Path Finder NodeJs Integration

Node Path Finder Library [MPL Internal]

This is library which provides solution for following tasks:

- Service Discovery via Consul (cluster mode)
- Support for GRPC client for communication (callback and promise support)
- Service Registration/Deregistration on consul
- Running the gRPC Server
- Add Service Check Registration and Meta Support
- Running Automatic Health Checks
- Automatic service deregistration on Server Shutdown
- Logging through Winston with support to update log level and enable/disable file level logs at runtime
- Read Configs from ZK using Path

Usage

Library Setup

Proceed with below steps:

Add in your `package.json` file, add the below line as dependency if not already present:

```
     "dependencies": {
       ...
       "lib-pathfinder-node":"bitbucket:mpl-gaming/lib-pathfinder-node.git#v1.0.7",
       ...
     }
```

Once this is done, delete your `node_modules` folder and run `npm install` in the main repo. And don't forget to commit and push to repo so that others can use the same as well.

At this point you are all set and good to go. In future, if we release a new version of the library and need that code, you can update the tag version in the `package.json` file.

In order to use this library, you need to initialize the same in the root/entry point of your application using below code.

```
import PathFinder from "lib-pathfinder-node";
// or
const { default: PathFinder } = require("lib-pathfinder-node");

/// other code
PathFinder.initialize({ appName: "service-rummy-deals" });
/// other code
```

> *<b>Note: </b> The `appName` in PathFinder.initialize is mandatory to provide as it is being used by logger running internally.*

Once the library setup is done, you can use this library for following use cases:

<br /> <br />

1. For Consul Service Discovery - IP Only

```
try {
  const URL = await PathFinder.getInstance().getServerUrl(SERVICE_NAME
[, TAG]);
  // make call to server here on the URL we got
} catch (e) {
  // Service is offline, and is not discoverable
}

// or

PathFinder.getInstance()
  .getServerUrl(SERVICE_NAME [, TAG])
  .then((URL) => {
    // make call to server here on the URL we got
  })
  .catch((e) => {
    // Service is offline, and is not discoverable
  });
```

Incase you need service discovery for certain country, this `getServerUrl` method accepts another optional parameter named `tag`, you can use that to pass in any tag you need, i.e. in or id or us etc.

<br /> <br />

2. Getting GRPC Client to call any RPC method [Recommended]

<br /> In order to communicate over gRPC, you can directly request this library to provide a client, using which the RPC calls can be made. <br /> We need to first provide all the proto definitions which we will be using at the pathfinder initialization time, so that protos are loaded at once.

```
// In entry point of the application
const protos = [
  {
    path: __dirname + "../../protos/HelloService.proto", // Path to the
proto file
    name: "service-hello", // Consul Service discovery name for this
service
  },
  {
    path: __dirname + "../../protos/RouterService.proto",
    name: "service-router",
  },
];
const isPromisified = false; // Set this to true if you want the client
method calls to be promisified
PathFinder.initialize({
  appName: "service-sample",
  protosToLoad: protos,
  promisify: isPromisified,
});
// we can use `PFInitOptions` to initialize our PathFinder

// in API middleware where you need to call any RPC method
try {
  const greeterClient = await PathFinder.getInstance().getClient({
    serviceName: "service-hello",
    serviceNameInProto: "GreeterService",
    // OPTIONAL PARAMETERS
    tag: "IN", // OPTIONAL: and is only required if the requested
client should call to server with this particular tag in consul
    options: {
      timeout: 5000,
      meta: {
        key: value,
        key2: value2,
      },
    },
    // OPTIONAL: if we need custom deadline timeout for this client, we
can supply `options.timeout`, and if we need to send any meta data
along with this request, we can provide a plain object for that on
`options.meta` key
  });

  // if we supplied `promisified` as `false` in the `PathFinder.
initialize` call
  greeterClient.sendHello(request, (err, response) => {
    if (err) {
      // GRPC Error occurred
    } else {
      // Response is received and can be used
    }
  });

  // if we supplied `promisified` as `true` in the `PathFinder.
initialize` call
  greeterClient
    .sendHello()
    .sendMessage(request) // whatever request body that we need to send
    .then((response) => {
      // Response is received and can be used
    })
    .catch((err) => {
      // GRPC Error occurred
    });
} catch (e) {
  // Either Service is offline, and is not discoverable
  // or the proto is not loaded provided in PathFinder.initialize call
}
```

> *<b>Note:</b> Since the service discovery and consul calls are done over http, these calls are promisified. Majorly all the PathFinder.getInstance().method() should either be chained with `.then` or prefixed with `await`.*

<br /> <br />

3. Registering Your Service to Consul

<br /> In order to register your service on consul, you need to call

```
import { PFServer } from 'lib-pathfinder-node'
// or
const { PFServer } = require('lib-pathfinder-node');

  // Start the GRPC server, after that
  const server = new PFServer("service-group-name", 50051, "in",
"custom-tag");
  server.addService(pathToProtoFile,
pathToFileExportingAllTheRpcMethods);
  server.start();

  // we can also chain if we want to add multiple services
  server.addService(...).addService(...).addService(...).start();

  // Service is registered on the consul
```

<br /> <br />

4. Using `Logger` to log the data and error consistently

```
import { Logger } from "lib-pathfinder-node";
//or
const { Logger } = require("lib-pathfinder-node");

// use anywhere
Logger.error("SOME CRITICAL ISSUE");
Logger.warn("SHOULD WE FIX IT NOW??");
Logger.info("CHILL HAI");
Logger.debug("PTA NHI KU NHI CHL RAHA, DEBUG KRTE HAI");
```

When the logger runs with Grpc Server initiated using `PFServer` class. It starts a logger management service to update the log level, enable /disable file level logs at runtime. When the logger runs with a express server, in order to make it configurable at runtime using REST APIs, add the `loggerMiddleware` to the express app using below code:

```
import PathFinder, { loggerMiddleware } from "lib-pathfinder-node";
// or
const {
  default: PathFinder,
  loggerMiddleware,
} = require("lib-pathfinder-node");

PathFinder.initialize({ appName: "dashboard-api-gateway" });
const app = Express();

app.use(loggerMiddleware());
```

> *<b> Note:</b> This service will not register your service to consul if you are running your application locally on your machine. In order to inform the library, that you are running locally, set the `process.env.NODE_ENV` to either of the `DEV`, `dev`, or `development` in start/inspect scripts. Also, the logger service will not open file level transport when running with NODE_ENV set to DEV.*

for example, in your package.json file

```
  "start": "NODE_ENV=DEV node src/inex.js"
```

<br /> <br />

5. Reading Configs from Zookeeper

This library supports reading configs from Zookeeper which would automatically get updated when the node data changes. In order to use that, refer below code:

```
import { ConfigProvider } from "lib-pathfinder-node";
// or
const { ConfigProvider } = require("lib-pathfinder-node");

// your interface for configs stored in zk
interface IZkConfigs {
  strKey: string;
  numKey: number;
}
// any of your function
const main = async () => {
  try {
    const configs = await ConfigProvider.getInstance<IZkConfigs>().
getConfigs(
      CONFIG_PATH
    );
    console.log("Configs are: ", configs);
    console.log("strKey is: ", configs.strKey);
    console.log("numKey is: ", configs.numKey);
  } catch (e) {
    // THIS THROWS ERROR IF THERE WAS ANY ISSUE WHILE READING ZK CONFIGS
    console.log(
      "Unable to read configs from zk: use default values here for
fallback case"
    );
  }
};
```