TellStick ZNet Documentation

Release v1.3.0

Telldus Technologies

Nov 12, 2019

Getting started

TellStick ZNet allows developers to build own plugins and scripts run the device to extend the functionality with features not yet supported by Telldus.

It is also possible to alter the behaviour on how TellStick ZNet should interpret signals and messages from devices.

CHAPTER 1

Intro

TellStick ZNet offers two ways of integrating custom scripts. They can be written in either Python or Lua. The difference is outlined below.

1.1 Python

Python plugins are only available for TellStick ZNet Pro. Python plugins cannot be run on TellStick ZNet Lite. Python plugins offers the most flexible solution since full access to the service is exposed. This also makes it fragile since Python plugins can affect the service negative.

1.2 Lua

Lua code is available on both TellStick ZNet Pro and TellStick ZNet Lite. Lua code runs in a sandbox and has only limited access to the system.

To create a Lua script you need to access the local web server in TellStick ZNet. Browse to: http://{[}ipaddress{]}/lua to access the editor.

Lua codes works by signals from the server triggers the execution.

1.2.1 Installation

This is the software running on the late generation TellStick ZNet Lite and TellStick Net

Prerequisites

This software is only supported under Linux and macOS.

Although most of the prerequisites can be installed sandboxed, locally in the project folder, there are some libraries that must exist on the system.

- **The following applications must be available:**
  - python (2.7)
  - virtualenv
  - Node.js

Clone the Server code from here: https://github.com/telldus/tellstick-server

Linux

In many Linux distributions the packages for *python* and *virtualenv* already exist. On a Debian/Ubuntu based system these can be installed using this command:

```
sudo apt-get install python virtualenv
```

If *virtualenv* is not available on your system, you can install it with either *pip* or *easy install*:

```
sudo pip install virtualenv
```

or

```
sudo easy_install virtualenv
```

macOS

Pyhon is already shipped on macOS. You only need to install *virtualenv* using:

```
sudo easy_install virtualenv
```

Setup

To setup the source and prepare the base plugins run the following script:

```
./tellstick.sh setup
```

This vill create a *virtualenv* folder under the *build* folder and download and install the required libraries for running the software. The installation is completely sandboxed and nothing is installed in the system. You can wipe the *build* folder at any time to remove any installed packages.

Running

Start the server by issuing:

```
./tellstick.sh run
```

By default the server will restart itself any time it detects a file has been modified.

Check out and follow the instructions on getting the server software running on a computer here: https://github.com/telldus/tellstick-server

After installation the tellstick server is installed without any plugins. For development the lua-plugin is a recommended plugin to install. Install it with:

```
./tellstick.sh install lua
```

1.2.2 Overview

1.2.3 API Documentation

1.2.4 Python plugins

Python plugins offers the most flexible way of extending the functionality of TellStick. To get started a development environment should first be setup on a computer running Linux or macOS. Windows is not supported at the moment.

Telldus own plugins are open source and can be used as a base for new plugins. These can be found here: https://github.com/telldus/tellstick-server-plugins

This guide will describe the example plugin found here: https://github.com/telldus/tellstick-server-plugins/tree/master/templates/device

The plugin adds one dummy device to the system.

During the development it is recommended to install it within the server software. This way the software will restart itself whenever a file has changed. To install it use the tellstick command `install`:

```
./tellstick.sh install [path-to-plugin]
```

Replace *[path-to-plugin]* with the path to the plugin root folder.

Anatomy of a plugin

TellStick plugins are packaged as python eggs combined in a zip file. The eggs are signed with a pgp signature.

The metadata for a plugin is described in the file setup.py. This is a standard setuptools file with a couple custom configurations added.

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-

try:
      from setuptools import setup
except ImportError:
      from distutils.core import setup

setup(
      name='Dummy device',
      version='1.0',
      author='Alice',
      author_email='alice@wonderland.lit',
      category='appliances',
      color='#2c3e50',
      description='Dummy template for implementings device plugins',
      icon='dummy.png',
      long_description="""
              This plugin is used as a template when creating plugins that support␣
↪new device types.
      """,
      packages=['dummy'],
```

(continues on next page)

(continued from previous page)

```
      package_dir = {'':'src'},
      entry_points={ \
              'telldus.startup': ['c = dummy:Dummy [cREQ]']
      },
      extras_require = dict(cREQ = 'Base>=0.1\nTelldus>=0.1'),
)
```

Most of the fields can be found in the setuptools documentation.

author The name of the developer of the plugin. This name must match the pgp signin certificate.

author_email The email of the developer of the plugin. This must match the pgp singning certificate.

category This must be one of:

- security
- weather
- climate
- energy
- appliances
- multimedia
- notifications

color A color used in plugin selection user interface in the format #000000.

compatible_platforms Reserved for future use.

description A short description of the plugins. This should only be one line.

entry_points TellStick plugins can be loaded by one of two entry points.

telldus.startup This plugin will auto load on startup. Use this when it is important that the plugin is always loaded.

telldus.plugins This plugin will be loaded on-demand. This speeds up loading times and keep the memory footprint to a minimum.

icon Filename of icon in size 96x96.

long_description A long description describing the plugin. Markdown can be used.

name The name of the plugin.

packages A list of python packages included in the plugin. This should match the folder structure of the files. Please see setuptools documentation for more information.

required_features Reserved for future use.

version The version of the plugin.

Building a deployable plugin

Once development is finished it's time to package the code into a deployable package. Before this command a working pgp code signing key must be setup on the computer. The name and email must match the metadata `author` and `author_email` specified in setup.py.

Setting up a key

You can safely skip this step if you already have a pgp-key setup on your computer.

```
gpg --gen-key
```

This will take you through a few questions that will configure your keys.

```
Please select what kind of key you want: (1) RSA and RSA (default)
What keysize do you want? 4096
Key is valid for? 0
Is this correct? y
Real name: Enter the same name as in setup.py
Email address: Enter the same email as in setup.py
Comment:
Change (N)ame, (C)omment, (E)mail or (O)kay/(Q)uit? O
Enter passphrase: Enter a secure passphrase here (upper & lower case, digits, symbols)
```

Build the plugin

To build the package use the `build-plugin` command to tellstick.sh

```
./tellstick.sh build-plugin [path-to-plugin]
```

Replace *[path-to-plugin]* with the path to the plugin root folder. During building the plugin will be signed using your pgp key and if a passphrase has been setup you will be asked for your password.

This will build a .zip file ready to be uploaded to a TellStick.

1.2.5 Concepts

Read more about the concepts used for building plugins

Methods, actions and sensor values

The core object in tellstick-server is the `Device` object.

Device vs sensor?

In Telldus Live! devices and sensors are separated and two separate things. In tellstick-server these are one object and all inherits `telldus.Device`.

There is a convenience class `telldus.Sensor` that sets some initial parameters for devices that only supports sensor-values.

Methods

TellStick can control many different types of devices that support different features. For example, a bell does not support the on-signal and not all lamp switches support dimming.

To determine which methods a device supports, call the function `methods()` on the device.

See the following example to determine if a device supports on and off:

Listing 1: Python example:

```
methods = device.methods()
if methods & Device.TURNON:
  logging.debug('The device supports turning on')
if methods & Device.TURNON:
  logging.debug('The device supports turning off')
if methods & Device.DIM:
  logging.debug('The device is dimmable')
```

Listing 2: Lua example:

```
ON = 1
OFF = 2
DIM = 16

local methods = device:methods()
if (BitAND(methods, ON) == ON) then
  print("The device supports turning on")
end
if (BitAND(methods, OFF) == OFF) then
  print("The device supports turning off")
end
if (BitAND(methods, DIM) == DIM) then
  print("The device is dimmable")
end

-- The lua version shipped with TellStick does not support bitwise operators
function BitAND(a,b)--Bitwise and
  local p,c=1,0
  while a>0 and b>0 do
    local ra,rb=a%2,b%2
    if ra+rb>1 then c=c+p end
    a,b,p=(a-ra)/2,(b-rb)/2,p*2
  end
  return c
end
```

Observer pattern

Inter plugin communication is based on an observer pattern. Any plugin that wishes to publish events or pull information from other plugins may register an interface to observe.

Both observers and publishers must inherit from the `Plugin` base class.

Publish the observer interface

An observer interface is a class that inherits `IInterface` and contains only static functions. Example:

```
class IExampleObserver(IInterface):
      """Example observer interface exporting two functions"""
      def foo():
              """Example function no 1"""
```

(continues on next page)

(continued from previous page)

```
	def bar():
		"""Example function no 2"""
```

The class that uses this creates and observer collection.

```
class ExamplePlugin(Plugin):
	observers = ObserverCollection(IExampleObserver)
```

Multiple plugins may observe the interface but only one plugin may create the observer collection. Calling the observers can be made all by one och one by one by iterating the collection. The calling the whole collection at once the returned value by each observer will be discarded. It the return value is needed you must iterate and call each observer individually.

```
def callAllTheFoes(self):
	self.observers.foo()  # Note, the returned value cannot be used

def enterTheBars(self):
	for observer in self.observers:
		retVal = observer.bar()
		if retVal is True:
			logging.info("This bar is awesome")
```

Full example

```
class IExampleObserver(IInterface):
	"""Example observer interface exporting two functions"""
	def foo():
		"""Example function no 1"""
	def bar():
		"""Example function no 2"""

class ExamplePlugin(Plugin):
	observers = ObserverCollection(IExampleObserver)
```

Implementing the observer

To observe an interface the observing plugin must mask itself that it observes the interface by using the `implements()` function.

```
class ExampleObserverPlugin(Plugin):
	implements(IExampleObserver)
```

It can then implement the interface functions. Note that it's not necessary to implement all the functions from the interface.

```
class ExamplePlugin(Plugin):
	implements(IExampleObserver)

	def bar(self):
		return self.isThisBarAwseome
```

Signals and slots

Signals and slots is another way of inter plugin communication. It is built on the *observers framework* but offers a more loosley coupled integration. It's benefit against observers is:

- The observers (slots) does not need to import the interface from the publishers (signal). This allows the plugin to be loaded even when the publishing plugin is not available.
- The slots can be named freely so name collision is not as likley to occur.

The drawbacks instead are:

- The publishers does not know anything about the observers so it's not possible the return a value from the observers. If a return value is required then use *observers* instead.

Signals

To trigger an event or send some data to an observer a Signal is used. A signal is created by using the decorator *@signal*.

Example:

```
from base import Plugin, signal

class MyPlugin(Plugin):
  @signal('mysignal')
  def foobar(self):
    # Every time this function is called the signal "mysignal" is fired
    logging.warning("mysignal is fired")

  @signal
  def hello(self):
      # This signal takes the function name as the signal name
      logging.warning("signal hello is fired")
```

Slots

To receive a signal a plugin declares a slot. The plugin must implement *base.ISignalObserver* to be able to receive any signals. Then use the decorator *@slot* on the function you wish to be called when the signal is fired.

Example:

```
from base import implements, ISignalObserver, Plugin, slot

class AnotherPlugin(Plugin):
  implements(ISignalObserver)

  @slot('mysignal')
  def foobar(self):
    logging.warning('Slot foobar was triggered by the signal "mysignal"')
```

1.2.6 Examples

Read more about the concepts used for building plugins

Configurations for plugins

If your plugin needs user set configuration values it can add this by using configurations.

Wrap your pluginclass with the *@configuration* decorator.

```
from base import configuration, ConfigurationString, Plugin, ConfigurationList

@configuration(
  companyName = ConfigurationString(
      defaultValue='Telldus Technologies',
      title='Company Name',
      description='Name of the Company'
  ),
  contacts = ConfigurationList(
      defaultValue=['9879879879', '8529513579', '2619859867'],
      title='company contacts',
      description='Contacts of the company',
      minLength=2
  ),
  username = ConfigurationString(
      defaultValue='admin@gmail.com',
      title='Username',
      description='Username of the company Administrator'
  ),
  password = ConfigurationString(
      defaultValue='admin1234',
      title='Password',
      description='Password of the company Administrator',
      minLength=8,
      maxLength=12
  )
)
class Config(Plugin):
  def companyInfo(self):
      # access the companyName information from the configuration
      return {
          'companyName' : self.config('companyName'),
          'contacts' : self.config('contacts'),
          'username' : self.config('username'),
          'password' : self.config('password'),
      }
```

Here, the configuration store company information and return it when it called.

The configuration has following classes:

base.ConfigurationString: this function use to store configuration value as a string.

base.ConfigurationNumber: this function use to store configuration value as a number.

base.ConfigurationList: this function use to store configuration value as a list.

base.ConfigurationDict: this function use to store configuration value as a dictionary.

Call configuration to get company information using lua script :

```
local companyObject = require "configuration.Config"
local companyData = companyObject:companyInfo()
```

Sensor plugin development

This example shows how to extend the server with a new type of sensor.

Prepare the Sensor

In order to create the sensor, you have to import `Sensor` class form the `telldus` package and extend it in your TemperatureSensor class

```
from base import Plugin, Application
from telldus import DeviceManager, Sensor

class TemperatureSensor(Sensor):
  def __init__(self):
    super(TemperatureSensor,self).__init__()

  ...
  ...
  ...
```

Export functions

All sensors exported must subclass Sensor

Minimal function to reimplement is : `localId` and `typeString`

```
def localId(self):
  '''Return a unique id number for this sensor. The id should not
  be globally unique but only unique for this sensor type.
  '''
  return 1

def typeString(self):
  '''Return the sensor type. Only one plugin at a time may export sensors using
  the same typestring'''
  return 'temperaturesensor'
```

Add Sensor

To add a sensor into plugin `Temperature`:

```
class Temperature(Plugin):
  '''This is the plugins main entry point and is a singleton
  Manage and load the plugins here
  '''
  def __init__(self):
    # The devicemanager is a globally manager handling all device types
    self.deviceManager = DeviceManager(self.context)

    # Load all devices this plugin handles here. Individual settings for the devices
    # are handled by the devicemanager
    self.deviceManager.addDevice(TemperatureSensor())
```

(continues on next page)

(continued from previous page)

```
    # When all devices has been loaded we need to call finishedLoading() to tell
    # the manager we are finished. This clears old devices and caches
    self.deviceManager.finishedLoading('temperaturesensor')
```

A complete example for Temperature sensor

```
# -*- coding: utf-8 -*-

from base import Application, Plugin
from telldus import DeviceManager, Sensor

class TemperatureSensor(Sensor):
  '''All sensors exported must subclass Sensor

  Minimal function to reimplement is:
  localId
  typeString
  '''
  @staticmethod
  def localId():
    '''Return a unique id number for this sensor. The id should not be
    globally unique but only unique for this sensor type.
    '''
    return 2

  @staticmethod
  def typeString():
    '''Return the sensor type. Only one plugin at a time may export sensors using
    the same typestring'''
    return 'temperature'

  def updateValue(self):
    """setTempratureSensor value constantly."""
    # This is dummy data
    self.setSensorValue(Sensor.TEMPERATURE, 35, Sensor.SCALE_TEMPERATURE_CELCIUS)

class Temperature(Plugin):
  '''This is the plugins main entry point and is a singleton
  Manage and load the plugins here
  '''
  def __init__(self):
    # The devicemanager is a globally manager handling all device types
    self.deviceManager = DeviceManager(self.context)

    # Load all devices this plugin handles here. Individual settings for the devices
    # are handled by the devicemanager
    self.sensor = TemperatureSensor()
    self.deviceManager.addDevice(self.sensor)

    # When all devices has been loaded we need to call finishedLoading() to tell
    # the manager we are finished. This clears old devices and caches
    self.deviceManager.finishedLoading('temperature')

    Application().registerScheduledTask(self.updateValues, minutes=1, runAtOnce=True)
```

(continues on next page)

(continued from previous page)

```
  def updateValues(self):
    self.sensor.updateValue()
```

Web interface

This example shows how a plugin can extend the local webinterface with its own page in the menu.

Base files

Before making any plugin we have to create a setup file for the plugin. For more about *setup.py*.

Add following code in setup.py:

```
try:
  from setuptools import setup
  from setuptools.command.install import install
except ImportError:
  from distutils.core import setup
  from distutils.command.install import install
import os

class buildweb(install):
  def run(self):
    print("generate web application")
    os.system('npm install')
    os.system('npm run build')
    install.run(self)

setup(
  name='Welcome',
  version='0.1',
  packages=['welcome'],
  package_dir = {'':'src'},
  cmdclass={'install': buildweb},  #Call the fuction buildweb
  entry_points={ \
  'telldus.plugins': ['c = welcome:Welcome [cREQ]']
  },
  extras_require = dict(cREQ = 'Base>=0.1\nTelldus>=0.1\nTelldusWeb>=0.1')
)
```

Now create plugin file :

For Interacting with UI import interface `IWebReactHandler` from `telldus.web` and implements it into the plugin.

```
from base import Plugin, implements
from telldus.web import IWebReactHandler

class Welcome(Plugin):
  implements(IWebReactHandler)

  @staticmethod
  def getReactComponents():
```

(continues on next page)

(continued from previous page)

```
  return {
    'welcome': {
    'title': 'Welcome',
    'script': 'welcome/welcome.js',
    'tags': ['menu'],
  }
}
```

Here, the function `getReactComponents()` Return a list of components this plugin exports.

HTML interface

Run `npm init` command in the root folder, It will ask for the package details and fill it out.

It will create a package.json file in your root folder.

Configure package.json

Make following modifications in the package.json file.

```
{
  "name": "welcome",
  "version": "1.0.0",
  "scripts": {
    "build": "gulp",
    "watch": "gulp watch"
  },
  "devDependencies": {
    "babel-cli": "^6.18.0",
    "babel-preset-es2015": "^6.16.0",
    "babel-preset-react": "^6.16.0",
    "babel-preset-stage-0": "^6.16.0",
    "gulp": "^3.9.1",
    "gulp-babel": "^6.1.2",
    "gulp-cli": "^1.2.2",
    "gulp-requirejs-optimize": "^1.2.0",
    "requirejs": "^2.3.2"
  }
}
```

That are major dependencies to run and display UI.

Create gulpfile

Now create gulpfile.js in the root folder.

Gulp is a toolkit for automating painful or time-consuming tasks in your development workflow, so you can stop messing around and build something. more

Add following task in gulpfile:

```
var gulp = require('gulp');
var babel = require("gulp-babel");
```

(continues on next page)

(continued from previous page)

```
var requirejsOptimize = require('gulp-requirejs-optimize');

gulp.task('default', ['scripts'], function() {
});

gulp.task('jsx', function () {
  return gulp.src('src/welcome/app/**/*.jsx')
    .pipe(babel({
      presets: ['es2015', 'stage-0', 'react']
    }))
    .pipe(gulp.dest('src/welcome/build'));
});

gulp.task('js', function () {
  return gulp.src('src/welcome/app/**/*.js')
    .pipe(gulp.dest('src/welcome/build'));
});

gulp.task('scripts', ['jsx', 'js'], function () {
  return gulp.src('src/welcome/build/welcome/welcome.js')
    .pipe(requirejsOptimize({
      paths: {
        'react': 'empty:',
        'react-mdl': 'empty:',
        'react-router': 'empty:'
      },
      baseUrl: 'src/welcome/build',
      name: 'welcome/welcome'
    }))
    .pipe(gulp.dest('src/welcome/htdocs'));
});

gulp.task('watch', ['default'], function() {
  gulp.watch('src/welcome/app/**/*.jsx', ['default']);
});
```

Here, gulp task `jsx` will copy all file from the specified path and convert it and paste it into the destination path.

gulp task `js` and `script` will do same as `jsx`.

Create UI design

Now Design UI using react and give extension `.jsx` and save this file to the path that you have given in gulpfile.

```
define(
  ['react', 'react-mdl', 'react-router'],
  function(React, ReactMDL, ReactRouter) {
    class WelcomeApp extends React.Component {
      render() {
        return (
          <div>
            <h1>hello world!</h1>
          </div>
        );
      }
```

(continues on next page)

(continued from previous page)

```
    };

  return WelcomeApp;
  }
);
```

Now the plugin is ready to install and use.

1.2.7 API Documentation

This is where you can find API-level documentation for the various Python libraries available in TellStick firmware.

Module: base

Classes in the base module are only accessible from Python applications.

Application

class base.**Application**(*run=True*)

This is the main application object in the server. There can only be once instance of this object. The default constructor returns the instance of this object.

registerScheduledTask(*fn*, *seconds=0*, *minutes=0*, *hours=0*, *days=0*, *runAtOnce=False*, *strictInterval=False*, *args=None*, *kwargs=None*)

Register a semi regular scheduled task to run at a predefined interval. All calls will be made by the main thread.

Parameters

- **fn** (*func*) – The function to be called.
- **seconds** (*integer*) – The interval in seconds. Optional.
- **minutes** (*integer*) – The interval in minutes. Optional.
- **hours** (*integer*) – The interval in hours. Optional.
- **days** (*integer*) – The interval in days. Optional.
- **runAtOnce** (*bool*) – If the function should be called right away or wait one interval?
- **strictInterval** (*bool*) – Set this to True if the interval should be strict. That means if the interval is set to 60 seconds and it was run ater 65 seconds the next run will be in 55 seconds.
- **args** (*list*) – Any args to be supplied to the function. Supplied as *args.
- **kwargs** (*dict*) – Any keyworded args to be supplied to the function. Supplied as **kwargs.

Note: The interval in which this task is run is not exact and can be delayed one minute depending on the server load.

Note: Calls to this method are threadsafe.

static defaultContext()

Returns the default context used by the application

queue(*fn*, **args*, ***kwargs*)
Queue a function to be executed later. All tasks in this queue will be run by the main thread. This is a thread safe function and can safely be used to syncronize with the main thread

Returns True if the task was queued

Returns False if the server is shutting down

registerShutdown(*fn*)
Register shutdown method. The method fn will be called the the server shuts down. Use this to clean up resources on shutdown.

Parameters **fn** (*func*) – A function callback to call when the server shuts down

static signal(*msg*, **args*, ***kwargs*)
Send a global signal to registered slots. It is not recommended to call this method directly but instead use the signal decorator. Any extra parameters supplied will be forwarded to the slot.

Parameters **msg** (*str*) – The signal name

@base.**mainthread**
This decorator forces a method to be run in the main thread regardless of which thread calls the method.

Configurations

@base.**configuration**
This decorator should be applied on the Plugin class to add configuration values. Configuration values will be exposed automatically to the user.

Example:

```
@configuration(
  companyName = ConfigurationString(
    defaultValue='Telldus Technologies',
    title='Company Name',
    description='Name of the Company'
  ),
)
class MyPlugin(Plugin):
  pass
```

class base.**ConfigurationValue**(*valueType*, *defaultValue*, *writable=True*, *readable=True*, *hidden=False*, *sortOrder=0*)
Base class for configuration values. Do not use this class directly but use one of the subclasses instead.

Changed in version 1.2: Added parameter *sortOrder*

Parameters

- **valueType** (*str*) – The type of the configuration value. Only set this in subclasses..
- **defaultValue** (*str*) – The default value used if not value is set.
- **writable** (*bool*) – If this value can be set by the user in the UI.

- **readable** (*bool*) – *True* if the current value could be read by user interfaces. Set this to *False* for fields such as password where the current value should not be exposed to the UI.
- **hidden** (*bool*) – If this field should be user configurable in the UI or not.
- **sortOrder** (*int*) – The order the values should be sorted by in the UI.

class base.**ConfigurationBool**(*defaultValue=False*)
Configuration class used to store boolean values

New in version 1.2.

class base.**ConfigurationDict**(*defaultValue={}*)
Configuration class used to store dictionaries

class base.**ConfigurationNumber**(*defaultValue=0*)
Configuration class used to store numbers

class base.**ConfigurationList**(*defaultValue=[]*)
Configuration class used to store lists

class base.**ConfigurationSelect**(*defaultValue=''*, *options={}*)
Configuration class used to store one value out of a predefined selection

New in version 1.2.

class base.**ConfigurationString**(*defaultValue=''*, *minLength=0*, *maxLength=0*)
Configuration class used to store strings

Signals & Slots

class base.**ISignalObserver**
Bases: base.Plugin.IInterface

Implement this IInterface to recieve signals using the decorator *@slot*

@base.**signal**
This is a decorator for sending signals. Decorate any of your Plugins methods with this decorator and whenever the method is called the signal will be fired.

Parameters **name** (*str*) – The signal name. This can be omitted and then the function name will be used as the name of the signal.

@base.**slot**(*message = ''*)
This is a decorator for receiveing signals. The class must implement *base.ISignalObserver*

Parameters **message** (*str*) – This is the signal name to receive

Module: scheduler

Module: telldus

1.2.8 Intro

TellStick ZNet has a local REST interface to integrate into third party applications not running on the TellStick ZNet itself

A list of all available functions can be browsed on the device itself. Browse to: http://{[}ipaddress{]}/api to list the functions.

1.2.9 Authentication

Before making any REST calls to TellStick ZNet the application must request a token that the user has authenticated.

Step 1 - Request a request token

Request a request token by performing a PUT call to the endpoint /api/token. You need to supply the application name as a parameter "app"

```
$ curl -i -d app="Example app" -X PUT http://0.0.0.0/api/token
HTTP/1.1 200 OK
Date: Fri, 15 Jan 2016 13:33:54 GMT
Content-Length: 148
Content-Type: text/html;charset=utf-8
Server: CherryPy/3.8.0

{
  "authUrl": "http://0.0.0.0/api/authorize?token=0996b21ee3f74d2b99568d8207a8add9",
  "token": "0996b21ee3f74d2b99568d8207a8add9"
}
```

Step 2 - Authenticate the app

Redirect the user to the url returned in step 1 to let him/her authenticate the app.

Step 3 - Exchange the request token for an access token

When the user has authenticated the request token in step 2 the application needs to exchange this for an access token. The access token can be used in the API requests. To exchange the request token for an access token perform a GET call to the same endpoint in step 1. Supply the request token as the parameter "token".

```
$ curl -i -X GET http://0.0.0.0/api/token?token=0996b21ee3f74d2b99568d8207a8add9
HTTP/1.1 200 OK
Date: Fri, 15 Jan 2016 13:39:22 GMT
Content-Length: 230
Content-Type: text/html;charset=utf-8
Server: CherryPy/3.8.0

{
  "allowRenew": true,
  "expires": 1452951562,
  "token":
↪"eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCIsImF1ZCI6IkV4YW1wbGUgYXBwIiwiZXhwIjoxNDUyOTUxNTYyfQ.
↪eyJyZW5ldyI6dHJ1ZSwidHRsIjo4NjQwMH0.HeqoFM6-K5IuQa08Zr9HM9V2TKGRI9VxXlgdsutP7sg"
}
```

If the returned data contains allowRenew=true then the token was authorized to renew its expiration itself without letting the user authorize the app again. The application must renew the token before it expires or else the application must start the autorization again from step 1.

If allowRenew is not set to true it is not possible for the app to renew the token and it will always expire after the time set in the parameter "expires".

Step 4 - Making a request

To make a requst to a TellStick ZNet API endpoint the token in step 3 must be supplied as a bearer token in the header. This is an example requesting a list of devices:

```
$ curl -i -X GET http://0.0.0.0/api/devices/list?supportedMethods=3 -H
↪"Authorization: Bearer␣
↪eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCIsImF1ZCI6IkV4YW1wbGUgYXBwIiwiZXhwIjoxNDUyOTUxNTYyfQ.
↪eyJyZW5ldyI6dHJ1ZSwidHRsIjo4NjQwMH0.HeqoFM6-K5IuQa08Zr9HM9V2TKGRI9VxXlgdsutP7sg"
HTTP/1.1 200 OK
Date: Tue, 19 Jan 2016 10:21:29 GMT
Content-Type: Content-Type: application/json; charset=utf-8
Server: CherryPy/3.7.0

{
  "device": [
    {
      "id": 1,
      "methods": 3,
      "name": "Example device",
      "state": 2,
      "statevalue": "",
      "type": "device"
    }
  ]
}
```

Refreshing a token

If the user allowed the application to renew the token in steg 2 it can be renewed by the calling application. The token must be refreshed before it expires. If the token has expired the authentication must be restarted from step 1 again.

```
$ curl -i -X GET http://0.0.0.0/api/refreshToken -H "Authorization: Bearer␣
↪eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCIsImF1ZCI6IkV4YW1wbGUgYXBwIiwiZXhwIjoxNDUyOTUxNTYyfQ.
↪eyJyZW5ldyI6dHJ1ZSwidHRsIjo4NjQwMH0.HeqoFM6-K5IuQa08Zr9HM9V2TKGRI9VxXlgdsutP7sg"
HTTP/1.1 200 OK
Date: Tue, 19 Jan 2016 10:21:29 GMT
Content-Type: Content-Type: application/json; charset=utf-8
Server: CherryPy/3.7.0

{
  "expires": 1455295348,
  "token":
↪"eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCIsImF1ZCI6IkV4YW1wbGUgYXBwIiwiZXhwIjoxNDU1Mjk1MzQ4fQ.
↪eyJyZW5ldyI6dHJ1ZSwidHRsIjo4NjQwMH0.M4il4_2SqJwlCjmuXlU5DS6h-gX7493Tnk9oBJXbgPw"
}
```

The new token returned must be used from now on and the old be discarded.

1.2.10 Extending

It's possible to extend the API with new functions from *custom plugins*.

Prepare the plugin

In order for the API plugin to know about this plugin it must implement the interface `IApiCallHandler`

```
from api import IApiCallHandler
from base import Plugin, implements

class HelloWorld(Plugin):
      implements(IApiCallHandler)
```

Export a function

Use the decorator `@apicall` on the function you want to export. This example exports the function `helloworld/foobar`:

```
@apicall('helloworld', 'foobar')
def myfunction(self, arg1, arg2):
```

A complete example

```
from api import IApiCallHandler, apicall
from base import Plugin, implements

class HelloWorld(Plugin):
      implements(IApiCallHandler)

      @apicall('helloworld', 'foobar')
      def myfunction(self, arg1, arg2):
              """
              Docs for the function goes here
              """
              return True
```

Index