07/24/2024

Sisense Dashboard Monitors

Overview

The Sisense Dashboards project provides a backend solution for managing dashboard links and refresh intervals via a Node.js and Express application. It uses a YAML file (dashboards.yaml) to store configurations, which include sections, each containing a refresh interval and a list of links. The project includes endpoints to get, add, and delete links for each section.

Features

- **Serve Static Files**: Serves HTML files from the public directory.
- **Manage YAML Configurations**: Load, modify, and save configurations stored in a dashboards.yaml file.
- **API Endpoints**: Provide endpoints to get, add, and delete links in the YAML file.

File Structure

```
Unset
SisenseDashboards/
├── .vs/
├── SisenseDashboards/
│   ├── bin/
│   ├── node_modules/
│   ├── obj/
│   ├── public/
│   │   ├── javascripts/
│   │   │   └── settings.js
│   │   ├── stylesheets/
│   │   ├── idle_report.html
│   │   ├── index.html
│   │   ├── mission_coordination.html
│   │   ├── mission_summary.html
│   │   ├── pilot_sourcing.html
│   │   ├── gaqc.html
│   │   └── settings.html
│   ├── routes/
│   ├── views/
│   ├── app.js
│   ├── dashboards.yaml
│   ├── package-lock.json
│   ├── package.json
│   └── README.doc
└── SisenseDashboards.sln
```

Getting Started

Note: Installation is necessary only if one is wanting to modify and rebuild the source code itself. If one is only wanting to run the program, see Running the Application below for further details. To open the project in Visual Studio open the SisenseDashboards.sln located in the project directory above.

Prerequisites

- Node.js (v14 or higher recommended)
- Microsoft Visual Studio
- npm (v6 or higher recommended)

Installation

1) Navigate to the root folder for the project in either the Windows/Linux terminal, or the terminal in Visual Studio (Ctrl + `).

```
Unset
cd …path_to_project…/SisenseDashboards/SisenseDashboards
```

2) Install the npm dependencies.

```
Unset
npm install
```

app.js Overview

Sets up an Express server to manage and serve a YAML-based dashboard configuration file for a web application. It includes endpoints to retrieve, add, and delete links for different dashboard sections, ensuring persistent storage and updates to the configuration file.

Importing/Defining Modules

```
JavaScript
const express = require('express');
```

```
const path = require('path');
const fs = require('fs');
const yaml = require('yaml');
const logger = require('morgan');
const cookieParser = require('cookie-parser');
const bodyParser = require('body-parser');
const os = require('os');

// Initialize Express app
const app = express();
console.log("app: " + app);
```

Sets up the application by importing necessary modules and initializing the Express app.

Writable Directory

JavaScript

```
const originalFilePath = path.join(__dirname, 'dashboards.yaml');
const writableDir = path.join(os.homedir(), 'SisenseDashboards');
const writableFilePath = path.join(writableDir, 'dashboards.yaml');

console.log(originalFilePath);

console.log(writableDir);

if (!fs.existsSync(writableDir)) {
    fs.mkdirSync(writableDir);
}

if (!fs.existsSync(writableFilePath)) {
    fs.copyFileSync(originalFilePath, writableFilePath);
}

console.log(writableFilePath);
```

Ensures the writable directory and YAML configuration file exist, creating or copying them if necessary.

Middleware

JavaScript

```
// Middleware setup
app.use(logger('dev'));
app.use(bodyParser.json());
```

```
app.use(bodyParser.urlencoded({ extended: false }));
app.use(cookieParser());
app.use(express.static(path.join(__dirname, 'public')));


// Error handling
app.use((req, res, next) => {
    const err = new Error(`The requested URL ${req.originalUrl} was not found on this server.`);
    err.status = 404;
    next(err);
});

app.use((err, req, res, next) => {
    res.status(err.status || 500);
    res.json({
        message: err.message,
        error: err
    });
});
```

Sets up middleware for logging, parsing request bodies and cookies, serving static files, and handling errors.

API Endpoints

JavaScript

```
// Endpoint to get links and refresh time
app.get('/data/:section', (req, res) => {
    try {
        const section = req.params.section;
        const data = loadYaml();
        res.json(data[section] || { refresh: 60000, links: [] });
        console.log("Back-End: Links received from loadYaml()!");
    } catch (error) {
        console.error("Back-End: Error processing GET request:", error);
        res.sendStatus(500);
    }
});

// Endpoint to add a link
app.post('/data/:section', (req, res) => {
    try {
        const section = req.params.section;
        const url = req.body.url;

        console.log(`Back-End: Received request to add link to section: ${section}, url: ${url}`);

        const data = loadYaml();
```

```
        if (!data[section]) {
            data[section] = { refresh: 60000, links: [] };
        }

        if (url) {
            data[section].links.push({ url });
        }

        saveYaml(data);
        console.log("Link successfully added!");
        res.sendStatus(200);
    } catch (error) {
        console.error("Back-End: Error processing POST request:", error);
        res.sendStatus(500);
    }
});

// Endpoint to delete a link
app.delete('/data/:section', (req, res) => {
    try {
        const section = req.params.section;
        const index = req.body.index;

        console.log(`Back-End: Received request to delete link from section: ${section}, index: ${index}`);

        const data = loadYaml();
        console.log("loadYaml() called!");

        if (!data[section]) {
            data[section] = { refresh: 60000, links: [] };
        }

        if (index >= 0) {
            data[section].links.splice(index, 1);
        }

        saveYaml(data);
        console.log("Link successfully deleted!");
        res.sendStatus(200);
    } catch (error) {
        console.error("Back-End: Error processing DELETE request:", error);
        res.sendStatus(500);
    }
});
```

The endpoints in this Express server provide functionality for managing dashboard links in a YAML configuration file.

- **GET /data/:section endpoint:** retrieves the links and refresh time for a specified dashboard section, returning default values if the section does not exist.

- **POST /data/:section endpoint:** adds a new link to the specified section, initializing the section if needed and updating the YAML file accordingly.
- **DELETE /data/:section endpoint:** removes a link at a given index from the specified section, ensuring the section exists before making the modification.

loadYaml() Function

```
JavaScript
// Load YAML file
function loadYaml() {
    try {
        console.log("Back-End: loadYaml() called!");
        console.log("Does fs.existsSync(" + writableFilePath + ")? " + fs.existsSync(writableFilePath));
        if (!fs.existsSync(writableFilePath)) {
            return {};
        }
        const file = fs.readFileSync(writableFilePath, 'utf8');
        console.log("Sending yaml data!");
        return yaml.parse(file);
    } catch (error) {
        console.error("Back-End: Error loading YAML file:", error);
        return {};
    }
}
```

The loadYaml function reads and parses a YAML configuration file. It first checks if the file exists at the specified path (writableFilePath). If the file is missing, it returns an empty object. If the file is present, it reads the file's contents as a UTF-8 string, parses it into a JavaScript object using the yaml library, and returns this object.

saveYaml() Function

```
JavaScript
// Save YAML file
function saveYaml(data) {
    try {
        const yamlData = yaml.stringify(data);
        fs.writeFileSync(writableFilePath, yamlData, 'utf8');
        console.log("fs.writeFileSync() called!");
        console.log(writableFilePath + ", " + yamlData);
    } catch (error) {
        console.error("Back-End: Error saving YAML file:", error);
    }
```

```
}
```

The saveYaml function serializes a JavaScript object (data) into a YAML format and writes it to a file at the specified path (writableFilePath). It uses yaml.stringify to convert the object to a YAML string and fs.writeFileSync to save the string to the file in UTF-8 encoding.

package.json Overview

Holds various metadata relevant to the project, including dependencies, scripts, and configuration settings. Please note that package-lock.json is pretty much the same thing, with the main difference being that it allows more control over versioning. Despite this it should automatically update if package.json is modified by npm installs.

Basic Information

```JavaScript
{
  "name": "sisense-dashboards",
  "version": "0.0.0",
  "private": true,
  "description": "SisenseDashboards",
  "author": {
    "name": "Tucker Styles"
  },
  ...
}
```

- **name**: The name of the project. In this case, it's "sisense-dashboards".
- **version**: The current version of the project. It follows the semantic versioning convention. Here, it's "0.0.0".
- **private**: A boolean indicating whether the package is private. Setting this to true prevents the package from being accidentally published to the public npm registry.
- **description**: A brief description of the project.
- **author**: Creator is this project is Tucker Styles, a FlyGuys intern during the summer of 2024. If contact is needed please send email here.

Scripts

```JavaScript
   "scripts": {
   "start": "node app.js"
 },
```

- **scripts**: This section defines command-line scripts that can be run using npm.
- **start**: This script starts the application by executing the command node app.js. You can run this script by using the command npm start.

Binary Configuration

```JavaScript
 "bin": "app.js",
```

- **bin**: This field specifies the entry point of the application when it's run as a binary. Here, it points to app.js, which is the main file for starting the server.

Packaging Configuration

```JavaScript
 "pkg": {
   "scripts": "app.js",
   "assets": [
     "public/**/*",
     "dashboards.yaml"
   ]
 },
```

- **pkg**: This section is used when creating a package using pkg, which allows you to package your Node.js project into an executable.
- **scripts**: Specifies the script to run when executing the packaged application. In this case, it is app.js.
- **assets**: An array of file paths to include as assets in the package. The pattern "public/**/*" includes all files in the public directory, and "dashboards.yaml" ensures the YAML configuration file is also included.

Dependencies

```JavaScript
  "dependencies": {
  "body-parser": "^1.15.0",
  "cookie-parser": "^1.4.0",
  "debug": "^2.2.0",
  "express": "^4.19.2",
  "fs": "^0.0.1-security",
  "morgan": "^1.7.0",
  "pug": "^2.0.0-beta6",
  "serve-favicon": "^2.3.0",
  "yaml": "^2.4.5"
 },
```

- **dependencies**: This section lists all the packages that your project depends on to run. Each package is specified with its version.
- **body-parser**: Middleware for parsing incoming request bodies in a middleware before your handlers, available under the req.body property.
- **cookie-parser**: Middleware for parsing cookies attached to the client request object.
- **debug**: A small debugging utility for Node.js.
- **express**: A minimal and flexible Node.js web application framework that provides a robust set of features for web and mobile applications.
- **fs**: The file system module for interacting with the file system, though it is part of the Node.js core and does not require installation.
- **morgan**: HTTP request logger middleware for Node.js.
- **pug**: A template engine for Node.js used to render HTML.
- **serve-favicon**: Middleware for serving a favicon.
- **yaml**: A library to parse and stringify YAML.

Development Dependencies

```JavaScript
"devDependencies": { "eslint": "^8.21.0" },
```

- **`devDependencies`**: This section lists packages that are only needed for development and testing, rather than for the production version of the application.
- **`eslint`**: A tool for identifying and reporting on patterns in JavaScript, helping to maintain code quality and consistency.

ESLint Configuration

```
JavaScript
"eslintConfig": {}
```

- **`eslintConfig`**: This section can hold configuration settings for ESLint. Pretty sure it was added automatically so I don’t think it's really necessary.

dashboards.yaml Overview

The dashboards.yaml file is a configuration file written in YAML (YAML Ain't Markup Language) format. It defines various sections of your dashboard and specifies the refresh intervals and the URLs for each section. Each section corresponds to a different html page within the program. Below is an example structure of a section in the dashboards.yaml file:

Example Structure

```
Unset
section_name:
  refresh: refresh_interval_in_milliseconds
  links:
    - url: url_to_dashboard_1
    - url: url_to_dashboard_2
    . . .
```

- refresh: Specifies the refresh interval for the dashboard in milliseconds.
- links: Contains a list of URLs that are associated with the dashboard.

Actual Example

```
Unset
idle_report:
  refresh: 60000
  links:
```

```
    - url:
https://flyguys.sisensepoc.com/app/main/dashboards/665670a879fa94004123165a?embed=true&r=false&t=
true&theme=6679bff079fa940041232ac5
    - url:
https://flyguys.sisensepoc.com/app/main/dashboards/663e7dc479fa9400412309ee?embed=true&r=false&t=
true&theme=6679bff079fa940041232ac5
    - url:
https://flyguys.sisensepoc.com/app/main/dashboards/663e813f79fa940041230a09?embed=true&r=false&t=t
rue&theme=6679bff079fa940041232ac5
```

Creating Additional Sections

When making a new section, it is highly recommended to set the section name the same as the HTML the section will be used on. For example, if there is a new HTML page for RevOps named “rev_ops.html” then the section name in the dashboards.yaml should be set to just “rev_ops”. Also keep in mind that each section should have a corresponding HTML file for it, this program was designed so that each HTML page should have only one section tied to it and the inverse is true.

Editing YAML Files

If you need to make changes to the dashboards.yaml file, please be aware that editing the file within the project folder won't affect the application's behavior. The dashboards.yaml file in the project directory serves as the "default configurations" and is bundled with the executable. Therefore, any modifications to this file won't take effect when the application runs. Instead, the executable creates a writable copy of the configuration file on the first run, which it uses for subsequent operations. This writable copy can be found at the following path:

```
Unset
C:\Users\...user_account_...\SisenseDashboards
```

Dashboards HTML

The following HTML pages: idle_reports.html, mission_duration.html, mission_summary.html, pilot_sourcing.html, and qaqc.html, are designed to load and display an embedded Sisense dashboard within an iframe. Each includes a script to fetch configuration data from the backend and periodically update the iframe's source URL based on the data received. You will need to sign in to the FlyGuys Designer Sisense account to be able to view the dashboards. I recommend checking the “Remember Me” box on the sign in page to avoid having to log in after every refresh.

Basic <iframe> Implementation

```
Unset
<iframe width="100%" frameborder="0" src="" id="dashboardFrame"></iframe>
```

loadDashboard() Function

```
JavaScript
async function loadDashboard() {
    try {
        const response = await fetch('/data/section_name_here'); // Specify section here
        if (!response.ok) {
            throw new Error('Network response was not ok');
        }
        const data = await response.json();
        const links = data.links || []; // Adjust to match the structure in YAML
        const refreshTime = data.refresh || 60000; // Default to 1 minute if not provided
        let currentIndex = 0;

        function updateIframe() {
            if (links.length > 0) {
                const iframe = document.getElementById('dashboardFrame');
                iframe.src = links[currentIndex].url;
                currentIndex = (currentIndex + 1) % links.length;
            }
        }

        updateIframe();
        setInterval(updateIframe, refreshTime);
    } catch (error) {
        console.error('Failed to load dashboard data:', error);
    }
}
loadDashboard();
```

The loadDashboard() function in the HTML file is responsible for dynamically loading and displaying Sisense dashboards within an iframe.

1) First, it sends a GET request to `/data/section_name_here` using the fetch function and retrieves dashboard links and refresh intervals from the backend server.

2) Second, it checks if the response is successful, if so it parses the JSON response to get the data (The YAML has to be converted to JSON). Otherwise, if the response fails it throws an error.

3) Third, it calls the updateIframe() function and sets the iframe's source to the next link in the array, which will also cause the page to refresh. Once the index reaches the end of the array it then goes back to 0 and continues.

4) Finally, once the webpage is updated, it sets a periodic call to the updateIframe() function based on the refresh interval time defined in the corresponding section within dashboards.yaml.

Settings HTML

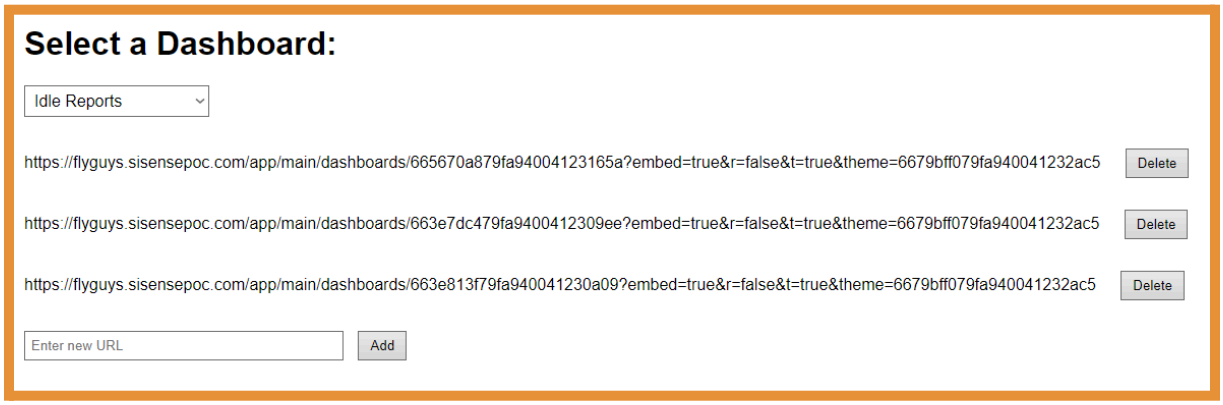
Provides a user interface for selecting a dashboard and managing its associated links. Here's a brief description of its structure and functionality:

Functionality

- **Dashboard Selection:** Users can select a dashboard from the dropdown menu. The fetchLinks() function (defined in settings.js) is called whenever a new dashboard is selected, which fetches and displays the links associated with the selected dashboard.

- **Link Management:** The dashboardLinks container is used to display the links for the selected dashboard. Each link will be displayed with options to edit or delete it (handled by JavaScript). The addLinkContainer provides an input field for users to enter a new link's URL and a button to add this link to the selected dashboard. The addLink() function (defined in settings.js) handles the addition of new links.

Preview



settings.js Overview

A JavaScript script that handles the frontend logic for dynamically displaying and managing links associated with selected dashboards in the settings.html file.

fetchLinks() Function

```javascript
function fetchLinks() {
    const section = document.getElementById('dashboardSelect').value;
    console.log(`Front-end: fetchLinsk() called for section: ${section}!`);
    fetch(`/data/${section}`)
        .then(response => response.json())
        .then(data => {
            const linksContainer = document.getElementById('dashboardLinks');
            linksContainer.innerHTML = ''; // Clear existing links
            data.links.forEach((link, index) => {
                const linkElement = document.createElement('div');
                linkElement.className = 'link-item';
                linkElement.innerHTML = `
                    <p>${link.url}</p>
                    <div class="edit-delete">
                        <button onclick="deleteLink(${index})">Delete</button>
                    </div>
                `;
                linksContainer.appendChild(linkElement);
            });
        });
}
```

The fetchLinks() function retrieves and displays a list of links for a selected dashboard section on a webpage.

1) First, it fetches the value of the currently selected option in the dropdown element with the ID dashboardSelect
2) Second, it sends a GET request to the server to fetch links associated with the selected section. Upon receiving the response, it converts it to JSON
3) Third, the function clears any existing links in the designated container element.
4) Forth, it will iterate over the fetched links, creating a div for each link that contains the link URL and a "Delete" button, which calls the deleteLink() function with the link's index when clicked
5) Finally, each div is then appended to the container, displaying the links on the webpage.

addLinks() Function

```javascript
function addLink() {
    const section = document.getElementById('dashboardSelect').value;
    const newLink = document.getElementById('newLinkInput').value;

    console.log(`Front-End: Sending request to add ${newLink} into section: ${section}`);
    if (!newLink) return;

    fetch(`/data/${section}`, {
        method: 'POST',
        headers: { 'Content-Type': 'application/json' },
        body: JSON.stringify({ url: newLink })
    }).then(response => {
        if (response.ok) {
            console.log("Front-End: addLinks() response received!");
            fetchLinks(); // Refresh the list
            console.log(``);
            document.getElementById('newLinkInput').value = ''; // Clear input
        } else {
            console.error("Front-End: Error adding link:", response.statusText);
        }
    }).catch(error => {
        console.error("Front-End: Fetch error:", error);
    });
    console.log(`Front-End: addLink() finished!`);
}
```

The addLinks() function adds a new link to a selected dashboard section and then refreshes the displayed list of links.

1) First, it retrieves the selected section from a dropdown menu and the new link URL from an input field. If the input field is empty, the function returns without doing anything.

2) Second, it sends a POST request to the server with the new link data in JSON format.

3) Finally, calling the fetchLinks() function to refresh the displayed links, and clears the input field.

deleteLinks() Function

```javascript
function deleteLink(index) {
    const section = document.getElementById('dashboardSelect').value;
    console.log(`Front-End: Sending request to remove from section: ${section}`);


    fetch(`/data/${section}`, {
        method: 'DELETE',
        headers: { 'Content-Type': 'application/json' },
        body: JSON.stringify({ index: index })
    }).then(response => {
        if (response.ok) {
            fetchLinks(); // Refresh the list
        }
    });
    console.log("Front-End: deleteLink() finished!")
}
```

The deleteLinks() function removes a link from a selected dashboard section and then refreshes the displayed list of links.

1) First, it retrieves the selected section from a dropdown menu and logs the action of removing a link from this section.
2) Second, it will send a DELETE request to the server with the index of the link to be removed, formatted as JSON.
3) Finally, if the server response is successful, the function calls fetchLinks() to refresh the displayed links.

Executables

Building/Packaging

It is highly recommended that you build the executable in the device you want to use the program on. For example, when building an executable file for the Raspberry Pi 4, a Linux based operating system, it will have a much greater chance of succeeding if building it on the Raspberry Pi 4 itself. Same thing for Windows and most likely Mac, I don't have any Mac laptops so I don't have a real opportunity to test this out. To build the open the terminal on either the Windows terminal or the Visual Studios terminal (Ctrl + `) and navigate to the root folder:

```
cd …path_to_project…/SisenseDashboards/SisenseDashboards
```

After navigating, write the following command into the terminal:

```
Unset
pkg .
```

This will result in both a Windows executable or a Linux executable in the root folder. Please remember that if you built the executable in Windows, the Linux executable will not work unless built in a Linux environment; preferably on the device you're planning to use the executable on. The vice-versa is also true, although the Windows executable should work on most Windows devices.

Creating Shortcuts

- **Windows**
  Move the executable or project folder to a preferred location. Then, create a text file, preferably on the desktop, and enter the following chunk into it. Please be sure to modify the file path to match the location where you moved the executable or project folder. When done save the text file as a .bat file.

```
Unset
@echo off
start "" ""C:\...path_to_executable\sisense-dashboards.exe""
timeout /t 5 /nobreak >nul
start http://localhost:1337
```

- **Linux**
  Create a new file on the desktop and enter the following chunk below. When finished be sure to save the file as a `.sh` file.

```
Unset
#!/bin/bash

/home/flyguys/SisenseDashboards/SisenseDashboards/sisense-dashboards-linux &
sleep 5
xdg-open http://localhost:1337
```

Next, open the Linux terminal and navigate to the directory containing the shortcut file and run the following command:

```
Unset
chmod +x /path/to/your/script.sh
```

Running the Application

When running the application from the **desktop shortcut**, it should either automatically open the terminal or prompt the user if they want to execute with the terminal depending on which operating system they are running it on. I highly recommend opening the application with the terminal as it makes fully closing the application easier. After the terminal booted up the user should see the following line:

```
Unset
Back-End: Server is running on port 1337
```

After a few seconds, the webpage **`http://localhost:1337`** should boot up automatically on the main index page. If running the application directly from the executable file, it will open the terminal but it won't open the webpage automatically. Open your browser and enter the webpage onto the address bar and it should take you to the main index page. The server can only be booted up once, **attempting to boot up the server more than once will not work.** However, once the server is booted up you can access the webpage across multiple browsers, tabs, and windows. When closing the application, closing the terminal window will essentially kill the server. If the terminal can't be found and is running in the background, search for it in the task manager and end the task from there to kill the server. This will be the most likely reason as to why the server won't run.

The Untouchables

These are files or folders that were generated by the IDE or the dependencies while constructing this program, so I am not totally sure exactly what these do. **I highly recommend avoiding touching these unless you know what you are doing!**

- bin
- node_modules
- obj
- routes
- views
- .vs