05

Utilizando logs na aplicação

Utilizando logs na aplicação

Um dos pontos mais críticos de uma aplicação é quando se precisa identificar aquele famoso "erro que só acontece em produção". Nessa situação o programador não tem como *debugar* o código e muitas vezes não consegue reproduzir a situação em um ambiente de testes porque não tem como saber de fato qual foi o cenário ocorrido.

Quando isso acontece é muito importante que se tenha disponível toda informação possível sobre aquela execução. E a melhor maneira de conseguir isso é através de **Logs**. Pois ele são o rastro da execução da aplicação e podem ser tão detalhados quanto o programador desejar.

Logs simples com Winston

Uma api bem simples, mas bastante eficaz para o uso de logs é a **winston**. Ela foi projetada para ser uma API de logs simples e universal com suporte para múltiplas camadas de logs. É possível que cada instância do **winston** tenha múltiplas camadas de logs e diferentes níveis em cada uma dessas camadas.

Ela também desacopla bastante a sua implementação interna de escrita dos logs das interfaces que são expostas para o programador. Isso facilita muito a vida do programador e é algo que muitas APIs de log acabam não fazendo.

Vejamos um exemplo básico passo a passo de uso do **winston** para entender melhor os conceitos. O primeiro passo é obviamente instalar a lib através do npm e em seguida carregá-la no arquivo desejado:

```
var winston = require('winston');
```

Com objeto em mãos é possível criar um nova instância de *logger* com as camadas desejadas:

```
var logger = new winston.Logger({
  transports: [
    new winston.transports.File({
      level: "info",
      filename: "logs/payfast.log",
      maxsize: 1048576,
      maxFiles: 10,
      colorize: false
    });
  ]
});
```

Cada camada é representada por um `transport` e você pode ver que a criação do objeto recebe um *array* deles. Então de fato é possível ter várias camadas. Ou vários `transports`.

Dentro de cada `transport` são definidas as características específicas de cada camada, como onde o log será escrito por exemplo: se será num arquivo, num banco de dados, no console, etc. O primeiro requisito definido no exemplo acima foi que o log deveria ser escrito em um arquivo:

```
new winston.transports.File({ ...
```

Dentro desse objeto agora são passadas as informações referentes à escrita nesse arquivo:

- `level: "info"` : indica o nível do log.
- `filename: "logs/payfast.log"` : o arquivo onde o log será escrito.
- `maxsize: 1048576` : o tamanho máximo a que pode chegar o arquivo de log, para que comece a ser rotacionado.
- `maxFiles: 10` : a quantidade máxima de arquivos que devem ser mantidos para essa camada de log.
- `colorize: false` : se o log deve usar cores ou não.

Existem ainda diversas outras possíveis configurações e você pode encontrar a relação completa no GitHub do projeto: https://github.com/winstonjs/winston (https://github.com/winstonjs/winston)

Para finalmente escrever o log agora basta utilizar o objeto que contem a camada criada e invocar um dos métodos de escrita:

```
logger.log('Log utilizado nível que foi configurado na camada: info.');
logger.log('info', 'Log forçando o nível info via parâmetro na função log().');
logger.info('Log forçando o nível info via invocação direta função info()');
```

Existem opções para invocação direta dos níveis de logs através de funções que possuem os mesmos nomes, ou através da função `log()` que recebe o nível de log como parâmetro. Quando esta informação não é passada, ela usa o nível que já estiver definido na configuração daquela camada.

Logando as requisições com Morgan

Claro que a ideia principal do uso dos logs é que eles possam ficar localizados nas partes mais críticas dos sistemas para que se tenha as informações detalhadas de cada execução. Uma forma de fazer isso, seria escrevendo uma camada do **winston** para cada rota criada no sistema.

Mas essa opção obviamente seria mais trabalhosa que o necessário e teríamos que espalhar código repetido e difícil de manter por todo o projeto. Uma maneira mais elegante de fazer essa implementação seria interceptando todas as requisições em um ponto único do código.

Como o **express** é quem coordena as requisições, o seu arquivo de configurações `custom-express.js` parece ser um forte candidato para receber este código. E de fato, já existe uma lib cujo objetivo é exatamente esse: o **morgan**, que é um *middleware* escritor de logs HTTP para Node.js.

Após instalado na aplicação via npm, o **morgan** precisa ser definido como um novo *middleware* do express. Essa implementação deve ser feita no `custom-express.js` . O primeiro passo é fazer o `require` da lib:

```
...
var morgan = require('morgan');
var logger = require('../persistencia/logger.js');
...
```

Um detalhe importante é que **morgan** é basicamente um *middleware* mesmo, ou seja, a especialidade dele é saber se integrar corretamente com o **express**. Mas ele não é um especialista em escrever os logs em si. Por esse motivo é que foi

carregado também o arquivo `persistencia/logger.js` , que contém as implementação do **winston**. A ideia então é aproveitar aquilo que o **winston** faz de melhor para que seja aproveitado pelo **morgan**. Um belo exemplo de reuso de código.

Em seguida, deve ser feita de fato a adição do **morgan** como um novo *middleware* do express:

```
...
module.exports = function() {
  var app = express();

  app.use(morgan("common", {
    stream: {
      write: function(message){
        logger.info(message)
      }
    }
  }));
...
```

Essa adição é feita invocando a função `use` a partir do objeto do express e dentro dessa invocação é passada como parâmetro a chama ao objeto do **morgan**: `morgan()` . Essa chamada também recebe alguns parâmetros: o primeiro deles indica o formato do log e o segundo é um json com configurações mais específicas.

O formato do log pode ser um formato já pré-definido ou algum que tenha sido criado. Nesse caso foi utilizado o "commom". Este é um formato pré-definido que significa a saída de log padrão no nível comum segundo o Padrão da Apache. Em outras palavras, um log com os seguintes itens para cada requisição:

```
:remote-addr - :remote-user [:date[clf]] ":method :url HTTP/:http-version" :status :res[content-
```

No segundo parâmetro, o json, foi definido que o log será escrito via **stream** e que o código que deve ser executado no momento que o **morgan** interceptar o request para fazer uma escrita deve ser a função que foi definida no atributo `write` :

```
write: function(message){
  logger.info(message)
}
```

Essa função usa objeto `logger` para escrever uma mensagem. Esse objeto é o qe foi definido com o **winston** e carregado para este arquivo no início dessa seção. A `message` que chega como parâmetro é passada pelo próprio **express** e seu formato foi definido pelo atributo "commons" conforme já demonstrado.