Protector Stats API Configuration Guide

Forcepoint

Protector Stats API

Description
The provided code is a Node.js application that utilizes various modules to create an API server for retrieving server statistics, processing and logging server logs and exporting logs in CSB format. This guide breaks down the code to understand its functionality.

Required Modules
The code requires the following modules, which should be installed beforehand:

`express`: A web application framework for Node.js that simplifies the process of building APIs.

`child_process`: A built-in Node.js module that allows executing shell commands.

`prettier`: A module for formatting JSON responses.

`http`: A built-in Node.js module for creating HTTP servers.

`fs`: A built-in Node.js module for working with the file system.

`csv-writer`: A module for creating CSV files.

Server Setup
The code initializes an Express application and creates an HTTP server using **http.createServer()**.

```javascript
const express = require('express');
const { exec } = require('child_process');
const prettier = require('prettier');
const http = require('http');
const fs = require('fs');
const csv = require('csv-writer').createObjectCsvWriter;
const app = express();
const server = http.createServer(app);
```

API Endpoints
The code defines several API endpoints using the **app.get()** method from Express. The endpoints are as follows:

1. "**/server-stats**"**:** This endpoint retrieves ICAP statistics by executing the command **info stats**. It processes the command output, extracts relevant statistics and sends a formatted JSON response to the client. Additionally, it appends the latest log entry to a log file.
2. "**/server-data**"**:** This endpoint returns the current **timeSeriesData** array, which contains ICAP logs in JSON format.
3. "**/server-csv**"**:** This endpoint prepares a CSV writer and transforms **timeSeriesData** by separating the text and number in the **message** field. It then writes the transformed data to a CSV file named **logs.csv** and sends the file as a download to the client.

```javascript
app.get('/server-stats', (req, res) => {
// Code for retrieving server stats and generating JSON response
});
app.get('/server-data', (req, res) => {
// Code for returning timeSeriesData as JSON response
});
app.get('/server-csv', (req, res) => {
// Code for preparing CSV writer, transforming data, and writing to CSV file
});
```

Command Execution and Output Processing
The code uses the **exec()** function from the **child_process** module to execute the shell command **info stats**. It captures the command output, processes it and extracts relevant statistics and logs.

```javascript
exec('info stats', (error, stdout, stderr) => {
// Code for handling command execution, output processing, and log appending
});
```

Log Processing and Storage
The code defines a helper function called **processOutput()** to process the command output and extract statistics. It splits the output into lines, filters the lines containing the word **icap**, generates JSON logs for each filtered line and appends the logs to the **timeSeriesData** array.

```javascript
function processOutput(output) {
// Code for splitting output, filtering lines, generating logs, and storing them in timeSeriesData
}
```

Response Formatting and Sending
The code uses the **prettier** module to format the JSON response before sending it to the client. It converts the response object to a JSON string and formats it using prettier's **format()** function with the JSON parser.

```javascript
const formattedResponse = prettier.format(JSON.stringify(response), { parser: 'json' });
```

The formatted response is then sent as a JSON response using **res.type().send()**.

```javascript
res.type('application/json').send(formattedResponse);
```

Log File Appending
After sending the JSON response, the code appends the latest log entry to a log file named **logs.txt**. It retrieves the latest log entry from **timeSeriesData** and formats it as a log text. The log text is then appended to the file using the **fs.appendFile()** function.

```javascript
const logEntry = timeSeriesData[timeSeriesData.length - 1];
const logText = `[${logEntry.time}] ${logEntry.message}\n`;
fs.appendFile('logs.txt', logText, (err) => {
// Code for handling log appending errors
});
```

CSV File Creation and Download

For the **/server-csv** endpoint, the code prepares a CSV writer using the **csv-writer** module. It defines the file path and the CSV header, which includes various column titles.

```javascript
const csvWriter = csv({
path: 'logs.csv',
header: [
{ id: 'time', title: 'Time' },
{ id: 'messageText', title: 'Message (Text)' },
{ id: 'messageNumber', title: 'Message (Number)' },
{ id: 'host', title: 'Host' },
{ id: 'source', title: 'Source' },
{ id: 'sourcetype', title: 'Sourcetype' },
{ id: 'index', title: 'Index' },
],
});
```

The code then transforms the **timeSeriesData** array by extracting the text and number from the **message** field for each log entry. It creates a new array of objects with additional fields **messageText** and **messageNumber**.

Finally, the transformed data is written to the CSV file using the CSV writer's **writeRecords()** function. Upon successful creation of the CSV file, it is sent as a download to the client using **res.download()**.

```javascript
csvWriter
.writeRecords(transformedData)
.then(() => {
// Code for successful CSV creation and file download
})
.catch((err) => {
// Code for handling CSV creation errors
});
```

Server Startup

The code listens for incoming connections on the specified port (3000) using **server.listen()**.

```javascript
const port = 3000;
server.listen(port, () => {
console.log(`API server running on port ${port}`);
});
```

Conclusion

The provided code uses the Node.js application that creates an API server for retrieving ICAP statistics and exporting logs in CSV format. It utilizes various modules such as **express**, **child_process**, **prettier**, **http**, **fs** and **csv-writer** to achieve these functionalities. The code defines API endpoints for different operations and includes helper functions for processing command output, formatting, responses, and handling file operations.

forcepoint.com/contact

About Forcepoint

Forcepoint simplifies security for global businesses and governments. Forcepoint's all-in-one, truly cloud-native platform makes it easy to adopt Zero Trust and prevent the theft or loss of sensitive data and intellectual property no matter where people are working. Based in Austin, Texas, Forcepoint creates safe, trusted environments for customers and their employees in more than 150 countries. Engage with Forcepoint on www.forcepoint.com, Twitter and LinkedIn.

[Protector Stats API Configuration Guide] 13Nov2023