Red Hat build of Node.js 14

Getting started with the circuit breaker add on for Red Hat build of Node.js

Using the circuit breaker add-on in your Node.js applications

Last Updated: 2021-05-27

Red Hat build of Node.js 14 Getting started with the circuit breaker add on for Red Hat build of Node.js

Using the circuit breaker add-on in your Node.js applications

Legal Notice

Abstract

The circuit breaker add-on for Red Hat build of Node.js is a fully supported implementation of the open source Opossum module that provides circuit breaker functionality. You can use the circuit breaker add-on in your Node.js applications to ensure that application failures are monitored and handled appropriately. When using the circuit breaker add-on, you can also define a fallback function to perform a specified action whenever an application failure occurs. You can also define event handlers to listen for different types of circuit breaker events.

Table of Contents

CHAPTER 1. INTRODUCTION TO THE CIRCUIT BREAKER ADD-ON FOR RED HAT BUILD OF NODE.JS

Red Hat provides a circuit breaker add-on for Red Hat build of Node.js, which is a fully supported implementation of the open source Opossum module. Opossum is a Node.js module developed by the NodeShift project team to provide circuit breaker functionality. You can use the circuit breaker add-on in your Node.js applications to ensure that application failures are monitored and handled appropriately.

NOTE

In the subsequent sections of this document, the term "circuit breaker" refers to the circuit breaker add-on for Red Hat build of Node.js.

1.1. IMPORTANCE OF HANDLING APPLICATION FAILURES IN MICROSERVICE ARCHITECTURES

The circuit breaker add-on helps to reduce the impact of failures on service architectures, where services asynchronously invoke other services.

A failure can occur in either of the following situations:

- If a service is unable to respond to requests in a timely manner because of latency issues and unusually high traffic volumes
- If a service is temporarily unavailable because of network connection issues

In these situations, other services might exhaust critical resources by attempting to contact a service that is too slow or unable to respond. These other services can then become unusable, which causes a cascading failure that affects the entire microservice architecture. By using the circuit breaker add-on in your Node.js application code, you can make your applications more fault tolerant and resilient.

1.2. CIRCUIT BREAKER DESIGN PATTERN

The circuit breaker pattern is designed to handle failures in service architectures and to prevent cascading failures across multiple systems. A circuit breaker can act as a proxy for a client function that is sending requests to a remote service endpoint that might fail. You can wrap any requests sent to the remote service endpoint in a circuit breaker object. The circuit breaker then calls the remote service and monitors its status. If the number of failures reaches a specified percentage threshold, the circuit breaker is triggered. The circuit breaker then ensures that any further requests to the remote service are automatically blocked with an error message or fallback response for a specified time period.

Based on the circuit breaker design pattern, the circuit breaker has one of the following states at any specific time:

Closed

The number of failures is below the specified percentage threshold that can trigger the circuit breaker. You can define the percentage threshold as an optional parameter that is passed to the circuit breaker in your Node.js application. The default percentage threshold is 50%. When the number of failures is low and the circuit breaker is **closed**, the client function can continue to send requests to the remote service endpoint.

Open

The number of failures has reached the specified percentage threshold that triggers the circuit breaker. For example, based on the default options, the circuit breaker moves to an **open** state if

50% of requests fail. The circuit breaker then blocks all calls to the remote service endpoint, with an error message or fallback response, for a specified timeout period. You can define the timeout period as an optional parameter that is passed to the circuit breaker in your Node.js application. The default timeout period is 30 seconds.

Half open

The specified timeout period has ended. For example, based on the default options, the timeout period ends and the circuit breaker moves to a **half-open** state after 30 seconds. The circuit breaker then attempts to send a limited number of test calls to the remote service endpoint. If the test calls succeed, the circuit breaker moves to a **closed** state, and the client function can continue to send requests to the remote service. However, if the test calls fail, the circuit breaker reverts to an **open** state, and it continues to block all calls to the remote service for the specified timeout period.

CHAPTER 2. INSTALLING THE CIRCUIT BREAKER ADD-ON FROM THE RED HAT CUSTOMER REGISTRY

The circuit breaker add-on for Red Hat build of Node.js is a fully supported implementation of the open source Opossum module. Red Hat provides the circuit breaker add-on as the **@redhat/opossum** module. You can download and install the **@redhat/opossum** module from the Red Hat customer registry.

Procedure

1. On the command line, access the root directory of your Node.js application.
2. To specify the download paths for installing Node.js modules, complete the following steps:
   a. In the root directory of your application, create a file named **.npmrc**.
   b. To specify the path to the Red Hat customer registry, enter the following line in the **.npmrc** file:

```
@redhat:registry=https://npm.registry.redhat.com
```

   c. To specify the path to the npm registry, enter the following line in the **.npmrc** file:

```
registry=https://registry.npmjs.org
```

   d. Save the **.npmrc** file.
3. To download and install the **@redhat/opossum** module, enter the following command:

```
$ npm install @redhat/opossum
```

Verification

1. If a **node_modules** directory already exists in your project, the **@redhat/opossum** module is automatically installed in this directory.
2. If a **node_modules** directory does not exist in your project, a **node_modules** subdirectory is automatically created in the root directory of your Node.js application, and the **@redhat/opossum** module is automatically installed in this subdirectory.

Additional resources

- The Red Hat customer registry is located at https://npm.registry.redhat.com.

CHAPTER 3. EXAMPLE APPLICATION USING THE CIRCUIT BREAKER ADD-ON

You can use the circuit breaker add-on to implement a circuit breaker pattern in your Node.js applications. This example shows how to use the circuit breaker add-on to report the failure of a remote service and to limit access to the failed service until it becomes available to handle requests.

3.1. OVERVIEW OF THE EXAMPLE APPLICATION

This example application consists of two microservices:

greeting-service

This is the entry point to the application. A web client calls **greeting-service** to request a greeting. **greeting-service** then sends a request that is wrapped in a circuit breaker object to the remote **name-service**.

name-service

name-service receives the request from **greeting-service**. The web client interface contains a toggle button that you can click to simulate the availability or failure of the remote **name-service**. If the toggle button is currently set to **on**, **name-service** sends a response to complete the greeting. However, if the toggle button is currently set to **off**, **name-service** sends an error to indicate that the service is currently unavailable.

3.2. EXAMPLE GREETING-SERVICE

greeting-service imports the circuit breaker add-on. **greeting-service** protects calls to the remote **name-service** by wrapping these calls in a circuit breaker object.

greeting-service exposes the following endpoints:

- The **/api/greeting** endpoint receives a call from the web client that requests a greeting. The **/api/greeting** endpoint sends a call to the **/api/name** endpoint in the remote **name-service** as part of processing the client request. The call to the **/api/name** endpoint is wrapped in a circuit breaker object. If the remote **name-service** is currently available, the **/api/greeting** endpoint sends a greeting to the client. However, if the remote **name-service** is currently unavailable, the **/api/greeting** endpoint sends an error response to the client.

- The **/api/cb-state** endpoint returns the current state of the circuit breaker. If this is set to **open**, the circuit breaker is currently preventing requests from reaching the failed service. If this is set to **closed**, the circuit breaker is currently allowing requests to reach the service.

The following code example shows how to develop **greeting-service**:

```
'use strict';
const path = require('path');
const http = require('http');
const express = require('express');
const bodyParser = require('body-parser');

// Import the circuit breaker add-on
const Opossum = require('@redhat/opossum');

const probe = require('kube-probe');
const nameService = require('./lib/name-service-client');
```

```
const app = express();
const server = http.createServer(app);

// Add basic health check endpoints
probe(app);

const nameServiceHost = process.env.NAME_SERVICE_HOST || 'http://nodejs-circuit-breaker-
redhat-name:8080';

// Set some circuit breaker options
const circuitOptions = {
  timeout: 3000, // If name service takes longer than 0.3 seconds,
                 // trigger a failure
  errorThresholdPercentage: 50, // When 50% of requests fail,
                                // trip the circuit
  resetTimeout: 10000 // After 10 seconds, try again.
};

// Create a new circuit breaker instance and pass the remote nameService
// as its first parameter
const circuit = new Opossum(nameService, circuitOptions);

// Create the app with an initial websocket endpoint
require('./lib/web-socket')(server, circuit);

// Serve index.html from the file system
app.use(express.static(path.join(__dirname, 'public')));
// Expose the license.html at http[s]://[host]:[port]/licences/licenses.html
app.use('/licenses', express.static(path.join(__dirname, 'licenses')));

// Send and receive json
app.use(bodyParser.json());

// Greeting API
app.get('/api/greeting', (request, response) => {
  // Use the circuit breaker's fire method to execute the call
  // to the name service
  circuit.fire(`${nameServiceHost}/api/name`).then(name => {
    response.send({ content: `Hello, ${name}`, time: new Date() });
  }).catch(console.error);
});

// Circuit breaker state API
app.get('/api/cb-state', (request, response) => {
  response.send({ state: circuit.opened ? 'open' : 'closed' });
});

app.get('/api/name-service-host', (request, response) => {
  response.send({ host: nameServiceHost });
});

module.exports = server;
```

3.3. EXAMPLE NAME-SERVICE

name-service exposes the following endpoints:

- The **/api/name** endpoint receives a call from **greeting-service**. If **name-service** is currently available, the **/api/name** endpoint sends a **World!** response to complete the greeting. However, if **name-service** is currently unavailable, the **/api/name** endpoint sends a **Name service down** error with an HTTP status code of **500**.
- The **/api/state** endpoint controls the current behavior of the **/api/name** endpoint. It determines whether the service sends a response or fails with an error message.

The following code example shows how to develop **name-service**:

```
'use strict';

const path = require('path');
const http = require('http');
const express = require('express');
const bodyParser = require('body-parser');
const cors = require('cors');
const probe = require('kube-probe');

const app = express();
const server = http.createServer(app);

// Adds basic health-check endpoints
probe(app);

let isOn = true;
const { update, sendMessage } = require('./lib/web-socket')(server, _ => isOn);

// Send and receive json
app.use(bodyParser.json());
app.use(bodyParser.urlencoded({ extended: false }));

// CORS support
app.use(cors());

// Name service API
app.get('/api/name', (request, response) => {
  if (isOn) {
    response.send('World!');
  } else {
    response.status(500).send('Name service down');
  }
  sendMessage(`${new Date()} ${isOn ? 'OK' : 'FAIL'}`);
});

// Current state of service
app.put('/api/state', (request, response) => {
  isOn = request.body.state === 'ok';
  response.send({ state: isOn });
  update();
});

app.get('/api/info',
  (request, response) => response.send({ state: isOn ? 'ok' : 'fail' }));
```

```
// Expose the license.html at http[s]://[host]:[port]/licenses/licenses.html
app.use('/licenses', express.static(path.join(__dirname, 'licenses')));

module.exports = server;
```

Additional resources

- For more information about deploying and running the example circuit breaker application, see the Node.js Runtime Guide.
- For more information about the member types that you can use with the circuit breaker add-on, see the Circuit Breaker Add-on 5.0.0 API Documentation .

CHAPTER 4. EXAMPLE APPLICATION USING A FALLBACK FUNCTION WITH THE CIRCUIT BREAKER ADD-ON

You can define a fallback function that you can use with the circuit breaker add-on in your Node.js applications. A fallback function specifies an action that is performed whenever a call to a remote service fails. For example, you can use a fallback function to send a customized message about service failures to the client. When the circuit breaker has an **open** state, the fallback function is executed continually for each attempt to contact the remote service.

4.1. FALLBACK METHOD SUPPLIED WITH THE CIRCUIT BREAKER ADD-ON

You can use the **fallback** method of the circuit breaker object to add your defined fallback function to the circuit breaker. If the call to the remote service fails after the circuit breaker **fire** method is executed, the fallback function is then called automatically.

For more information about the **fallback** method and other member types that you can use with the circuit breaker add-on, see the Circuit Breaker Add-on 5.0.0 API Documentation .

4.2. EXAMPLE APPLICATION CODE THAT DEFINES A FALLBACK FUNCTION

This example is based on the **greeting-service** that sends calls to the remote **name-service** that might fail. This example shows how to define a fallback function that prints the following message to the web console: **This is the fallback function**.

```
...

// Import the circuit breaker add-on
const Opossum = require('@redhat/opossum');

...

// Create a new circuit breaker instance and pass the
// remote nameService as its first parameter
const circuit = new Opossum(nameService, circuitOptions);

// Define a fallback function that will be called when
// the remote nameService fails
function fallback(result) {
  console.log('This is the fallback function', result);
}

// Use the circuit breaker's fallback method to add the
// fallback function to the circuit breaker instance
circuit.fallback(fallback);

...

// Greeting API
app.get('/api/greeting', (request, response) => {
  // Use the circuit breaker's fire method to execute the call
  // to the name service
```

```
  circuit.fire(`${nameServiceHost}/api/name`).then(name => {
    response.send({ content: `Hello, ${name}`, time: new Date() });
  }).catch(console.error);
});

...
```

Additional resources

- For more information about deploying and running the example cicuit breaker application, see the Node.js Runtime Guide.
- For more information about the member types that you can use with the circuit breaker add-on, see the Circuit Breaker Add-on 5.0.0 API Documentation .

CHAPTER 5. EVENT HANDLING FOR EVENTS EMITTED BY THE CIRCUIT BREAKER ADD-ON

The circuit breaker add-on emits events for the different types of actions that can occur based on the changeable states of the circuit breaker pattern. You can implement event handling in your Node.js applications to work with these different types of events and to perform some action when they occur. By using event handlers, you can control how your applications respond to the current behavior of the circuit breaker.

5.1. TYPES OF EVENTS EMITTED BY THE CIRCUIT BREAKER ADD-ON

The circuit breaker add-on emits the following types of events:

fire

This event is emitted when the circuit breaker **fire** method is executed to call a remote service.

reject

This event is emitted when the circuit breaker has an **open** or **half-open** state.

timeout

This event is emitted when the timeout period for the circuit breaker action expires.

success

This event is emitted when the circuit breaker action completes successfully.

failure

This event is emitted when the circuit breaker action fails and the circuit breaker returns an error response.

open

This event is emitted when the circuit breaker moves to an **open** state.

close

This event is emitted when the circuit breaker moves to a **closed** state.

halfOpen

This event is emitted when the circuit breaker moves to a **half-open** state.

fallback

This event is emitted when the circuit breaker has a fallback function that is executed after a call to the remote service fails.

semaphoreLocked

This event is emitted when the circuit breaker is at full capacity and it cannot execute the request.

healthCheckFailed

This event is emitted when a user-supplied health check function returns a rejected promise.

5.2. EXAMPLE APPLICATION THAT USES EVENT HANDLERS FOR CIRCUIT BREAKER EVENTS

This example shows how to listen for different types of events that the circuit breaker emits. In this example, a **fetch** function is used to request a microservice URL. Depending on the event type, the relevant message is printed to the web console.

```
...
```

```
// Import the circuit breaker add-on
const Opossum = require('@redhat/opossum');

...

// Create a new circuit breaker instance and pass the
// protected function call as its first parameter
const circuit = new Opossum(() => $.get(route), circuitOptions);

...

// Listen for success events emitted when the
// circuit breaker action completes successfully
circuit.on('success',
  (result) => {
    console.log(`SUCCESS: ${JSON.stringify(result)}`);
  }

// Listen for timeout events emitted when the timeout
// period for the circuit breaker action expires
circuit.on('timeout',
  (result) => {
    console.log(`TIMEOUT: ${url} is taking too long to respond.`);
  }

// Listen for reject events emitted when the
// circuit breaker has an open or half-open state
circuit.on('reject',
  (result) => {
    console.log(`REJECTED: The breaker for ${url} is open. Failing fast.`);
  }

// Listen for open events emitted when the
// circuit breaker moves to an open state
circuit.on('open',
  (result) => {
    console.log(`OPEN: The breaker for ${url} just opened.`);
  }

// Listen for halfOpen events emitted when the
// circuit breaker moves to a half-open state
circuit.on('halfOpen',
  (result) => {
    console.log(`HALF_OPEN: The breaker for ${url} is half open.`);
  }

// Listen for close events emitted when the
// circuit breaker moves to a closed state
circuit.on('close',
  (result) => {
    console.log(`CLOSE: The breaker for ${url} has closed. Service OK.`);
  }

// Listen for fallback events emitted when
// a fallback function is executed
```

```
circuit.on('fallback',
  (result) => {
    console.log(`FALLBACK: ${JSON.stringify(data)}`);
  }
```

Additional resources

- For more information about the member types that you can use with the circuit breaker add-on, see the Circuit Breaker Add-on 5.0.0 API Documentation .