function.

Handle Response

It uses the `Context` from `context.js`, `Response` from `response.js`, `Request` from `request.js` as prototype to build new objects for each request. They extended request object and response object from built-in `http` module.

For example, our project used the `redirect` method and set return data to `body` setter.

```
// https://github.com/koajs/koa/blob/6134b3c8bfd5ad6646c23614315443f6f1689e41/lib/response.js#L267
redirect (url, alt) {
  // location
  if (url === 'back') url = this.ctx.get('Referrer') || alt || '/'
  this.set('Location', encodeURIComponent(url))

  // status
```



```

    if (!statuses.redirect[this.status]) this.status = 302

    // html
    if (this.ctx.accepts('html')) {
      url = escape(url)
      this.type = 'text/html; charset=utf-8'
      this.body = `Redirecting to <a href="${url}">${url}</a>.`
      return
    }

    // text
    this.type = 'text/plain; charset=utf-8'
    this.body = `Redirecting to ${url}.`
  }
}

```

In the redirect method, it sets `Location` Header, set the http respond code to 302, and set the `body` to a text that says it's going to redirect.

```

// https://github.com/koajs/koa/blob/6134b3c8bfd5ad6646c23614315443f6f1689c41/lib/application.js#L156 of Koa.JS
callback () {
  const fn = this.compose(this.middleware)
  if (!this.listenerCount('error')) this.on('error', this.onerror)
  const handleRequest = (req, res) => {
    const ctx = this.createContext(req, res)
    return this.handleRequest(ctx, fn)
  }
  return handleRequest
}

```

Then, after creating the context, the callback function calls the `this.handleRequest` with the koa context object and the composed middlewares.

```

//
https://github.com/koajs/koa/blob/6134b3c8bfd5ad6646c23614315443f6f1689c41/lib/application.js#L182
handleRequest (ctx, fnMiddleware) {
  const res = ctx.res
  res.statusCode = 404
  const onerror = err => ctx.onerror(err)
  const handleResponse = () => respond(ctx)
  onFinished(res, onerror)
  return fnMiddleware(ctx).then(handleResponse).catch(onerror)
}

```

Koa uses a [union middleware model](#), all middleware are written in [Node.js Promise](#). When all the middlewares (controller and routers are also treated as middlewares in Koa) are done, the promise calls `handleResponse` to build the HTTP Response.

```

//
https://github.com/koajs/koa/blob/6134b3c8bfd5ad6646c23614315443f6f1689c41/lib/application.js#L261
function respond (ctx) {
  // allow bypassing koa
  if (ctx.respond === false) return

  if (!ctx.writable) return

  const res = ctx.res
  let body = ctx.body

```

```

const code = ctx.status

// ignore body
if (statuses.empty[code]) {
  // strip headers
  ctx.body = null
  return res.end()
}

if (ctx.method === 'HEAD') {
  if (!res.headersSent && !ctx.response.has('Content-Length')) {
    const { length } = ctx.response
    if (Number.isInteger(length)) ctx.length = length
  }
  return res.end()
}

// status body
if (body == null) {
  if (ctx.response._explicitNullBody) {
    ctx.response.remove('Content-Type')
    ctx.response.remove('Transfer-Encoding')
    ctx.length = 0
    return res.end()
  }
  if (ctx.req.httpVersionMajor >= 2) {
    body = String(code)
  } else {
    body = ctx.message || String(code)
  }
  if (!res.headersSent) {
    ctx.type = 'text'
    ctx.length = Buffer.byteLength(body)
  }
  return res.end(body)
}

// responses
if (Buffer.isBuffer(body)) return res.end(body)
if (typeof body === 'string') return res.end(body)
if (body instanceof Stream) return body.pipe(res)

// body: json
body = JSON.stringify(body)
if (!res.headersSent) {
  ctx.length = Buffer.byteLength(body)
}
res.end(body)
}

```

The `res.end` function process the body, and pass it to the `end` method of the built-in Response object to send data.

```

//
https://github.com/nodejs/node/blob/7d80ca36833905d315560fc84f7245a3684d5189/lib/\_http\_outgoing.js#L972
OutgoingMessage.prototype.end = function end(chunk, encoding, callback) {
  ...
  if (chunk) {

```

```

    if (this.finished) {
      onError(this,
        new ERR_STREAM_WRITE_AFTER_END(),
        typeof callback !== 'function' ? nop : callback);
      return this;
    }

    if (this.socket) {
      this.socket.cork();
    }

    write_(this, chunk, encoding, null, true);
  } else if (this.finished) {
    ...
  }
  return this;
};

```

In the `end` function of the `ServerResponse` class (inherits from `OutgoingMessage` class), it feed the body to the socket.

Also the `status` of the Koa Response setter pass the `statusCode` to `statusCode` of the built-in Response object to change to HTTP Response Code.

```

//
https://github.com/koajs/koa/blob/6134b3c8bfd5ad6646c23614315443f6f1689c41/lib/response.js#L83
set status (code) {
  if (this.headerSent) return

  assert(Number.isInteger(code), 'status code must be a number')
  assert(code >= 100 && code <= 999, `invalid status code: ${code}`)
  this._explicitStatus = true
  this.res.statusCode = code
  if (this.req.httpVersionMajor < 2) this.res.statusMessage =
    statuses[code]
  if (this.body && statuses.empty[code]) this.body = null
}

```

