Contents

1. Installation

Lynxari is designed to operate on a wide variety of Linux operating systems running on ARM and Intel CPUs. Windows 10 and other less popular operating systems are addressed as required. Linux Distributions based on Debian, Red Hat, and SuSe as well as macOS will run Lynxari without issue. The core Lynxari product does not have any dependencies, so no other software installation is required. While the program employs node.js, the runtime includes the proper version of node, so separate installation is not required, but an existing installation of node will also not interfere.

Installing Lynxari on an individual machine or server is a straight-forward task. The steps involved are downloading, selection of a location, and unpacking.

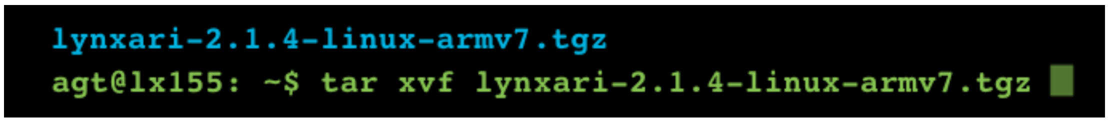
1.1. Downloading the platform

Complete Lynxari packages are available at agilatech.com/software. From that location, you can select the version and operating system desired. While all versions come with an evaluation license, ongoing use requires the purchase of a license key. Note that license keys are tied to a major version, so you should select a version which matches the key you have purchased or intend to purchase.

Lynxari can run in any location and with any user permissions. However, for security purposes, some care should be taken to avoid exposing the programs and configuration files to unauthorized users. While it is beyond the scope of this document to describe O/S filesystems and user permissions, Agilatech recommends that a user account is specifically created to run the Lynxari software, and only that user or the superuser shall have access to the files.

1.2. Unpacking

Both Linux and macOS versions are compressed using gzipped tar. Various graphical apps exist for both operating systems to unpack this type of file, but the simplest and quickest method is to simply use a terminal command-line interface. Linux and macOS both include the tar utility in their distributions which are invoked from the command line.



```
lynxari-2.1.4-linux-armv7.tgz
agt@lx155: ~$ tar xvf lynxari-2.1.4-linux-armv7.tgz
```

Uncompressing the package file will result in a directory of the same name, unless you specifically request otherwise.

1.3. Install as a system service

There are many different methods which may be used so that the operating system runs the lynxari process automatically. As of 2018, the majority of Linux distributions have adopted systemd as the default init system, so that is what is presented here.

1.3.1. Linux and macOS systemd

In just a few easy steps, you can have Lynxari running automatically at boot, and restarting in case the program ever stops prematurely (the marketing department doesn't like the word crash).

a. Create the service file. An example is included with Lynxari in the toolbox/systemctl directory. It looks like this:

```
[Unit]
Description=Lynxari IoT Server

# An example of requiring another service to run first
After=mysql.service

[Service]
WorkingDirectory=/home/agt/iot
ExecStart=/home/agt/iot/lynxari
Restart=always
# Restart service after 1/2 second if node service crashes
RestartSec=500ms
# stdout and stderr to journalctl
StandardOutput=journal
StandardError=journal
SyslogIdentifier=lynxari-server
User=agt
Group=agt
Environment=NODE_ENV=production PORT=1107

[Install]
WantedBy=multi-user.target
Alias=lynxari.service
```

This sample could be used after editing the values for `After`, `WorkingDirectory`, `ExecStart`, `User`, and `Group`. You may wish to add or modify other values as well, and '`man systemd.service`' will explain all the options and possible values.

b. As root, copy the service file to the proper location:

```
sudo cp lynxari.service /lib/systemd/system
```

c. As root, Enable the service:

```
systemctl enable lynxari.service
```

d. When ready to begin, as root, start the service:

```
systemctl start lynxari.service
```

If you ever need to stop or restart the service, simply substitute ‘stop’ or ‘restart’ in place of the ‘start’ command above. As written, the example service file uses journalctl for logging. Too view the log, issue the command ‘`sudo 'journalctl -u lynxari.service'`.

2. Platform Configuration

All program, module, and configuration files reside within the main Lynxari platform directory. Some files have configurable paths, and outside files may be linked to, but for the most part everything required fits neatly inside the main directory.

Lynxari embraces node.js and the node package manager (npm) because such technologies offer excellent performance, utility, and interface capabilities, coupled with an ease of configuration and expansion. In order to support all the advantages, Lynxari does adhere to the conventions of npm, which means that all device driver modules and application modules are expected to reside in the node_modules directory in the main Lynxari platform directory.

Configuration of the Lynxari platform itself is straightforward. Except for the license key itself, all configuration entities are JSON (Javascript Object Notation) objects, which offers easy readability by humans and machines. A good overview of the format and grammar is available at http://json.org.

2.1. Server configuration

The `configuration.json` file for the server defines the name, host, port, and the location of the TLS certificates. The name, host, and port are simple key:value pairs, while certificates is itself an object made up of key:value pairs.

The name should be a string, representing the name of the server and how it will appear in API calls, queries, and all other places where the server is referenced by name. If the name entry is not given, the server name defaults to the machine hostname.

The host defines the hostname or IP under which the server runs. Note that in order to work properly, the host must a valid for one of the interfaces on the machine. The host may be a fully qualified domain name, an IP address, a local machine name, or localhost. If the host entry is not given, the server host defaults to localhost.

The port is a numeric value which defines the TCP port on which the server will listen and accept connections. Agilatech uses the port 1107 by default, and this is the port listened on if the port entry is not given. The port should be greater than 1024 and less than 49152 and avoiding other well-known ports used by other services.

The update entry defines whether or not the program will be updated automatically. If the value is "automatic", then every time the program is started, a check will be made to see if an update is available. If the entry is missing or the value is anything else, then automatic updating will not occur.

The certificates object contains key:value pairs for the file locations of keyfile, certfile, and cafile. The keyfile is the private server key, while the certfile is the public key certificate, and the cafile is the certificate signing authority (if necessary, for instance for a self-signed certificate).

A complete example is:

```
{
  "name":"agt155",
  "port":1107,
  "update": "automatic",
  "certificates": {
    "keyfile":"auth/certificates/private/agt155.key.pem",
    "certfile":"auth/certificates/certs/agt155.cert.pem",
    "cafile":"auth/certificates/certs/ca-chain.cert.pem"
  }
}
```

2.2 Lists for devices, applications, and peers

When Lynxari first starts, it reads three config files to load devices and applications, and to create peer connections. These files are named `devlist.json`, `applist.json`, and `peerlist.json`, respectively.

2.2.1. devlist

The `devlist.json` file is an object containing an array of devices. Each element in the devices array configures a device. A device contains a name, a module, and options. The name is a key:value pair, of the form `“name”:”<name of device>”`. The module key:value uses “module” as the key, and the Javascript module name conforming to npm conventions. The module does not have to be hosted by npm, but does reside on the local filesystem in ‘`node_modules`’. Device drivers and their associated modules are covered in detail in the document “Extending the Lynxari Platform”.

The device object may also contain an object listing options to be passed directly to the driver module. The structure of this options object is dependent upon the driver module itself, with some drivers accepting complex options, and other only accepting a few key:value pairs. Details about the options object are covered further in section 4. Device Driver Configuration.

Here is a example of a minimal `devlist.json` with two devices:

```
{
  "devices":[
    {
      "name":"TSL2561",
      "module":"@agilatech/lynxari-tsl2561-device"
    },
    {
      "name":"HTU21D",
      "module":"@agilatech/lynxari-htu21d-device"
    }
  ]
}
```

In this example, other modules may be present, but the two device driver modules must appear in the '`node_modules/@agilatech`' directory.

2.2.2 applist

`applist.json` is an object containing an array of apps. Every application listed in the array will be loaded and started by Lynxari when it starts. Each element in the array should list the name of the application and the Javascript module name. Since the module name conforms to npm standards, the first directory in the path is assumed to be `'node_modules'`, so it does not appear in the module name.

Here is an example of an `applist.json` which loads four applications at startup:

```
{
  "apps":[
    {
      "name":"powerup",
      "module":"@agilatech/lynxari-powerup-application"
    },
    {
      "name":"mysql",
      "module":"@agilatech/lynxari-mysql-application"
    },
    {
      "name":"fileout",
      "module":"@agilatech/lynxari-fileout-application"
    },
    {
      "name":"snapshot",
      "module":"@agilatech/lynxari-snapshot-application"
    }
  ]
}
```

2.2.3. peerlist

`peerlist.json` is an object containing an array of Lynxari peers. Upon startup, Lynxari reads this file and initiates a peer connection to every host listed. Each element of the peers array should contain a key:value pair which lists the name and host. The name is simply a string by which the host is referred. The host is the URL of the host machine, to include the protocol, host, and port number. No authorization strings nor paths should appear in this definition.

Here is a example of a `peerlist.json` which defines three peers:

```
{
  "peers":[
    {
      "name":"agt1",
      "host":"https://agtiot.com:1107"
    },
    {
      "name":"db-local",
      "host":"https://192.168.20.140:1204"
    },
    {
      "name":"log",
      "host":"https://agtiot.com:1108"
    }
  ]
}
```

2.3 License Key

The license key file is located in the Lynxari platform directory in `'auth/license/key'.` Lynxari must find this file in that location. By default, every distribution is shipped with an evaluation license key which allows the system to run for up to 2 hours. In order to convert the system from evaluation to production, a proper license key must be installed. Simply copy-and-paste the license key provided at purchase into the license key file in `'auth/license/key'` to activate a full-featured production system.

3. Web-based Configuration

It can be tedious to remember all the locations and formats of the various config files. To help make things easier, a Web-based configuration tool has been included.

3.1 Installing the configuration Web server

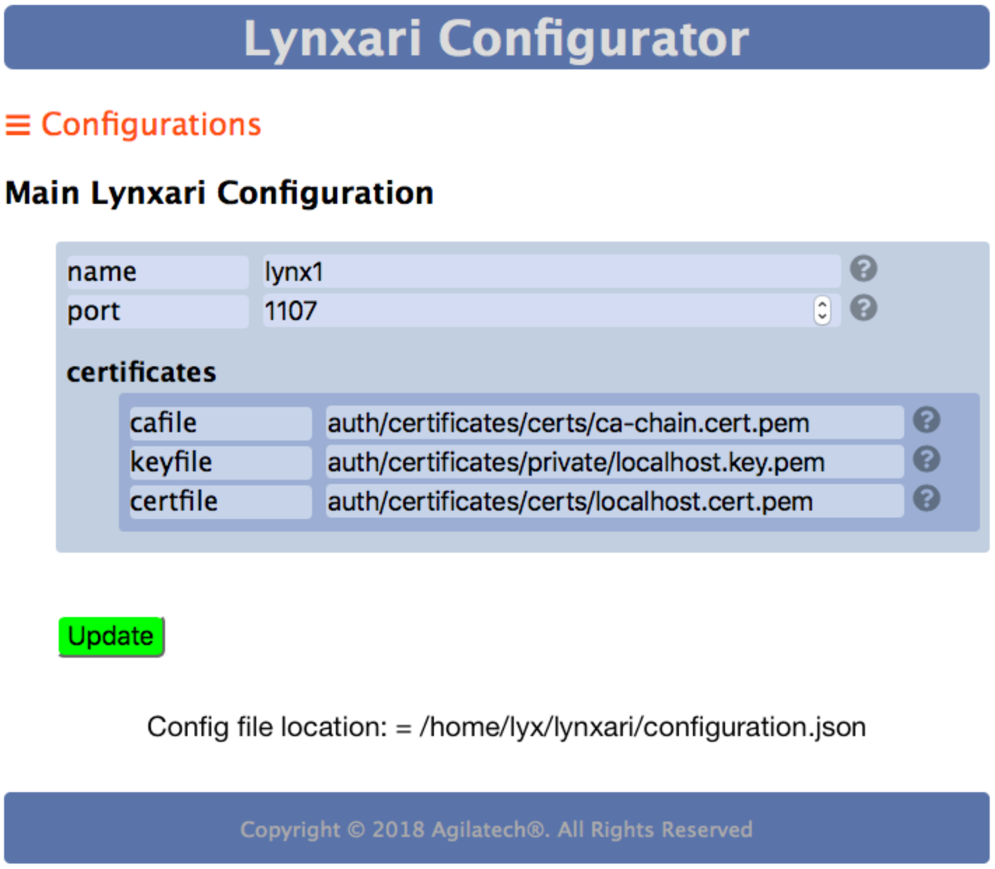
In order to save space, and also to ensure that the latest Web modules are used, the platform does not ship with pre-installed Web server modules. For this reason, if the Web configuration will be used, it must first be installed. There is a tiny convenience script named 'install' in the webconfig directory. Execute this script to install the required modules. Installation requires about 38Mb of disk space.

3.2 Running the configuration Web server

The webconfig Web server must be started in order to use the Web-based configuration tool. Starting it is a simple matter of executing the 'start' script link in the webconfig directory. This will start a node Express Web server listening on port 8107. This is intended to be short-use tool only for development to make editing config files easier. If the Web server remained running in a production environment, that would represent a gaping security hole. For this reason, the server will automatically exit after 30 minutes to protect against mistakenly leaving it running.

3.3 Using the configuration Web server

With the webconfig Web server running, point a Web browser at http://<host>:8107 where <host> is the hostname or IP address of the machine running the Web server. The webconfig root page shows the main Lynxari configuration, and should look like this:



Main Lynxari Configuration

name	lynx1
port	1107

certificates

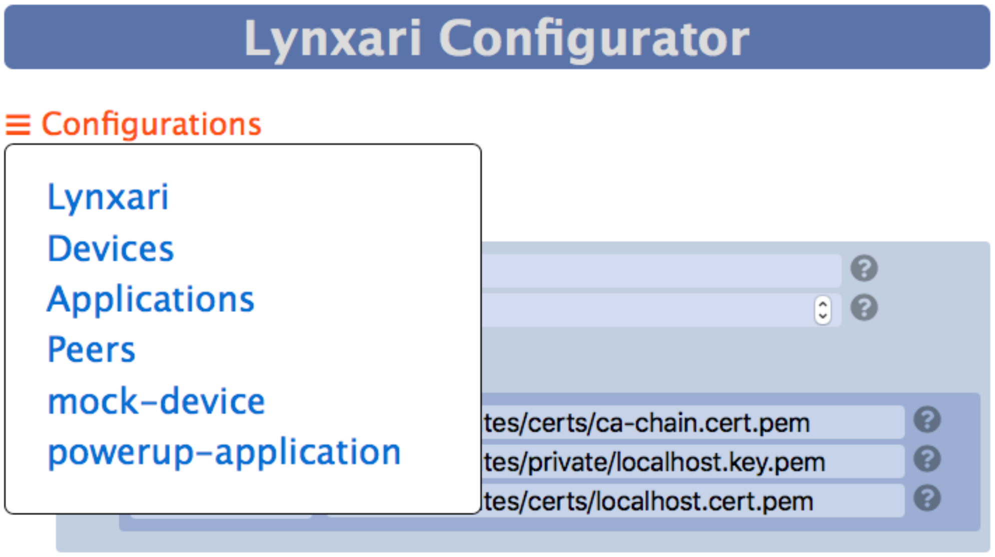
cafile	auth/certificates/certs/ca-chain.cert.pem
keyfile	auth/certificates/private/localhost.key.pem
certfile	auth/certificates/certs/localhost.cert.pem

Config file location: = /home/lyx/lynxari/configuration.json

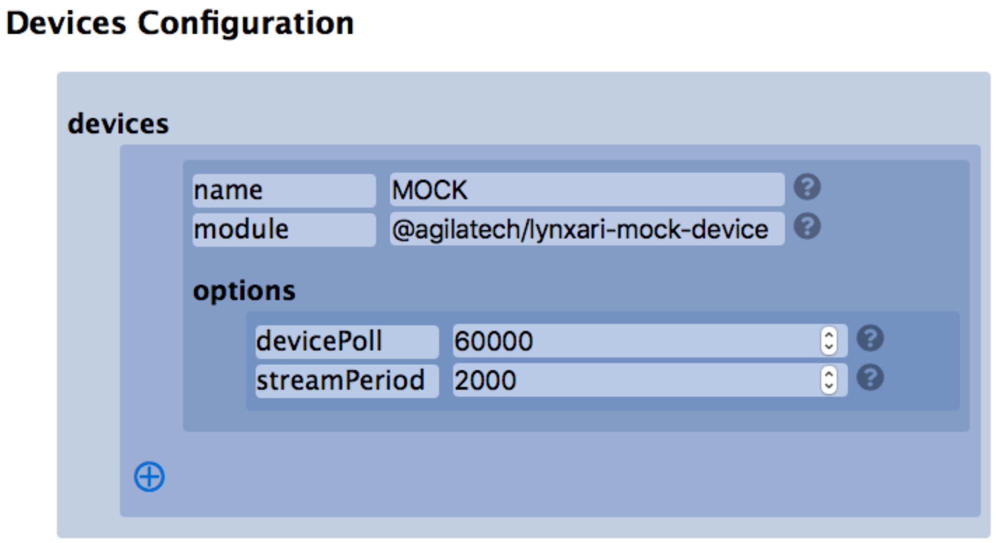
This shows the configuration held in the 'configuration.json' file in the Lynxari directory. Note that the absolute filesystem location of the current config file is always listed below.

All configuration files end up resolving to key:value pairs, so the webconfig tool is organized this way, with the keys appearing on the left and values appearing on the right. Values may be modified or deleted, but keys may not. Pairs within an object are grouped together, and nested objects are shaded in increasingly darker shades. The 'Update' button must be pressed for any changes to take effect. Upon a successful update, the file will be re-read and redisplayed, showing the new values.

The only menu is the upper left 'Configurations' drop-down. The menu always contains the Lynxari, Devices, Applications, and Peers configurations, as well as any and all other 'config.json' files found under the Lynxari directory. The search for new config files is dynamic and continuous, so that module additions can be configured immediately without restarting the webconfig server.

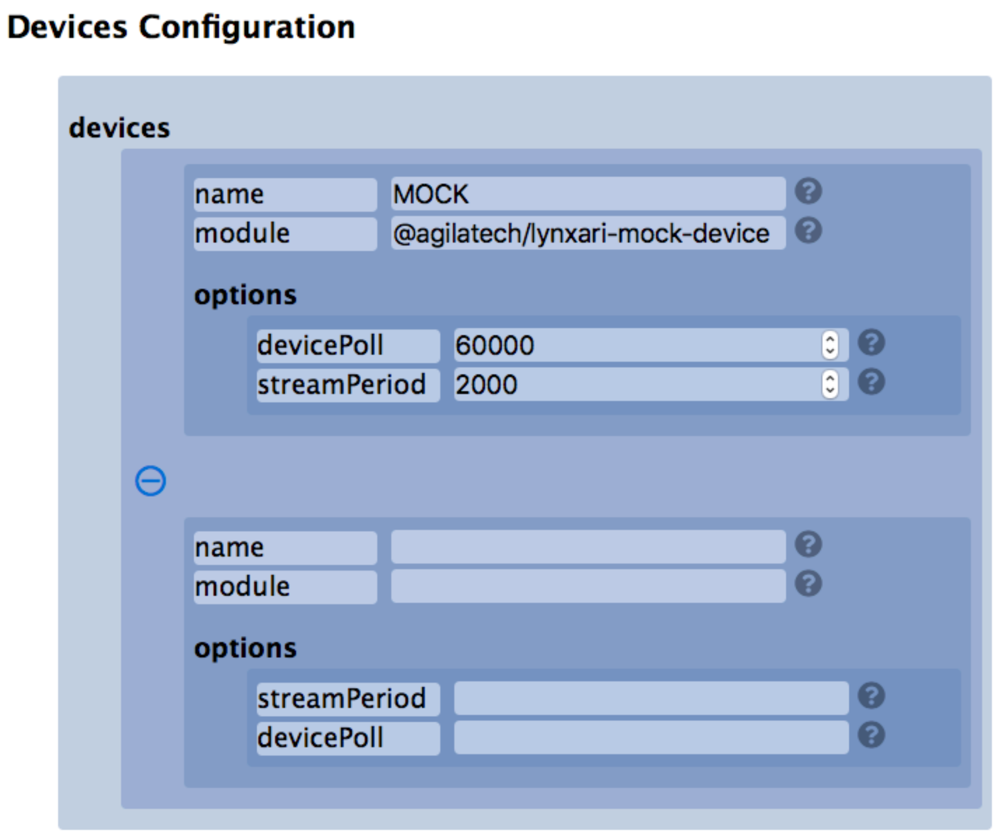


For the config files which have known repeatable sections (Devices, Applications, and Peers), a small ⊕ symbol appears under the last repeatable section to indicate the addition of a new section.



name	MOCK
module	@agilatech/lynxari-mock-device
devicePoll	60000
streamPeriod	2000

Clicking on the ⊕ symbol shows a new section with blank values. Fill in the blanks and click 'Update' to save the file with the new section. Only one new section may be added at a time.



name	MOCK
module	@agilatech/lynxari-mock-device
devicePoll	60000
streamPeriod	2000
name	
module	
streamPeriod	
devicePoll	

The webconfig tool is fairly simple and self-explanatory. It is not intended to be used to create complex new config files, but it does offer an easy alternative to text editing.

4. Securing the server

4.1 TLS encryption

Lynxari expects to run with encryption active. In fact, you must specifically start it with a `-u` or `--unsecure` flag in order to force it to run without encryption. But to use encryption, you must define a few key files for the program.

Lynxari employs Transport Layer Security (TLS) protocols to encrypt all traffic to and from the server. This public-key cryptography requires at least two keys: 1) the private key used by the server to encrypt a message, and 2) the public key used by clients to decrypt the message.

It is beyond the scope of this installation document to describe the TLS protocol, but sufficient to state that the keys comply with standard TLS 1.2 public key infrastructure policies. Appendix A shows how to create a certificate authority and self-signed public and private key certificates if your system does not have a certificate issued by a trusted 3rd party.

The private key is of course private, and all care should be taken to protect and enforce this. At no time should the private key ever be disclosed to any other system or server. The directory containing the key as well as the private key file itself should be readable only by the account owner of the Lynxari system in order to prevent undisclosed access to other users. It is used solely by the Lynxari server to encrypt messages. Although the default location of the private key is within the Lynxari platform directory in `'auth/certificates/private'`, the location may be anywhere on the filesystem, and is defined in the main configuration file.

The public key is contained within the public key certificate, and is used by the client to encrypt and decrypt messages from the server. The default location of the public key is within the Lynxari platform directory in 'auth/certificates/certs', but again the actual location is defined in the main configuration file.

Customarily, the keys are signed by a trusted 3rd party certificate signing authority. A list of certificate authority providers is contained within Lynxari itself, similar to the list contained by Web browsers such as Firefox or Chrome. However, if you use a self-signed certificate or one not well-known, it is necessary to explicitly install the CA certificate on the filesystem and list the location in Lynxari's main configuration file. Again, the default location of the CA certificate is within the Lynxari platform directory in 'auth/certificates/certs' but can be anywhere on the system.

Here again is the example configuration file shown in section 2.1:

```
{
  "name":"agt155",
  "port":1107,
  "update": "automatic",
  "certificates": {
    "keyfile":"auth/certificates/private/agt155.key.pem",
    "certfile":"auth/certificates/certs/agt155.cert.pem",
    "cafile":"auth/certificates/certs/ca-chain.cert.pem"
  }
}
```

In this example, the private key is `'agt155.key.pem'` and stored in the Lynxari platform directory in 'auth/certificates/private'. The public key/certificate is `'agt155.cert.pem'` and stored in `'auth/certificates/certs'`. Since these keys are self-signed, a certificate authority file is also defined, and is stored in `'auth/certificates/certs/ca-chain.cert.pem'`.

Without authentication, encryption is not of much value because the server must be able to verify that the client is not an imposter. Similarly, authentication without encryption is not useful since passwords should not be sent over unencrypted cleartext channels. Therefore, these two concepts go hand-in-hand, and Lynxari operates by default with both or by specific request with none at all. The system uses a username:password authentication system in order to verify authorized client access.

There are two types of connections used by Lynxari, 1) HTTP REST API and 2) peer server. Since the system needs to discriminate between access to these to types, each has its own access controls.

4.2 HTTP authentication

The HTTP REST API is used by outside clients to make API requests of the server. The server maintains a collection of valid username:password combinations, one of which may be used by a client to access the requested API. The username:password credentials are stored in a file with the Lynxari platform at `'auth/credentials/access.json'`. Since this file contains secret passwords, it should be protected in the same manner as the server's private key. The file should be readable only by the Lynxari account owner. The username:password combinations are defined as key:value pairs in the file. Here is an simple example showing two valid access combinations:

```
{
  "agt":"abcd1234",
  "sp-access":"rx7YYBq77m4B"
}
```

4.3 Peer authentication

Peer link connections are authenticated in a similar manner. A difference however, is that a server wishing to initiate a link with another peer must know a valid username and password which will be accepted by the receiving peer. This information is kept in a special object section named "peers" in the `'auth/credentials/access.json'` file. This section contains another nested object section for each and every link which will be connected. A typical section would look like this:

```
"peers": {
  "192.168.32.110:1107":{
    "username":"agt",
    "password":"abcd1234"
  },
  "vs1.cloudserver.com:1107":{
    "username":"vsl",
    "password":"RdQ-rsU-p3E-632"
  }
}
```

A peer object requires the key to be a valid server URL, including the port. Each peer object must then contain a 'username' and password' entries containing credentials for the receiving server. If an attempt is made to link with a peer which does not have an entry in the peers section, then the connection will not be initiated.

In this example, two peers are configured. When a link connection request to vs1.cloudserver.com on port 1107 is initiated, that receiving peer will be presented the username 'vsl' along with the password 'RdQ-rsU-p3E-632'. If these credentials match the receiving peer's expectations, then the link connection will be allowed, and denied otherwise. A Lynxari server running in secure mode (the default), should not attempt to link to a server running unsecure, since the proper two-way communications will not succeed due to access restrictions.

4.4 About Password Security

Usernames and passwords are stored in a file unencrypted. The system must store this sensitive security information somewhere in order to accept API requests and initiate peer links. If it were encrypted in some manner, then the passphrase to decrypt them would have to be stored somewhere, and this would not enhance security because then that passphrase would either be clear text or itself be encrypted. Pretty soon it's turtles all the way down! Any obfuscation or trickery would only increase confusion without increasing security, so this is the reason the passwords are stored in cleartext.

However, because there is no practical method to encrypt the secrets, it is imperative that all measures to keep them secret are not compromised. The file permissions should be such that only the platform account owner has access. All operating system, software, hardware, and physical security measures should be maintained so that unauthorized individuals cannot gain access to the sensitive files.

5. Device Driver Configuration

Section 2.2.1 covered the device list configuration file, `'devlist.json'`, used to tell the Lynxari server what device drivers to load, and the location of the driver module. In addition to that information, a device definition may also contain an object, listing options to be passed directly to the driver module. Device drivers may accept many various parameters, arguments, and other objects, but there are several certain parameters basic to the Lynxari system.

Two key:value pairs which are always available are 'streamPeriod' and 'devicePoll'. These are both time periods in milliseconds, and define the time between updated values being streamed and polled. There is a subtle difference between streaming and polling. With streaming, the device values are published on the stream each and every streamPeriod, regardless of whether or not the value has changed. The devicePoll period defines the time between polling of device sensors. Only if the value has changed between polling periods is the new value published—otherwise no change is registered. Therefore, streaming is a constant heartbeat which keeps ticking on the streamPeriod even if the value never changes, while polling checks the sensor to see if the value has changed and only updates if it has.

Another basic parameter is `'deltaPercent'` which is used to determine whether the value has changed enough to be published as "new". deltaPercent is the percentage difference that the value must show in order to be considered changed. For example, if the value in question is currently at 61.4 and has a deltaPercent of 1, then the value must change at least .614, plus or minus to be published on the device. Note that a streamPeriod less than devicePoll for most devices does not result in more rapidly changing stream values, since a physical hardware device must be polled in order to update the value.

Closely associated with deltaPercent is the concept of range. Any device value may have an optionally defined range. This helps determine the desired action for deltaPercent. Consider an example of temperature with the current value being 26.0. deltaPercent is set to be 0.5 with the desire that the temperature value be updated if the value changes ±½ degree. Without a concept of range though, this value must change by ±1.3 in order to be considered new, which is not what is desired. The answer is to define a parameter, `'temperature_range'` set to 100. Any device value may define this range by simply providing a key:value pair where the key is the value name concatenated with `'_range'` and the numeric range as the value.

Here is an example of a devlist.json config file which lists two devices, the first being an ultrasonic anemometer from RM Young, and the second being a precision barometer from Bosch.

```
{
  "devices":[
    {
      "name":"BMP183",
      "module":"@agilatech/lynxari-bmp183-device",
      "options":{
        "bus":"/dev/spidev1.0",
        "altitude":1750,
        "mode":3,
        "streamPeriod”:300000,
        "devicePoll":60000,
        "pressure_range": 215,
        "temperature_range" : 80,
        "deltaPercent":0.1
      }
    },
    {
      "name":"RMY85000",
      "module":"@agilatech/lynxari-rmy85000-device",
      "options":{
        "streamPeriod": 60000,
        "devicePoll": 1000,
        "deltaPercent": 0.5,
        "file":"/dev/ttyO0",
        "speed_range":40,
        "direction_range":360,
        "avg_speed_range":40,
        "avg_direction_range":360,
        "gust_speed_range":40,
        "gust_direction_range":360
      }
    }
  ]
}
```

Notice that both device options objects contain multiple `_range` definitions, corresponding to the various device values, and the the BMP183 device also defines additional parameters beyond just the basic ones discussed here.

Every device drive offers different capabilities, and may require additional configuration in addition or in place of the devlist.json options. A well-written driver will include some documentation with instructions covering installation, configuration, and usage.

Because Lynxari embraces node.js and the node package manager (npm), it adheres to npm conventions including module location and naming. That is why all device driver modules will be found within the `'node_modules'` directory, and then further in a directory the same as the module's name. Inside the module directory is where documentation, config files, samples, and other helpful resources will likely reside.

6. Application Configuration

Section 2.2.2 covered the application list configuration file, `'applist.json'`, used to tell the Lynxari server what application modules to load, and the location of the modules. In addition to that information, an application may also store its own config files and other resources in the module directory. Look in the `'node_modules'` directory inside the Lynxari platform directory, and then into a directory with the module name for additional documentation and resources for the specific application.

Although Agilatech cannot guarantee the quality, completeness, and documentation of 3rd party applications, we can promise that all applications published by Agilatech will adhere to platform policies, which includes a config.json configuration file and a README documentation file.

7. Connecting with peers

Lynxari offers the ability to link to other peer servers through a secure Web Sockets tunnel. This makes it easy to distribute devices, applications, and APIs across disparate networks and machines and have them all work together. It also eliminates the hassle and overhead of connecting and reconnecting to HTTP APIs. An application running on a Lynxari server in the cloud can query a device running on a Lynxari server on the edge with the same ease as if it were running locally. In addition, peer connections will automatically reconnect in the case of a network outage. All the functionality is built-in to Lynxari, requiring minimal configuration.

A good deal of the configuration involved with linking to peers has been covered in sections 2.2.3 and 3.3. Peer linking needs to integrate configuration and security, so these subjects will be covered together here. To make the nomenclature unambiguous, we'll refer to the Lynxari server initiating the peer connection as the local server, and the Lynxari server accepting the peer connection as the remote server.

For starters, either secure or unsecure linking to a remote peer requires an entry in the `'peerlist.json'` config file, as covered in section 2.2.3. If the host is correct in the peerlist entry, and the remote server is up and running, then the local server will initiate the peer link connection. If the remote server does not require authentication (a practice we discourage), then no other configuration is required. If the remote server does require credentials, this is when an entry in the peers section of `'auth/credentials/access.json'` file must appear as detailed in section 3.3.

When the local server is using TLS but with a self-signed certificate, the first peer connection attempt will fail, since only after an attempt can the remote server see that the public certificate was self-signed and respond with the appropriate error. Upon receiving this connection error, the local server responds with a second connection attempt, but this time also supplies the certificate signing certificate as discussed in section 3.1. If the local server's log is viewed for this exchange, a normal flow will look something like this:

```
Jul-24-2017 18:10:17 [peer-client] Peer connection error (wss://remote.io:1107/peers/
local): Error: self signed certificate in certificate chain
Jul-24-2017 18:10:18 [peer-client] Peer connection attempt using supplied cert for
(remote.io)
Jul-24-2017 18:10:19 [peer-client] WebSocket to peer established (wss://remote.io:1107/
peers/local)
Jul-24-2017 18:10:19 [peer-client] Peer connection established (wss://remote.io:1107/
peers/local)
```

This shows a successful connection has been established after the original rejection due to a self-signed certificate.

If there were no credentials defined for the remote server, then the log would show the following error:

```
Jul-24-2017 18:20:08 [http_server] No credentials found for link host remote.io:1107
```

If incorrect credentials are defined for the remote server, then the log would show the unauthorized result error:

```
Jul-24-2017 18:40:24 [peer-client] Peer connection error (wss://remote.io:1107/peers/
local): server returned 401
```

The log entry which reads, "WebSocket to peer established..." indicates that not only has the TLS handshake succeeded, but that the peer connection credentials have been accepted and the connection has been upgraded to a secure web socket.

In summary, to accept peer connections:

1. Define a username:password in the file `'auth/credentials/access.json'`.
2. Ensure that any peer wishing to connect uses the correct username:password combination.
3. Enable TLS by providing the proper certificates.
   a. Supply a CA certificate if the public certificate is self-signed.

To initiate peer connections:

1. Define remote peers in `'peerlist.json'` config file.
2. For each remote host to peer to, define a host object in the peers section of the file `'auth/credentials/access.json'`.
   a. Insure the username is correct for the host.
   b. Insure the password is correct for the host.
3. Enable TLS by providing the proper certificates.
   a. Supply a CA certificate if the public certificate is self-signed.

Appendix

Creating self-signed certificates

There are many methods for creating self-signed certificates. Agilatech has developed tools to make the task a bit easier. The tools for creating signing certificates and operational server certificates are found in the Lynxari platform directory under `'toolbox/ca'`. This scripts and configurations should work on all Linux-based operating systems, including macOS. The only requirement is that openssl be installed.

Note: The configuration files and scripts in the ca directory will create certificates, directories, indices, and text files intended to support creation of many self-signed domain certificates. It is advisable to copy the entire directory to some administrative machine and create all certificates for all machines in one place.

The first step is to create root and intermediate CA signing certificates. For this example, we'll assume the ca directory has been moved to `'/home/agt'`. cd to that directory and run the script `'./createCA.sh'`. This script creates several subdirectories and files, and then uses the openssl program to generate a root signing certificate.

This program will first ask for a pass phrase for the **root** private key:

```
Enter pass phrase for private/ca.key.pem
```

Just to make sure you didn't make a mistake when entering the pass phrase, it will ask you to verify the pass phrase. Enter the same pass phrase again for verification:

```
Verifying - Enter pass phrase for private/ca.key.pem:
```

The script will then create the private root CA key.

```
#
# Creating Root certificate
#
```

You will be asked to provide the **root** pass phrase in order to create the private key:

```
Enter pass phrase for private/ca.key.pem:
```

The script now asks for information to be incorporated into the Distinguished Name:

```
You are about to be asked to enter information that will be incorporated
into your certificate request.
What you are about to enter is what is called a Distinguished Name or a
DN.
There are quite a few fields but you can leave some blank
For some fields there will be a default value,
If you enter '.', the field will be left blank.
```

Six questions will be asked about your location and organization to provide a unique name for the certificate, starting with country code:

```
Country Name (2 letter code) []:
```

Enter your ISO ALPHA-2 country code (i.e. US for United States, AU for Australia, etc.)

```
State or Province Name []:
```

Enter a complete name or abbreviation of your state or province (i.e. Queensland, etc.)

```
Locality Name []:
```

This may be left blank, or enter your city or other locality.

```
Organization Name []:
```

Enter the name of your company or organization.

```
Organizational Unit Name []:
```

This may be left blank, or enter your business unit name (i.e. Engineering, Production, etc.)

```
Common Name []:
```

For Web certificates, the Common Name is typically the domain name, but this will be used for a root authority, so enter a text field which identifies your company name and the fact that this is a root certificate (i.e. ABC Company Root CA).

```
Email Address []:
```

This field may be left blank, or enter a contact email address.

The script will then created and verify the root certificate. The script will print out information about the root certificate. The next step after root key verification is to create a similar intermediate key using the root key. The intermediate CA signing key is the one which will be used to sign domain certificates.

Just like with the root key creation, you will be asked for a pass phrase for the **intermediate** key. For good security, this should be different from the root pass phrase, but does not absolutely have to be:

```
Enter pass phrase for /home/agt/ca/intermediate/private/
intermediate.key.pem
```

And once again verify the **intermediate** pass phrase:

```
Verifying - Enter pass phrase for /home/agt/ca/intermediate/private/
intermediate.key.pem
```

A intermediate certificate signing request must be created.

```
#
# Creating Intermediate CSR
#
```

The intermediate key is used to create the intermediate CSR, so enter the pass phrase for the **intermediate** key:

```
Enter pass phrase for /home/agt/ca/intermediate/private/
intermediate.key.pem
```

Just as with the root key, the intermediate key requests information for the distinguished name. The values entered are required to be the same, *except* for the Common Name. You **must** enter a different value for the *intermediate* certificate Common Name, and should identify your company name and the fact that this is a intermediate certificate (i.e. ABC Company Intermediate CA).

The root CA is used to create the intermediate signing certificate, so now you must enter the pass phrase to the **root** key (*not* the one just provided for the intermediate key):

```
Enter pass phrase for /home/agt/ca/private/ca.key.pem:
```

After checking The program will sign the intermediate CA signing certificate using the root key as long as you answer affirmatively:

```
Certificate is to be certified until Jul 16 14:43:33 2027 GMT (3650 days)
Sign the certificate? [y/n]:
```

And commit this certificate.

```
1 out of 1 certificate requests certified, commit? [y/n]
```

This first program has created certificate authority (CA) signing certificates. At this point, there are no domain/machine certificates created. The 'createCert.sh' script in the intermediate directory will create domain/machine certificates.

cd to the `intermediate` directory and run the `'createCert.sh'` script. It requires one argument, which is the hostname of the server. If your machines are on a private intranet without fully qualified hostnames, you can use their I.P. addresses (i.e. createCert.sh 10.0.124.160).

Once again, information for creation of the Distinguished Name needs to be collected. The values entered are required to be the same, *except* for the Common Name. Since you are creating a server certificate, in this case the common name should be the hostname, or IP address (the same as given as the argument to the script).

The server certificate is being signed by the intermediate certificate authority, so you must provide the passphrase used for the creation of the *intermediate* certificate:

```
Enter pass phrase for /home/agt/ca/intermediate/private/
intermediate.key.pem:
```

The machine certificate is created, and the details are presented. Check over the fields in the Subject to see that they look correct. Notice also that this certificate expires in one year and ten days.

Type 'y' to sign the certificate.

```
Sign the certificate? [y/n]
```

And then again to commit this certificate.

```
1 out of 1 certificate requests certified, commit? [y/n]
```

The result of this script is two certificates: the private key certificate saved in the private directory, and the public key certificate saved in the certs directory. In the earlier step, a CA signing authority certificate was created. All three of these certificates are necessary to use TLS encryption on your Lynxari server. See section 3.1 for instructions about installing the certificates.

You may continue to use this script over and over to create as many machine certificates as needed. Every time a new set of certificates is signed and created, the serial number is incremented, and the cert is noted in an file named 'index.txt'. This is a simple database of the certificates.

Steps should be taken to protect this ca directory, as it contains private keys--notably the root signing private key which could be used by an attacker to compromise your systems. Insure that directory and file permissions are set to prevent unauthorized access.

In this example, the certificates required to provide TLS encryption on the machine at 10.0.124.160 are located:

Private Key:	/home/agt/ca/intermediate/private/10.0.124.160.key.pem
Certificate Authority Certificate:	/home/agt/ca/intermediate/certs/ca-chain.cert.pem
Public Key Certificate:	/home/agt/ca/intermediate/certs/10.0.124.160.cert.pem