Providing Devices

Basics

Devices are added to pimatic by first providing a config for it. Every device needs an entry in the devices-Section of the `config.json` file

A device config looks like that:

```
{
  "id": "light",
  "class": "MySwitch",
  "name": "Kitchen Light"
}
```

You must first register your new Device by calling the `registerDeviceClass` function of the `deviceManager`. The framework reads every device config on startup and calls the provided `createCallback` function for all registered device of the registered class. The callback gets the device-config from the config file and should construct and return the instance of the device:

```
module.exports = (env) ->

  class MyPlugin extends env.plugins.Plugin

    init: (app, @framework, config) =>
      # ...

      deviceConfigDef = require("./device-config-schema")

      @framework.deviceManager.registerDeviceClass("MySwitch", {
        configDef: deviceConfigDef.MySwitch,
        createCallback: (config) => new MySwitch(config)
      })
```

```
  class MySwitch extends env.devices.PowerSwitch
    # ...

  myPlugin = new MyPlugin
  return myPlugin
```

Device-class

Any device must be subclass of the device class. A device has Attributes and Actions

Attributes

Attributes are properties of the device like the state of a switch or sensor values like temperature. For Sensor-Devices these are displayed at the frontend as values. For other devices the display type depends on the device type. For example the state of a switch is displayed as an switch button.

Attributes are defined by adding defining the `attributes` property of the `Device` class:

```
class MyDevice extends env.devices.Device

  attributes:
    temperature:
      description: "the messured temperature"
      type: Number
      unit: '°C'

  getTemperature: -> Promise.resolve(30)
```

Each attribute must have a getter function that returns a Promise fulfilled with the current attribute value. The getter must be named like the attribute prefixed with `get` and Uppercase first letter of the attribute name.

If an attribute changes an event with the attribute name should be emitted that contains the new value:

```
  inSomeFunctionOfMyPlugin: =>
    # Got a new value, so emit it to the framework:
```

```
    @emit "temperature", 20
```

For more attribute examples take a look at the predefined Devices.

Actions

In additional to attributes a device can have Actions. Actions are functions that can be called by the framework. Actions are defined like attributes:

```
class MyDevice extends env.devices.Device

  actions:
    turnOn:
      description: "turns the switch on"
    turnOff:
      description: "turns the switch off"
    changeStateTo:
      description: "changes the switch to on or off"
      params:
        state:
          type: Boolean

  # Returns a promise
  turnOn: -> @changeStateTo on

  # Retuns a promise
  turnOff: -> @changeStateTo off

  # Retuns a promise that is fulfilled when done.
  changeStateTo: (state) -> #...
view raw
```

For each Action there should be a function with the same name that executes the action and returns a promise which gets fulfilled when done.

For more actions examples take a look at the predefined Devices.

Constructor

Your constructor function must assign a name and id to the device (typical from the device

config).

```
class MyDevice extends env.devices.Device

    # ####constructor()
    # Your constructor function must assign a name and id to the device.
    constructor: (@config) ->
      @name = @config.name
      @id = @config.id
      # ...
      super()
```

Destroy Function

The destroy function is called when the device is deleted or modified. It must unregister event handlers and clear timers typically used for polling updates. Finally, the `super()` function must be called.

Predefined devices

If you don't want to define an exotic device try to use a predefined Device as base class.

Switch Device

For Switch devices you can use the `PowerSwitch` class as base and you have to define the `changeStateTo` and `getState` function.

```
class MySwitch extends env.devices.PowerSwitch

    # ####constructor()
    # Your constructor function must assign a name and id to the device.
    constructor: (@config) ->
      @name = @config.name
      @id = @config.id
      # ...
```

An full example for an switch device can be found in the with-switch-device branch

Temperature Sensor

A Temperature sensor can use the `TemperatureSensor` class as base. You must define a `getTemperature` function and emit the "temperature" attribute event, if a new value was read.

```
class YourTemperatureSensor extends env.devices.TemperatureSensor
  temperature: null

  constructor: (@config) ->
    @name = "some name or from config"
    @id = "some-id"
    # update the temperature every 5 seconds
    setInterval( ( =>
      @doYourStuff()
    ), 5000)

  doYourStuff: () ->
    # temperature = ...
    @temperature = temperature
    @emit "temperature", temperature

  getTemperature: -> Promise.resolve(@temperature)
```