Visitas

Project Overview

Visitas is a **npm** module for log analysis with integrated logging capabilities.

Visitas can be used to analyze existing **CLF** logs (Apache's default format), to analyze the logs generated by other module (for example, coexisiting with Morgan) or to both log and analyze.

Development Journal

First thoughts

Originally, **Visitas** was going to be more focused on logging rather than in analysis.

The firsts prototypes of the module processed the `request` object supplied by **Express** extracting the information that was going to be stored; however... Why should I reinvent the wheel?

Morgan is the de-facto logging standard; even being automatically included in the **Express** project generation. This is easily understandable taking into account that it originally was part of the **Express** core.

After analyzing the source code (which is just one file, and quite understandable), I decided to use **Morgan** for the log retrieval (as its functions are also exposed from the module). Up to this point the idea was:

1. Recieve request
2. Obtain log information by **Morgan**
3. Store the log in a **Mongo** database

However, I realized two possible drawbacks. First, the possible performance penalty and lack of commodity of having a **Mongo** database running all the time, which would be receiving logs even several times per seconds. Second, this approach would lock the user with this module.

The solution was to offer an optional logging functioanlity which would create **CLF logs**. Then, when the user want to analyse them, the logs will be processed and stored into a **Mongo** database that will be queried by **Visitas**.

The project then was divided in three phases: module design, logging, analysis.

Module design

Designing the architecture and organization of the module problably was the most challenging task of the project. First, I studied how other modules work, specially **Morgan**, **parseClf**, **platform** and **Faker** (mainly because they were all used in **Visitas**), in order to follow stablished conventions and observe how they export its functionality.

Then, I had to design and implement the basic skeleton, deciding the configuration options that will be offered and the basic flow.

Logging

As commented, the logging functionality should be optional so users can choose their solution. In this way Visitas could also be usen as a off-line log analysis tool (with existing logs).

The idea behind the logging capability is to easily configure and register **Morgan**, saving user's effort.

Analysis

The analysis functionality is splitted between back-end and front-end.

Backend

After thinking about it, I decided the best thing would be to build an API that would offer several endpoints that the front-end can request.

Front-end

For the front-end, I choose to use the **Angular** framework for its simplicity to interact with an API and manipulate the data. In order to employ too much time in visual aspects, I used **Bootstrap**, and **Highcharts** for the graphic charts.

Implementation

Backend

visitas.js

Main file, in charge of loading the options, initiate the `Logger` (if required) and the `Dashboard`.

logger.js

Logging functionality, it:

1. Read the options
2. Create a write stream

3. Initiate `Morgan`

Main function:

```javascript
var getLogStream = function(logsDirectory, logsPrefix, rotateLogs) {

    // Rotate or not
    if (typeof rotateLogs === 'undefined')
        rotateLogs = DEFAULT_ROTATION;

    // If we want to rotate logs
    if (rotateLogs) {
        return FileStreamRotator.getStream({
            date_format: 'YYYYMMDD',
            filename: logsDirectory + '/' + logsPrefix + '-%DATE%.log',
            frequency: 'daily',
            verbose: false
        });
    }

    // Otherwise, create a simple write stream
    return fs.createWriteStream(logsDirectory + '/' + logsPrefix + '.log', {fla
};
```

dashboard.js

`Dashboard` main file. It initiates the the `API` and returns a request handler, which:

1. Parses the url
2. If the url is an `API` endpoint, execute it
3. If not, serve a static file

dashboard-api.js

`API` for the `Dashboard`. It returns a function as the following:

```javascript
function (endpoint, response) {

    switch (endpoint) {

        // Process the logs
        case 'api/process':
            processLogs(response);
            return true;

        // Logs
        case 'api/logs':
            getLogs(response);
            return true;

        ...

        default:
            return false;

}
```

log-processor.js

Component taking care of processing the logs, as following:

1. Find all the log files within the log directory
2. Read the file and split each line into logs
3. Parse the logs from strings to objects
4. Store the logs

Two modules are used in the parsing process:

- `parseClf` , to parse the logs
- `platform` , to extract information from the `user-agent` HTTP field

log-repository.js

Repository to manipulate the Mongo database.

generate-logs.js

Log generator for testing purposes. It can be executed as following:

```
node generate-logs.js [number of logs to generate] [directory to store the log
```

It uses the `Faker` module to generate fake IPs and user agents.

Front-end

The `Angular` framework is used in combination with `Bootstrap`.

Angular routing

Each route has its own controller, and the routing looks as following:

```javascript
app.config(['$routeProvider', function ($routeProvider) {
    $routeProvider
        .when('/', {
            templateUrl: 'partials/home.html',
            controller: 'DashboardCtrl'
        })
        .when('/visits', {
            templateUrl: 'partials/visits-overview.html',
            controller: 'VisitsCtrl'
        })
        .when('/os', {
            templateUrl: 'partials/technology-os.html',
            controller: 'OsCtrl'
        })
        .when('/browser', {
            templateUrl: 'partials/technology-browser.html',
            controller: 'BrowserCtrl'
        }).when('/resources', {
            templateUrl: 'partials/resources.html',
            controller: 'ResourcesCtrl'
        })
        .otherwise({
            redirectTo: '/'
        });
}]);
```

Angular controllers

The controllers are quite simple, and all look similar to the following simplified version:

JavaScript

```javascript
.controller('BrowserCtrl', ['$scope', '$http', function ($scope, $http) {
    $scope.chartConfig = ... default chart config ...
    var init = function () {
        $http
            .get('api/logs/browser')
            .then(function (response) {
                    ... update chart ...
                });
    };
    init();

}])
```

Bootstrap template

The template file is `index.html`. The application uses Angular partials, which are stored in the `/partials` directory.