readme.md for @serve.zone/coreflow

A comprehensive solution for managing Docker and scaling applications across servers, handling tasks from service provisioning to network traffic management.

Install

To install @serve.zone/coreflow, you can use npm with the following command:

```
npm install @serve.zone/coreflow --save
```

Given that this is a private package, make sure you have access to the required npm registry and that you are authenticated properly.

Usage

Coreflow is designed as an advanced tool for managing Docker-based applications and services, enabling efficient scaling across servers, and handling multiple aspects of service provisioning and network traffic management. Below are examples and explanations to illustrate its capabilities and how you can leverage Coreflow in your infrastructure. Note that these examples are based on TypeScript and use ESM syntax.

Prerequisites

Before you start, ensure you have Docker and Docker Swarm configured in your environment as Coreflow operates on top of these technologies. Additionally, verify that your environment variables are properly set up for accessing Coreflow's functionalities.

Setting Up Coreflow

To get started, you need to import and initialize Coreflow within your application. Here's an example of how to do this in a TypeScript module:

```
import { Coreflow } from '@serve.zone/coreflow';

// Initialize Coreflow
const coreflowInstance = new Coreflow();

// Start Coreflow
await coreflowInstance.start();

// Example: Add your logic here for handling Docker events
coreflowInstance.handleDockerEvents().then(() => {
  console.log('Docker events are being handled.');
});

// Stop Coreflow when done
await coreflowInstance.stop();
```

In the above example:

- The Coreflow instance is initialized.
- Coreflow is started, which internally initializes various managers and connectors.
- The method `handleDockerEvents` is used to handle Docker events.
- Finally, Coreflow is stopped gracefully.

Configuring Service Connections

Coreflow manages applications and services, often requiring direct interactions with other services like a database, message broker, or external API. Coreflow simplifies these connections through its configuration and service discovery layers.

```
// Assuming coreflowInstance is already started as per previous examples
const serviceConnection = coreflowInstance.createServiceConnection({
  serviceName: 'myDatabaseService',
  servicePort: 3306,
});

serviceConnection.connect().then(() => {
  console.log('Successfully connected to the service');
```

```
});
```

Scaling Your Application

Coreflow excels in scaling applications across multiple servers. This involves not just replicating services, but also ensuring they are properly networked, balanced, and monitored.

```
const scalingPolicy = {
  serviceName: 'apiService',
  replicaCount: 5, // Target number of replicas
  maxReplicaCount: 10, // Maximum number of replicas
  minReplicaCount: 2, // Minimum number of replicas
};

coreflowInstance.applyScalingPolicy(scalingPolicy).then(() => {
  console.log('Scaling policy applied successfully.');
});
```

In the above example:

- A scaling policy is defined with target, maximum, and minimum replica counts for the `apiService`.
- The `applyScalingPolicy` method of the Coreflow instance is used to apply this scaling policy.

Managing Network Traffic

One of Coreflow's key features is its ability to manage network traffic, ensuring that it is efficiently distributed among various services based on load, priority, and other custom rules.

```
import { TrafficRule } from '@serve.zone/coreflow';

const rule: TrafficRule = {
  serviceName: 'webService',
  externalPort: 80,
  internalPort: 3000,
  protocol: 'http',
};

coreflowInstance.applyTrafficRule(rule).then(() => {
```

```
  console.log('Traffic rule applied successfully.');
});
```

In the above example:

- A traffic rule is defined for the `webService`, redirecting external traffic from port 80 to the service's internal port 3000.
- The `applyTrafficRule` method is used to enforce this rule.

Continuous Deployment

Coreflow integrates continuous integration and deployment processes, allowing seamless updates and rollbacks for your services:

```
const deploymentConfig = {
  serviceName: 'userAuthService',
  image: 'myregistry.com/userauthservice:latest',
  updatePolicy: 'rolling', // or "recreate"
};

coreflowInstance.deployService(deploymentConfig).then(() => {
  console.log('Service deployed successfully.');
});
```

In the above example:

- A deployment configuration is created for the `userAuthService` using the latest image from the specified registry.
- The `deployService` method is then used to deploy the service using the specified update policy (e.g., rolling updates or recreating the service).

Observability and Monitoring

To keep track of your applications' health and performance, Coreflow provides tools for logging, monitoring, and alerting.

```
coreflowInstance.monitorService('webService').on('serviceHealthUpdate', (healthStatus) => {
  console.log(`Received health update for webService: ${healthStatus}`);
});
```

In the above example:

- The `monitorService` method is used to monitor the health status of the `webService`.
- When a health update event is received, it is logged to the console.

Detailed Example: Setting Up and Managing Coreflow

Here is a detailed example that covers various features, from setup to scaling and traffic management.

Step 1: Initialize Coreflow

```
import { Coreflow } from '@serve.zone/coreflow';

const coreflowInstance = new Coreflow();

async function initializeCoreflow() {
  await coreflowInstance.start();
  console.log('Coreflow initialized.');
  await manageServices();
}

initializeCoreflow().catch((error) => {
  console.error('Error initializing Coreflow:', error);
});
```

Step 2: Handling Docker Events

```
coreflowInstance.handleDockerEvents().then(() => {
  console.log('Docker events are being handled.');
});
```

Step 3: Configuring and Connecting to a Service

```
const serviceConnection = coreflowInstance.createServiceConnection({
  serviceName: 'databaseService',
  servicePort: 5432,
});

serviceConnection.connect().then(() => {
```

```
  console.log('Successfully connected to the database service.');
});
```

Step 4: Applying a Scaling Policy

```
const scalingPolicy = {
  serviceName: 'microserviceA',
  replicaCount: 3, // Starting with 3 replicas
  maxReplicaCount: 10, // Allowing up to 10 replicas
  minReplicaCount: 2, // Ensuring at least 2 replicas
};

coreflowInstance.applyScalingPolicy(scalingPolicy).then(() => {
  console.log('Scaling policy applied for microserviceA');
});
```

Step 5: Managing Network Traffic

```
import { TrafficRule } from '@serve.zone/coreflow';

const trafficRules: TrafficRule[] = [
  {
    serviceName: 'frontendService',
    externalPort: 80,
    internalPort: 3000,
    protocol: 'http',
  },
  {
    serviceName: 'apiService',
    externalPort: 443,
    internalPort: 4000,
    protocol: 'https',
  },
];

Promise.all(trafficRules.map((rule) => coreflowInstance.applyTrafficRule(rule))).then(() => {
  console.log('Traffic rules applied.');
});
```

Step 6: Deploying a Service

```
const deploymentConfig = {
  serviceName: 'authService',
  image: 'myregistry.com/authservice:latest',
  updatePolicy: 'rolling', // Performing rolling updates
};

coreflowInstance.deployService(deploymentConfig).then(() => {
  console.log('AuthService deployed successfully.');
});
```

Step 7: Monitoring a Service

```
coreflowInstance.monitorService('frontendService').on('serviceHealthUpdate', (healthStatus) =>
{
  console.log(`Health update for frontendService: ${healthStatus}`);
});
```

Advanced Usage: Task Scheduling and Traffic Configuration

In more complex scenarios, you might want to leverage Coreflow's ability to schedule tasks and manage traffic configurations.

Scheduling Tasks

Coreflow supports scheduling updates and other tasks using the `taskBuffer` API.

```
import { Task } from '@push.rocks/taskbuffer';

const checkinTask = new Task({
  name: 'checkin',
  buffered: true,
  taskFunction: async () => {
    console.log('Running checkin task...');
  },
});

const taskManager = coreflowInstance.taskManager;
```

```
taskManager.addAndScheduleTask(checkinTask, '0 * * * * *'); // Scheduling task to run every minute
taskManager.start().then(() => {
  console.log('Task manager started.');
});
```

Managing Traffic Routing

Coreflow can manage complex traffic routing scenarios, such as configuring reverse proxies for different services.

```
import { CoretrafficConnector } from '@serve.zone/coreflow';

// Assume coreflowInstance is already started
const coretrafficConnector = new CoretrafficConnector(coreflowInstance);

const reverseProxyConfigs = [
  {
    hostName: 'example.com',
    destinationIp: '192.168.1.100',
    destinationPort: '3000',
    privateKey: '<your-private-key>',
    publicKey: '<your-public-key>',
  },
  {
    hostName: 'api.example.com',
    destinationIp: '192.168.1.101',
    destinationPort: '4000',
    privateKey: '<your-private-key>',
    publicKey: '<your-public-key>',
  },
];

coretrafficConnector.setReverseConfigs(reverseProxyConfigs).then(() => {
  console.log('Reverse proxy configurations applied.');
});
```

Integrating with Cloudly

Coreflow is designed to integrate seamlessly with Cloudly, a configuration management and orchestration tool.

Starting the Cloudly Connector

```
const cloudlyConnector = coreflowInstance.cloudlyConnector;

cloudlyConnector.start().then(() => {
  console.log('Cloudly connector started.');
});
```

Retrieving and Applying Configurations from Cloudly

```
cloudlyConnector.getConfigFromCloudly().then((config) => {
  console.log('Received configuration from Cloudly:', config);

  coreflowInstance.clusterManager.provisionWorkloadServices(config).then(() => {
    console.log('Workload services provisioned based on Cloudly config.');
  });
});
```

Conclusion

Coreflow is a powerful and flexible tool for managing Docker-based applications, scaling services, configuring network traffic, handling continuous deployments, and ensuring observability of your infrastructure. The examples provided aim to give a comprehensive understanding of how to use Coreflow in various scenarios, ensuring it meets your DevOps and CI/CD needs.

By leveraging Coreflow's rich feature set, you can optimize your infrastructure for high availability, scalability, and efficient operation across multiple servers and environments. undefined

---

Revision #3
Created 2026-03-28 11:14:29 UTC by foss.global Team
Updated 2026-03-28 12:21:15 UTC by foss.global Team