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/koajs/koa/
License Type	MIT license
License Description	• Commercial use • Modification • Distribution • Private use
License Restrictions	• Liability • Warranty

Magic ★★⁀°・°☽°⁀★。°★彡✦༄

Dispel the magic of this technology. Replace this text with some that answers the following questions for the above tech:

- How does this technology do what it does? Please explain this in detail, starting from after the TCP socket is created
- Where is the specific code that does what you use the tech for? You ***must*** provide a link to the specific file in the repository for your tech with a line number or number range.
  - If there is more than one step in the chain of calls *(hint: there will be)*, you must provide links for the entire chain of calls from your code, to the library code that actually accomplishes the task for you.
  - Example: If you use an object of type HttpRequest in your code which contains the headers of the request, you must show exactly how that object parsed the original headers from the TCP socket. This will often involve tracing through multiple libraries and you must show the entire trace through all these libraries with links to all the involved code.

*This section will likely grow beyond the page

x

(Certain events have the link to the source code in GitHub)

Koa is a HTTP framework for Node.JS. When it starts service, it internally creates a Node.JS HTTPServer (a node built-in module) by `http.createServer` and pass a callback function as a parameter to create the HTTP server.

```
// /lib/application.js#L79 of Koa.JS
listen (...args) {
 debug('listen')
 const server = http.createServer(this.callback())
 return server.listen(...args)
}

// /lib/application.js#L135 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.

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`, then uses a buil-in `parser` library to parse the HTTP data comes from the TCP socket.

```
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;

 // 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
};
 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);

  // We are consuming socket, so it won't get any actual data
  socket.on('resume', onSocketResume);
  socket.on('pause', onSocketPause);

  // Overrides to unconsume on `data`, `readable` listeners
  socket.on = generateSocketListenerWrapper('on');
  socket.addListener = generateSocketListenerWrapper('addListener');
                       socket.prependListener                        =
generateSocketListenerWrapper('prependListener');
  socket.setEncoding = socketSetEncoding;

  // We only consume the socket if it has never been consumed before.
  if (socket._handle && socket._handle.isStreamBase &&
      !socket._handle._consumed) {
    parser._consumed = true;
    socket._handle._consumed = true;
    parser.consume(socket._handle);
}
  parser[kOnExecute] =
    onParserExecute.bind(undefined,
                         server, socket, parser, state);

  parser[kOnTimeout] =
    onParserTimeout.bind(undefined,
                         server, socket);

  socket._paused = false;
}
```

After handling the TCP connection, the built-in http library uses the Parser to parse the HTTP request/response, the report of it will be in the Parsing-Header Report.