@serve.zone/coretraffic

the pure TypeScript ingress server that handles SSL termination, load balancing and reverse proxying within a docker swarm.

- readme.md for @serve.zone/coretraffic
- changelog.md for @serve.zone/coretraffic

readme.md for @serve.zone/coretraffic

route traffic within your docker setup. TypeScript ready.

Install

To install `coretraffic`, you should have Node.js already set up on your system. Assuming Node.js and npm are ready, install the package via the npm registry with the following command:

```
npm install coretraffic
```

To make the most out of `coretraffic`, ensure TypeScript is set up in your development environment, as this module is TypeScript ready and provides enhanced IntelliSense.

Usage

`coretraffic` is designed to manage and route traffic within a Docker setup, offering robust solutions to your traffic management needs with an emphasis on efficiency and reliability. Utilizing TypeScript for static typing, you get enhanced code completion and fewer runtime errors. The module is set up to handle the intricacies of proxy configuration and routing, providing a powerful foundation for any Docker-based traffic management application.

Below, we'll delve into the capabilities and features of `coretraffic`, complete with comprehensive examples to get you started.

Initializing CoreTraffic

First, you'll want to create an instance of the `CoreTraffic` class. This serves as the entry point to accessing the module's capabilities.

```
import { CoreTraffic } from 'coretraffic';
```

```
// Create an instance of CoreTraffic
const coreTrafficInstance = new CoreTraffic();
```

This initializes the `coreTrafficInstance` with default properties, ready to configure routing settings and handle proxy configurations.

Starting and Stopping CoreTraffic

Controlling the lifecycle of your `CoreTraffic` instance is key to effective traffic management. You can start and stop the instance with straightforward method calls.

Starting CoreTraffic

To begin managing traffic, start the instance. This sets up the internal network proxy and SSL redirection services, making them ready to handle incoming requests.

```
async function startCoreTraffic() {
  await coreTrafficInstance.start();
  console.log('CoreTraffic is now running.');
}

startCoreTraffic();
```

This code initializes all internal components, including `NetworkProxy` and `SslRedirect`, beginning to route traffic as configured.

Stopping CoreTraffic

When you no longer need to route traffic, shutting down the instance cleanly is important. `CoreTraffic` provides a `stop` method for this purpose.

```
async function stopCoreTraffic() {
  await coreTrafficInstance.stop();
  console.log('CoreTraffic has stopped.');
}

stopCoreTraffic();
```

This ensures all background tasks are halted, and network configurations are cleaned up.

Configuring Network Proxy

A core feature of `CoreTraffic` is its ability to configure network proxies dynamically. At its heart is the `NetworkProxy` class, a powerful tool for managing routing configurations.

Adding Default Headers

You may wish to integrate unique headers across all routed requests, possible with the `addDefaultHeaders` method. This is useful for tagging requests or managing CORS.

```
coreTrafficInstance.networkProxy.addDefaultHeaders({
  'x-powered-by': 'coretraffic',
  'custom-header': 'custom-value'
});
```

This injects custom headers into all outgoing responses managed by `coretraffic`, thereby allowing customized interaction with requests as needed.

Proxy Configuration Updates

Dynamic updates to proxy configurations can be facilitated via tasks managed by `CoretrafficTaskManager`. This feature allows adjustment of routing rules without interrupting service.

```
import { IReverseProxyConfig } from '@tsclass/network';

const configureRouting = async () => {
  const reverseProxyConfig: IReverseProxyConfig[] = [{
    // Example configuration, adjust as needed
    host: 'example.com',
    target: 'http://internal-service:3000',
  }];

  await coreTrafficInstance.taskmanager.setupRoutingTask.trigger(reverseProxyConfig);
  console.log('Updated routing configurations');
};

configureRouting();
```

In this example, a reverse proxy configuration is defined, specifying that requests to `example.com` should be directed to an internal service.

SSL Redirection

`CoreTraffic` supports SSL redirection, an essential feature for secure communications. The `SslRedirect` component listens on one port to redirect traffic to the secure version on another port.

```
// SslRedirect is initialized on port 7999 by default
console.log('SSL Redirection is active!');
```

Out-of-the-box, this listens on the configurable port and safely forwards insecure HTTP traffic to its HTTPS counterpart.

Coreflow Connector

A unique aspect of `coretraffic` is its integration capability with `Coreflow`, allowing communication between different network nodes. The `CoreflowConnector` facilitates receiving configuration updates via a socket connection.

Setting up the CoreflowConnector

```
const coreflowConnector = coreTrafficInstance.coreflowConnector;

async function connectCoreflow() {
  await coreflowConnector.start();
  console.log('Coreflow connector activated.');
}

connectCoreflow();
```

This method enables a persistent connection to a Coreflow server, allowing real-time configuration updates and management of routing policies.

Stopping the CoreflowConnector

To disconnect cleanly:

```
async function disconnectCoreflow() {
  await coreflowConnector.stop();
  console.log('Coreflow connector terminated.');
}

disconnectCoreflow();
```

This halts the connection, ensuring no dangling resources remain when shutting down your application.

Task Management

The `CoretrafficTaskManager` handles complex, buffered tasks. Flexibility and power are at your fingertips with this system, ideal for timed or queued execution needs.

Managing Tasks

Here is how you would initiate the task manager:

```
const taskManager = coreTrafficInstance.taskmanager;

// Start tasks
taskManager.start()
  .then(() => console.log('Task manager is running'))
  .catch(err => console.error('Failed to start task manager', err));
```

Stop tasks once processing is no longer required:

```
taskManager.stop()
  .then(() => console.log('Task manager stopped'))
  .catch(err => console.error('Failed to stop task manager', err));
```

Logging and Debugging

Effective logging is provided using `Smartlog`, designed to track detailed application insights and report on activity and actions within `coretraffic`.

Configuring Log Levels

`coretraffic` supports log levels which can be adjusted as per your requirements:

```
import { logger } from './coretraffic.logging.ts';

logger.log('info', 'System initialized');
logger.log('debug', 'Detailed debugging process');
logger.log('warn', 'Potential issue detected');
logger.log('error', 'An error has occurred');
```

These log entries help monitor logic flow and catch issues during development or deployment in production environments.

Test Setup

For those interested in testing, `coretraffic` uses `tapbundle` and `tstest` to ensure reliability and correctness. A sample test module is provided to demonstrate initialization and lifecycle actions.

Here’s an example of a non-CI test scenario:

```
import * as coretraffic from '../ts/index.js';
import { tap, expect } from '@push.rocks/tapbundle';

let testCoreTraffic;

tap.test('should create and handle coretraffic instances', async () => {
  testCoreTraffic = new coretraffic.CoreTraffic();
  expect(testCoreTraffic).toBeInstanceOf(coretraffic.CoreTraffic);

  await testCoreTraffic.start();
  await new Promise(resolve => setTimeout(resolve, 10000)); // Keep alive for demonstration
  await testCoreTraffic.stop();
});

tap.start();
```

This test suite validates essential functionality within development iterations, ensuring `coretraffic` performs as expected.

`coretraffic` offers a vast landscape of operations within Docker environments, handling traffic with modularity and efficiency. Whether starting simple routing tasks or integrating with complex systems like Coreflow, this module provides robust support where needed most. Embrace your traffic management challenges with the dedicated features of `coretraffic`. undefined

changelog.md for @serve.zone/coretraffic

2024-12-29 - 1.0.186 - fix(core)

Removed GitLab CI configuration and updated npm, gitzone settings

- Deleted .gitlab-ci.yml, removing GitLab CI pipeline configuration.
- Updated baseImage and gitscope in npmextra.json for new docker settings.
- Corrected and updated dependencies and package details in package.json.
- Enhanced readme with initialization, running, and task management examples.

2024-05-15 - 1.0.183 to 1.0.185 - Core Updates

Minor updates and maintenance.

- Two minor fixes to core functionality were applied.