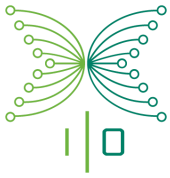
Backend - System Setup



System requirements

Operating system

Any OS that runs NodeJs and Postgres might be used. Otherwise, if Docker container service is going to be use used to run the application, docker should be installed.

We recommend Ubuntu to be used to run the backend app.

Node

The Node version we're currently using in this project is 4.2.3. In order to download this version, you can either download it from their web or use Node Version Manager (nvm), following the tutorial listed below.

References:

- Node download: https://nodejs.org/download/release/v4.2.3/
- NVM tutorial: http://stackoverflow.com/questions/7718313/how-to-change-to-an-older-version-of-node-js

NPM

Node Package Manager is the best tool for importing and updating external packages. The current version in this project is 2.14.7, and it's automatically installed when installing Node, but in case you need to install it manually, you can download it (see the references for the link).

Please make sure to run `npm install --save` once it's installed, so it downloads all the dependencies defined in package.json.

References:

- NPM download: https://registry.npmjs.org/npm/-/npm-2.14.7.tgz

PostgresQL

The database used is psql (PostgreSQL) version 9.4.8. PostgreSQL is a powerful, open source object-relational database system.

References:

- PostgresSQL download for Ubuntu: https://www.postgresql.org/download/linux/ubuntu/

System Configuration

Settings file

Backend application settings are divided in two sections when sensitive data needs to be added. A plain javascript file contains all the basic structure and configuration, and a private encrypted json file that extends those basic settings with private ones.

Each environment will have its corresponding section; *development, test* and *ci* are all public and don't have encrypted data, while *staging* and *production* does include them. The configuration can be found in *src/conf/environment/.*

Any key on the private section overrides the public one while merging, so any key can be arbitrarily moved from public to private as desired.

Settings breakdown

DB Connection

Defines the postgres connection, localhost in this case.

```
db: {
  connectionString: 'postgres://backendapp_user:XXXXXX@localhost/backendapp',
},
```

Logger

The application has a very detailed logging level, here you can define that level and define the output style, *prettyStdOut* in this case.

```
// Logger Streams
```

```
logStreams: [ {
  level: 'info',
  stream: prettyStdOut
} ],
```

Bitgo

Bitgo is the Bitgo Blockchain API provider used, this configuration will normally be private, as queue accessToken allows handling the Wallets and transactions operations.

```
bitgo: {
  useProduction: false,
  accessToken:
'************************************************************',
  previous: 'http://******************/bitgotest/index.php',
  webhook : 'http://localhost',
  env: 'test'
},
```

Sales App

In this section you should define the sales app api url and each of the endpoints used to communicate with.

```
const SALES_APP_ENDPOINT = 'http://localhost:8880/api/v1'
...
salesapp: {
  endpoint: SALES_APP_ENDPOINT,
  healthCheckEndpoint: SALES_APP_ENDPOINT + '/' + 'healthCheckEndpoint',
  holdingWalletNotificationEndpoint: SALES_APP_ENDPOINT + '/' +
'holdingWalletReceivedBtc',
  invoiceFundsReceivedNotificationEndpoint: SALES_APP_ENDPOINT + '/' +
'invoiceAddressReceivedBtc',
```

```
  commissionReadyEndpoint: SALES_APP_ENDPOINT + '/' +
'commissionsMovedToWallet',
  commissionPayoutDeliveredEndpoint: SALES_APP_ENDPOINT + '/' +
'commissionsPaid'
},
```

Backup Public Key

Every Wallet in the app is a MultiHD Wallet that requires 2 out of its 3 signatures to operate. One signature is kept by bitgo, and the other ones are stored encrypted in the DB to operate. But the third signature, the backup one, it's not needed to operate and it's store separately in a safe place. The Public Key is needed while creating new Wallets.

```
defaultBackupPub:
'**********************************************************************
*********************************',
```

Wallets and Addressed

Here you can configure the Commission and Holdings Wallets Ids, as well as the exodus Address and percentage and the *minConfirmations* required to consider a Transaction confirmed.

```
exodusAddress: '**********************************',
exodusPercentage: 0.8,
holdingWalletId: '**********************************',
commissionWalletId: '**********************************',
holdingFeeAddresses: [ '**********************************',
'**********************************' ],
customRefundAddress: '**********************************',
minConfirmations: 1,
```

Secret

All sensitive data in the DB is store encrypted by a combination of this secret, code, and environment as well as a salt on each entry.

```
secret: '**************************',
```

AWS Configuration

Logs are currently stored using AWS Cloud Watch services. here you can define the corresponding credentials.

```
AWS_accessKeyId: 'AWS_accessKeyId',
AWS_secretAccessKey: 'AWS_secretAccessKey',
```

Queue

Many processes are handled by an asynchronous queue. Here you can define the iteration time (in milliseconds) it runs on:

```
queuePollTime: 1000 * 60 // 1 min
```

New Relic

New Relic is used as a monitoring tool for the backend, it's very useful for detecting any possible downtime, network and application errors and making an analysis of application requests.

```
newRelicLicenseKey: '************************************',
```

Routes Access Permissions

There are three level of permissions to access any endpoint of the backend API:

- Public: open to the network
- Read only (R): Requires at least a Read or ReadWrite Token as authorization header.
- Read & Write (RW): Requires a ReadWrite Token as authorization header.

Each route is mapped to an authorization requirements in the *src/conf/routeAccess.js* file:

```
module.exports.routes = [
  /** PUBLIC **/
  { key : 'GET //healthcheck', access : '' },
  { key : 'POST //bitgoCallback', access : '' },
  /** PRICES **/
  { key : 'PUT //price/runService', access : 'RW' },
  { key : 'GET //price/makeRequest/.*', access : 'RW'},
  { key : 'GET //price/.*', access : 'R'},
  /** SALESAPP **/
  { key : 'GET //wallets/.*', access : 'R'},
  { key : 'GET //invoiceWallets/.*', access : 'R'},
  { key : 'POST //wallets.*', access : 'RW'},
  { key : 'PUT //wallets.*', access : 'RW'},
  { key : 'POST //invoiceWallets.*', access : 'RW'},
  { key : 'PUT //invoiceWallets.*', access : 'RW'},
  { key : 'PUT //holdingWallet/newAddress', access : 'RW'},
  /** MANAGEMENT**/
  { key : 'PUT //fixHoldingWhitepolicies', access : 'RW'},
  { key : 'PUT //cleanCommissionWalletWhitelist', access : 'R'},
  { key : 'GET //notifications/.*', access : 'R'}
]
```

Each key represents an endpoint and the HTTP protocol {PUT, GET, POST} matching and the access code { “”, “R”, “RW” } defines the access level. The “*” wildcard matches any character; the array is processed in sequence, so it will take the first match for each endpoint.

Note: Refer to “**IOHK Backend - SalesApp Authentication”** document to further details on the authorization process.

Deployment

Deployment process is as simple as starting up a NodeJs application. In order to do so the following steps are needed:

1. Get source code to be deployed

2. Run `npm install`
3. Export the following environment variables:
    a. `NODE_ENV={environment}` where environment is { staging, production }
    b. `SECRET={secret}` *see next section (Encrypting Private Settings)*
4. Run `node app.js`

In order to improve the deployment process, it's suggested to use Docker as a container service. The following steps are required:

1. `export RELEASE_VERSION={release_version} where release_version is a tag, for example 1.0.0, 1.1.3, etc`
2. `docker build inputoutput/backend-app:$RELEASE_VERSION`
3. `docker stop backend-app`
4. `docker run -d -e "NODE_ENV=${environment}" -e "SECRET={secret}" --restart=on-failure:10 -p 8888:8888 --name backend-app inputoutput/backend-app:$RELEASE_VERSION`

The following `Dockerfile` can be used to achieve the steps stated above:

```
FROM node:4.2.3

RUN mkdir -p /usr/src/app
WORKDIR /usr/src/app

COPY package.json /usr/src/app/
COPY . /usr/src/app

RUN npm install && npm install -g pm2 && pm2 install pm2-logrotate && pm2 set pm2-logrotate:max_size 500M

EXPOSE 8888

CMD ["pm2", "start", "app.js", "--no-daemon"]
```

Encrypting Private Settings

As mention in the Settings section, any sensitive key should be uploaded in an encrypted manner. The encryption key is sliced in two, the first part should go to a "*SECRET*" environment variable, and the second part fixed in the code. This process can be better explained by the line in the settings itself:

```
const decrypted = encryptionService.decrypt ( process.env.SECRET + '**************'
, data )
```

Even if the encryption process is standard following the *sjcl*[1] one, there are two grunt[2] scripts to help the user in charge of deploy.

```
grunt encrypt_env_config --env environment
grunt decrypt_env_config --env environment
```

Example for staging.json:

Given this staging.json file:

```
➜  src git:(master) ✗ grunt decrypt_env_config --env staging
Running "prompt:secret" (prompt) task
? Please enter decrypt/encrypt secret **********

Running "decrypt_env_config_task" task

Done, without errors.
➜  src git:(master) ✗ cat conf/environment/staging.json
{
        "bitgo": {
                "useProduction": false,
                "env": "test",
                "enterpriseId": "[redacted]",
                "accessToken": "[redacted]",
                "webhook": "https://[redacted]/api/v1/bitgoCallback"
        },
        "secret": "[redacted]",
        "AWS_accessKeyId": "[redacted]",
        "AWS_secretAccessKey": "[redacted]",
        "customRefundAddress": "[redacted]"
}
```

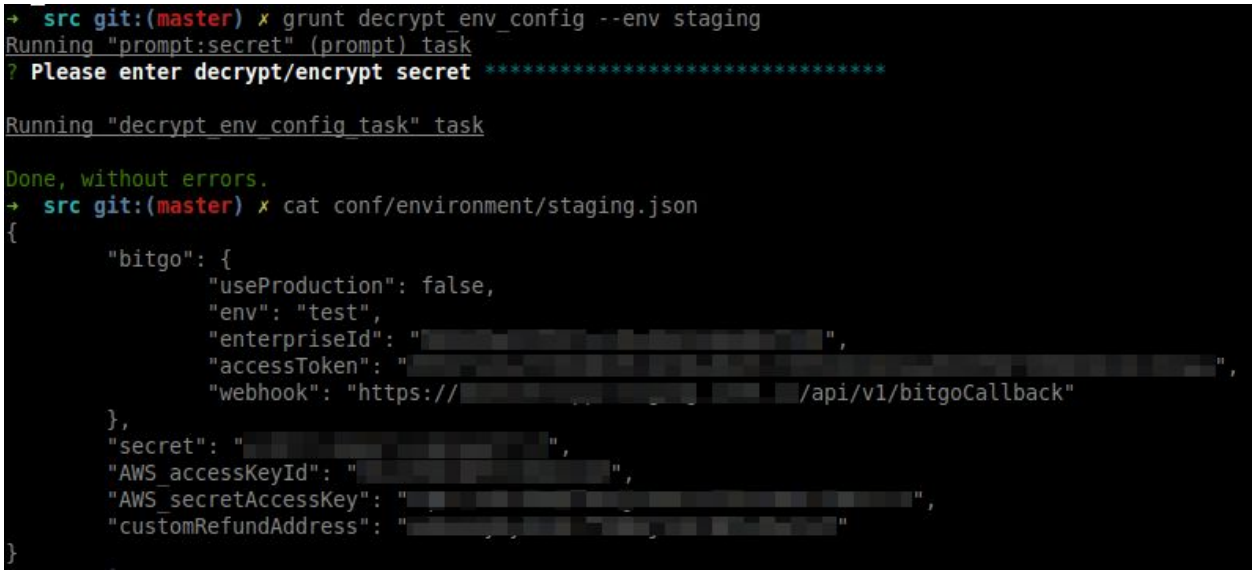
We can run the encryption process (using both key parts):

```
➜  src git:(master) ✗ grunt encrypt_env_config --env staging
Running "prompt:secret" (prompt) task
? Please enter decrypt/encrypt secret **********

Running "encrypt_env_config_task" task

Done, without errors.
```

The file now contains an encryption of the previous:

[1] https://crypto.stanford.edu/sjcl/
[2] https://gruntjs.com/

And if we run the decrypt scrypt, we get the same initial file:



Access Token generation

As mentioned in the “Routes Access permissions” in the System Configuration section, each endpoint can have access restriction. This permission is granted by a specific Token that needs to be previously generated and delivered to the allow third parties, the sales app should have RW access for example.

To generate Read (R) and ReadWrite (RW) access tokens, the user will need a grunt script, complete secret key, and have write access to the *database* to insert the new Token. As the token is salted and encrypted, there is no risk on data manipulation on a DB level.

For further details on Token generation please refer to the “*Backend - SalesApp Authentication*” Document, section “*Access Token Generation*”.

Recommendations

For deploying Backend in an easy and practical way, we suggest using Docker Containers.

Also, don’t forget to set a fixed version in the nvm packages to prevent any possible errors inserted by a package update.