Логирование, модули debug и winston

Наша следующая тема: Логирование или, иначе говоря, откладочный вывод. Если проект маленький, то вполне достаточно *console.log* для того, чтобы что-то вывести. Однако, проект имеет свойство расти. Например, тот же *server.js* естественным образом разделяется на *server* и собственно рабочий запрос – *request*.

1. **var** http = require('http');
2.
3. **var** server = http.createServer();
4.
5. server.on('request', require('./request'));
6.
7. server.listen(1337);
8.
9. console.log("Server is running");

 Со временем появляется работа с пользователями, база данных и т.д. каждый файл, по ходу своего выполнения, может захотеть что-то вывести. Этот вывод для нас очень важен, поскольку показывает, что происходит, особенно, если что-то происходит не так.

В текущем коде везде используется *console.log* для вывода.

1. **var** url = require('url');
2.
3. module.exports = **function**(req, res) {
4. **var** urlParsed = url.parse(req.url, **true**);
5.
6. console.log("Got request", req.method, req.url);
7.
8. **if** (req.method == 'GET' && urlParsed.pathname == '/echo' && urlParsed.query.message ) {
9. **var** message = urlParsed.query.message;
10. debug("Echo: " + message);
11. res.end(message);
12. **return**;
13. }
14.
15. console.log("Unknown URL");
16.
17. res.statusCode = 404;
18. res.end('Not Found');
19. };

Это означает, что, перейдя по браузерному url, я получу однообразную кашу из всех записей, которые делает скрипт. Что из этих записей сделано каким модулем совершенно непонятно. Иногда можно примерно сообразить, но вообще неочевидно. А по мере появления новых модулей будет все менее очевидно, логи будут расти, соответственно, найти то, что нас интересует, будет все сложнее. Кроме того, нужен какой-то способ включить вывод только в определенных файлах, в тех, которые нас на данный момент интересуют, в которых могут быть ошибки, которые мы сейчас разрабатываем. Для этого используются специализированные модули. Рекордсмен по простоте – модуль *debug*. Сейчас я его поставил, и буду использовать. Для этого подключаю. При подключении нужно указать идентификатор, которым он будет метить весь вывод из данного файла. В данном случае пусть это будет *server.* Вместо *console.log - debug:*

1. **var** http = require('http');
2. **var** debug = require('debug')(server);
3.
4. **var** server = http.createServer();
5.
6. server.on('request', require('./request'));
7.
8. server.listen(1337);
9.
10. debug("Server is running");

Аналогичную операцию я произвожу и с файлом *request.js*. Только на этот раз идентификатор будет не *server,* а *server:request.* Обратите внимание на двоеточие, это нам еще будет важно.

1. **var** url = require('url');
2. **var** debug = require('debug')('server:request'); // нужно установить npm i debug
3.
4. module.exports = **function**(req, res) {
5. **var** urlParsed = url.parse(req.url, **true**);
6.
7. debug("Got request", req.method, req.url);
8.
9. **if** (req.method == 'GET' && urlParsed.pathname == '/echo' && urlParsed.query.message ) {
10. **var** message = urlParsed.query.message;
11. debug("Echo: " + message);
12. res.end(message);
13. **return**;
14. }
15.
16. debug("Unknown URL");
17.
18. res.statusCode = 404;
19. res.end('Not Found');
20. };

Итак, запускаю не из WebStorm, а через терминал. Ввожу node server.js. Вы видите, что ничего не выводит. Для того, чтобы выводило, мне нужно указать, что выводить. Для передачи этой информации нужно установить переменное окружение с названием *DEBUG*. Если у вас Windows, то вы ее можете прямо сейчас установить. И теперь идет вывод именно от сервера. Если я перейду в браузер куда-то, вывода никакого не будет. Чтобы был вывод, мне нужно указать *DEBUG server:request.* Для того, чтобы в сервере выводилось что угодно, вместо *request* указываю \*. Теперь от сервера нет вывода, а от обработчика запроса он появился. Если же мне нужно указать вывод нескольких веток исполнения, то я могу перечислить их через запятую.

Модуль *debug –* это, с одной стороны, простое и гибкое решение задачи логинга, а с другой стороны, он иногда уж слишком прост. Например, здесь у нас дебагом обозначены сообщения, важность которых совершенно различная:

1. module.exports = **function**(req, res) {
2. **var** urlParsed = url.parse(req.url, **true**);
3.
4. debug("Got request", req.method, req.url);
5.
6. **if** (req.method == 'GET' && urlParsed.pathname == '/echo' && urlParsed.query.message ) {
7. **var** message = urlParsed.query.message;
8. debug("Echo: " + message);
9. res.end(message);
10. **return**;
11. }
12.
13. debug("Unknown URL");
14.
15. res.statusCode = 404;
16. res.end('Not Found');
17. };

Скажем, иформация 4 может быть средней важности при отладке, информация 8 может быть неважной, информация 13 может быть очень важной, поскольку это ошибка, *url* не найден. При помощи *debug* создать важность каким-то образом нельзя. Кроме того, он все пишет в стандартный поток вывода, а мы можем захотеть писать в файл либо в базу данных. Если такая потребность возникла или планируете, что она возникнет, тогда имеет смысл взглянуть на более навороченный модуль для логирования, который называется *winston*. Сейчас я его поставил и заменил *require debug* на *winston,* который возвращает объект *log:*

1. **var** url = require('url');
2. **var** log = require('winston');
3.
4. module.exports = **function**(req, res) {
5. **var** urlParsed = url.parse(req.url, **true**);
6.
7. log.info("Got request", req.method, req.url);
8.
9. **if** (req.method == 'GET' && urlParsed.pathname == '/echo' && urlParsed.query.message ) {
10. **var** message = urlParsed.query.message;
11. log.debug("Echo: " + message);
12. res.end(message);
13. **return**;
14. }
15.
16. log.error("Unknown URL");
17.
18. res.statusCode = 404;
19. res.end('Not Found');
20. };

Для того, чтобы логировать, я должен вызвать соответствующий метод этого объекта. *log.info* означает, что сообщение средней важности, *log.debug* – маленькой важности и *log.error* – что это ошибка. Все ошибки считаются очень важными. Кроме того, логгер настроен так, что по умолчанию он выводит сообщения только уровня *info* и более важные. Сейчас мы это увидим. Запускаю. В браузере перехожу на соответствующий *url.* Еще раз перехожу. Посмотрите, пожалуйста, внимательно, на то, что здесь происходит. Во-первых, как и говорилось ранее, *debug* вообще не выводится. Выводится только сообщение уровня *info* и более важные, такие как *error.* Во-вторых, здесь другой формат сообщений, но его можно легко настроить. В-третьих, здесь как-то получилось, что сообщение *error* почему-то находится выше *info,* хотя, на самом деле, вызов *error* был после. Это, к сожалению, особенность WebStorm. При запуске в консоле все будет хорошо.

В модуле *debug* можно было ограничить вывод только интересующими нас модулями, указав их переменное окружение *debug.* А как делает то же самое *winston?* К сожалению, в самом *winston* такой функциональности нету. Поэтому ее придется реализовать самим. Как? Просто сделать обертку на *winston,* которая будет находиться в отдельном модуле, и добавлять интересующую нас функциональность. Назовем этот модуль *log.*

1. **var** url = require('url');
2. **var** log = require('./log')(module);
3.
4. module.exports = **function**(req, res) {
5. **var** urlParsed = url.parse(req.url, **true**);
6.
7. log.info("Got request", req.method, req.url);
8.
9. **if** (req.method == 'GET' && urlParsed.pathname == '/echo' && urlParsed.query.message ) {
10. **var** message = urlParsed.query.message;
11. log.debug("Echo: " + message);
12. res.end(message);
13. **return**;
14. }
15.
16. log.error("Unknown URL");
17.
18. res.statusCode = 404;
19. res.end('Not Found');
20. };

Он будет принимать текущий объект модуля и возвращать по сути тот же *winston,* но по разному настроенный, в зависимости от того, какой именно модуль мы ему передаем. Для каких-то будем логировать так, для других – по-другому. Конфигурацию здесь можно делать сколь угодно гибкой.

Вот пример такого модуля:

1. **var** winston = require('winston');
2.
3. module.exports = **function**(module) {
4. **return** makeLogger(module.filename);
5. };
6.
7. **function** makeLogger(path) {
8.
9.
10. **if** (path.match(/request.js$/)) {
11.
12. **var** transports = [
13.
14. **new** winston.transports.Console({
15. timestamp: **true**, // function() { return new Date().toString() }
16. colorize: **true**,
17. level: 'info'
18. }),
19.
20. **new** winston.transports.File({ filename: 'debug.log', level: 'debug' })
21. ];
22.
23. **return** **new** winston.Logger({ transports: transports });
24.
25. } **else** {
26.
27. **return** **new** winston.Logger({
28. transports: []
29. });
30.
31. }
32. }

Он экспортирует функцию, которая принимает модуль, для которого нужно делать логирование, и возвращает *winston,* настроенный соответственно его пути. В данном случае, такая настройка заключается в том, что мы смотрим, на что оканчивается путь.

1. **if** (path.match(/request.js$/)) {
2.
3. **var** transports = [
4.
5. **new** winston.transports.Console({
6. timestamp: **true**, // function() { return new Date().toString() }
7. colorize: **true**,
8. level: 'info'
9. }),
10.
11. **new** winston.transports.File({ filename: 'debug.log', level: 'debug' })
12. ];
13.
14. **return** **new** winston.Logger({ transports: transports });
15.
16. } **else** {

Если это *request.js*, то возвращаем *winston* настроенный одним способом. А если что-то другое, то другим. Для настройки *winston* здесь используется концепция транспорта. Это нечто, что умеет передавать информацию, в данном случае информацию с логов. Например, есть встроенный *winston transports.console*, который можно настраивать вот так*:*

1. **new** winston.transports.Console({
2. timestamp: **true**, // function() { return new Date().toString() }
3. colorize: **true**,
4. level: 'info'
5. }),

И использовать его только для сообщений уровня *info* или выше. Второй транспорт здесь – файл, который записывает здесь, как указано, в *debug.log,* и будет включаться для сообщений уровня *debug* или выше, то есть, фактически для всех:

1. **new** winston.transports.File({ filename: 'debug.log', level: 'debug' })

Таким образом, если путь оканчиваетсяна *request.js,* то мы возвращаем *winston,* который будет записывать в консоль *info* или выше, и файл, и вообще все.

Ну а для других путей мы будем возвращать *winston* вообще без транспорта:

1. **return** **new** winston.Logger({
2. transports: []
3. });

Соответствующие вызовы *log,* с одной стороны, не будут вызывать ошибку, а с другой стороны, такая запись никуда не пойдет. Проверяю это в браузере. Мы видим, действительно, что информация *info* попала в консоль, а также есть файл *debug.log,* в котором есть все сообщения.

Итак, мы с вами рассмотрели отладку при помощи модуля *debug,* который, с одной стороны, весьма прост, а с другой стороны, позволяет указать, какую ветку кода логировать, и вообще, для разработки его вполне хватает. Но если требуется более серьезное логирование, в том числе нескольких мест одновременно, или в базу данных, или в файл, тогда имеет смысл обратить внимание на другой модуль, который называется *winston.* Его можно гибко настраивать, в том числе, при помощи обертки.

Следующее средство отладки, которое мы сейчас изучим, называется NODE\_DEBUG. Это переменное окружение, которое используется внутри самого Node.js. Есть ряд встроенных модулей, которые, если это переменная стоит, могут показывать, что происходит внутри их. Таким образом, NODE\_DEBUG – это средство для глубокой отладки. Его используют в тех случаях, когда наши возможности по отладке исчерпаны, и нам ничего не остается, кроме как, заглянуть внутрь самой Node и посмотреть, что же там делается. Например, нам интересно, что передается по сети. Для этого нужно запустить Node, указав NODE\_DEBUG через пробелы *http net*. В этом случае мы полностью поймем, что внутри сети творится. Я еще не успел никуда зайти, а уже начала поступать информация. Модуль вывел, что мы теперь слушаем ip, то есть, все интерфейсы, и порт. А теперь перейду на 127.0.0.1. Как видите, я получаю полный отчет о том, что происходит. Если захочу, я могу залезть внутрь этих самых модулей и там еще порыться. Это соединение закрылось.

Итак, у нас есть три основных инструмента логирования, которых вполне хватит как для отладки небольшого проекта, так и для полноценного логинг решения и, наконец, чтобы заглянуть внутрь Node.js.