Node JS Plugin Entwicklung

Voraussetzung für diese Anleitung ist das Express Server Plugin ab Version 0.0.3. Sei dir bitte darüber bewusst dass Dein Plugin von dem Express Plugin abhängig ist und dementsprechend bei dem Anwender installiert werden muss. Ein Prüfung auf das Vorhanden sein des plugins ist daher notwendig.

Der Port vom Express Server hat sich ab v0.0.3 geändert.

Plugin Generator benutzen

Das schöne am Generator ist, dass vieles der hier beschriebene Dinge bereits automatisiert ist wie z.b. Die Überprüfung ob das Express Plugin installiert ist, das rewrite für den Apache, und das installieren der `.htaccess` . Um ein neues Plugin zu erstellen kannst du einfach `npm init loxberry-plugin <ordner>` aufrufen und alle Fragen beantworten. Bei der Sprache musst du dann nur noch NodeJs auswählen.

Nach dem alles fertig ist, kannst du in den Ordner navigieren und direkt loslegen. Für den Entwicklungsserver kannst du einfach `npm run dev` eingeben und auf http://localhost:3300 dein Plugin entwickeln. Happy Coding

Wie funktioniert das Plugin

Die Idee des Plugins ist es einen ExpressJs Server bereitzustellen in dem du dich einbinden kannst. Um dies zu tun braucht dein Plugin eine `express.js` Datei im Ordner `webfrontend/htmlauth` oder `webfrontend/htmlauth/express`. Diese Datei wird automatisch vom Express Server gelesen und ausgeführt wenn eine Anfrage an den Express Server mit der entsprechenden url `/plugins/<pluginname>` ankommt. Da normalerweise Apache alle Anfragen bearbeitet muss dem Apache mitgeteilt werden, dass sich der Express Server darum kümmern soll. Dafür können wir die `mod_rewrite` Funktion des Apache Servers nutzen.
Der Express Server läuft auf Port 3300 und reagiert auf den Pfad `/plugins/<plugin name>` Der Name des Plugins ist der Ordnername der in der `plugin.cfg` Datei deines Plugins hinterlegt ist. Apache kann übergangen werden wenn der Request direkt an `http://<loxberry ip>:3300/plugin/<plugin name>` geschickt wird.

Die `express.js` Datei ist während der Laufzeit des Servers gecached. Jeder Server Neustart löscht den Cache. Ab v0.0.2 Werden Dateiänderungen nicht mehr automatisch erkannt. Das invalidieren des Caches kann per Post request ausgelöst werden. Dies sollte in der `postupgrade.sh` hinterlegt sein, dass die neuen Änderungen Anwendung finden.
`curl -X POST http://localhost:3300/system/express/plugin/<plugin_name>`

Das Plugin bietet auch die Möglichkeit den Express Server zu stoppen oder neu zu starten. Ausserdem können Logs gelesen werden und es werden Metriken angezeigt.

Der Express Server kommt mit der Handlebars Template Engine und das Loxberry Layout ist bereits automatisch implementiert. Die Nutzung von WebSockets ist mit dem Plugin ebenso einfach wie ein

normaler Http Call.

Express.js Handler

Um sich in den Express Server einzuklinken muss dein Plugin eine `express.js` Datei im Ordner `webfrontend/htmlauth` oder `webfrontend/htmlauth/express` bereitstellen und eine Funktion exportieren.

```
module.exports = ({router, static, logger, _, translate}) => {
  return router;
};
```

Parameters:

- *router*: Der Express Router um Routen oder URL Pfade die behandelt werden sollen
- *expressStatic*: Eine Referenz zu express.static ab v0.0.2. Vorher *static*
- *logger*: Eine logger Klasse die `info`, `debug`, `warn` und `error` methoden bereitstellt um Logausgaben zu ermöglichen. Mehr Details in den Logger Sektion.
- _: Die Lodash Bibliothek
- *translate*: Eine Funktion um im code Übersetzungen zu benutzen

Du kannst entscheiden welche Parameter benötigt werden und nur die angeben die benutz werden. Die Reihenfolge ist dabei egal. Wenn du nur den router brauchst dann benutzt du `module.exports = ({router}) => {...` und brauchst du lodash, der Router und den Logger dann kannst du dies mit `module.exports = ({router, logger, _}) => { ...` tun.

Definieren von Routen

Für das Definieren von Routen empfiehlt sich die Dokumentation von ExpressJS selbst. Hier ist alles so wie es auch in Express funktionieren würde.

Wichtig ist, dass der `router` am Ende der Funktion zurückgegeben wird.

```
module.exports = ({router}) => {
  router.get('/', (req, res) => {
    res.send('ok'); // for normal text content
    res.json({hello: 'world'}); // for json content
    res.render('index', {title: 'MyPluginTitle'}); // to use handlebars
template engine
  });
  return router;
};
```

Es können natürlich auch mehrere Routen zurückgegeben werden

```
module.exports = ({router}) => {
  router.get('/', (req, res) => {
    res.render('index', {title: 'MyPluginTitle'});
```

```
  });
  router.get('/hello/:name', (req, res) => {
    res.json({hello: req.params.name});
  });
  return router;
};
```

Alles was man auf Router Ebene mit Express gemacht werden kann, kannst du auch in dem Plugin machen.

Websockets

Die Websocket Implementierung ist eine eigenständige Entwicklung die von der Express-ws Bibliothek inspiriert ist. Für die Benutzung von Websockets muss `router.ws` anstelle von `router.get` aufgerufen werden. Die Argumente der Funktion unterscheiden sich minimal.

- ws: der Websocket Handler
- request: das Request Object
- next: die Standard Next Funktion von Express

```
const clients = [];
module.exports = ({router, logger}) => {
  router.ws('/foo', (ws, request, next) => {
    ws.on('open', () => clients.push(ws));
    ws.on('message', (message) => {
      logger.debug(`received message: ${message}`);
      ws.send(message.toString());
    });
  });

  return router;
};
```

Anhand der Url können auch mehrere WebSockets pro Plugin benutzt werden.

Rewrite Rules für Apache

Wie oben bereits beschrieben müssen wir dem Standard Apache Server mitteilen, dass für unser plugin der Express Server genutzt werden soll. Dafür brauchen wir eine `.htaccess` Datei im Ordner `webfrontend/htmlauth/express`. Gehen wir davon aus, dass wir ein Plugin namens “foobar” schreiben, dann wäre die Url zu unserem Plugin `/admin/plugins/foobar`. Standardmäßig wird die `index.cgi` in dem Ordner benutzt um Inhalte zu rendern die dann eine Existenzprüfung des Express Plugins vornimmt und an `express` weiterleitet (s.u.).

Wichtig ist, dass die Rewrite Regeln relative Pfade sind und erst ab `/admin/plugin/foobar/` funktionieren.

`.htaccess` Dateien müssen manuell vom Plugin installiert werden da diese beim Installationsprozess

ignoriert werden. Befolgen wir die Regeln vom “postinstall” dann kann man das mit einer einer Zeile umsetzen.
`cp webfrontend/htmlauth/express/.htaccess $ARGV5/webfrontend/htmlauth/plugins/$ARGV3/express/.htaccess`

```
RewriteEngine On # this is required
RewriteRule ^index.cgi http://localhost:3300/plugins/express [P,L] #the redirect
```

Wenn jegliche Anfrage umgeleitet werden soll

```
RewriteEngine On
RewriteRule ^index.cgi http://localhost:3300/plugins/express [P,L]
RewriteRule ^(.\*) http://localhost:3300/plugins/express/$1 [P,L]
```

Wenn ausschließlich `/admin/plugins/foobar/my-express-routes` umgeleitet werden soll.

```
RewriteEngine On
RewriteRule ^my-express-routes/(.\*)
http://localhost:3300/plugins/express/$1 [P,L]
```

Fehlerhinweis, wenn Express-Server nicht installiert ist

Dadurch dass vom Plugin eine Abhängigkeit zum Express Server besteht und der Plugin Content via Express ausgeliefert werden muss brauchen wir einen Mechanismus zu erkennen ob das Express Plugin installiert ist. Es kann durchaus vorkommen, dass ein Nutzer das Express Server Plugin noch nicht installiert hat oder deinstalliert. Leider gibt es während der Deinstallation keine Möglichkeit den Prozess zu stoppen wenn ein Plugin darauf aufbaut. Daher müssen wir dies in unserem Plugin abhandeln.

Die Idee ist, dass der Apache Server nur URL anfragen umleitet wenn die Plugin-Url `/admin/plugins/my_plugin/express` ist. Das Ermöglicht uns im Hauptordner deine Prüfung vorzunehmen ob das Express Plugin installiert ist. Wenn ja, wird eine Weiterleitung auf `/admin/plugins/my_plugin/express` vorgenommen. Falls nicht, wird eine Fehlerseite gerendert. Zusätzlich können wir die Version des Express Server abfragen sofern mindestens Version x benötigt wird.

`webfrontend/htmlauth/index.cgi`

```
#!/usr/bin/perl

require LoxBerry::Web;
use LoxBerry::System;
use CGI;

# This is to check if the express plugin is installed and in case it's not
# it will print an error with the hint that the unifi_presence plugin requires
# the express plugin.
```

```
my $requiredVersion = "0.0.3";
my $expressData = LoxBerry::System::plugindata("express");

if ($expressData && $expressData->{PLUGINDB_VERSION} ge $requiredVersion) {
    my $q = CGI->new;
    print $q->header(-status => 307, -location => 'express/');
    exit(0);
}

my $template = HTML::Template->new(
    filename => "$lbptemplatedir/error.html",
    global_vars => 1,
    loop_context_vars => 1,
    die_on_bad_params => 0,
);
$template->param( REQUIRED_VERSION => $requiredVersion);

%L = LoxBerry::System::readlanguage($template, "language.ini");
LoxBerry::Web::lbheader("Unifi Presence", "", "");
print $template->output();
LoxBerry::Web::lbfooter();
```

template/error.html

```
<h3><TMPL_VAR COMMON.MISSING_PLUGIN></h3>
<p style="color: red">
  <TMPL_VAR COMMON.REQUIRE_EXPRESS_1>
  <a
href="https://loxwiki.atlassian.net/wiki/spaces/LOXBERRY/pages/1673527328/Ex
press+Server
" target="_blank"><TMPL_VAR COMMON.EXPRESS_PLUGIN> (<TMPL_VAR
REQUIRED_VERSION>)</a>
  <TMPL_VAR COMMON.REQUIRE_EXPRESS_2>
<p>
```

template/lang/language_(de|en).ini

```
[COMMON]
MISSING_PLUGIN="Fehlendes Plugin"
REQUIRE_EXPRESS_1="Dieses Plugin benötigt das"
REQUIRE_EXPRESS_2="Plugin. Du kannst es entweder installieren oder das
Plugin deinstallieren."
EXPRESS_PLUGIN="Express Server"

//english
[COMMON]
MISSING_PLUGIN="Missing Plugin"
REQUIRE_EXPRESS_1="This plugin reguires the"
REQUIRE_EXPRESS_2="plugin. Either install the required plugin or uninstall
this one."
```

```
EXPRESS_PLUGIN="Express Server"
```

Handlebars Template Engine

Handlebars ist die Standard Template Engine für den Plugin Express Server und aktuell die einzige die vom Plugin angeboten wird. Die Template Dateien werden wie bei üblichen Plugins in den Ordner `templates` gelegt. Die Dateie-Endung muss `.hbs` sein.

index.hbs

```
<h1>This is my First Template</h1>
```

Um die Datei zu rendern muss in der `express.js` Datei die Funktion `res.render('index', {title: 'Test'})` benutz werden. Das Template wird dann im Loxberry Laut gerendert. Wenn du das Default Layout benutz solltest du mindestens den Titel definieren. Du kannst auch deine Eigene Seite definieren. Dazu muss `{layout: false}` angegeben werden. `res.render('index', {layout: false})`

Das Loxberry Layout hat 3 Variablen die benutz werden können:

- title: Der Seiten Titel im Header
- LB_helpLink: Eine Url für weitergehende Dokumentation
- LB_help: ein Template - nicht weiter geprüft

Diese Variablen sind equivalent zu `Loxberry::Web::lbheader($template_title, $helpurl, $helptemplate);`. Mehr dazu in der Dokumentation

Rendern mit Variablen

Wie die Syntax für Handlebars funktioniert kann der entsprechenden Dokumentation entnommen werden.

Hier ist ein kleines Beispiel

```
// templates/index.hbs
<h1>{{myTitle}}</h1>
<div>Hello {{name}}</div>

// express.js
res.render('index', {title:'My Page title', myTitle: 'Hello World Demo',
name: 'Foobar'});

// output wrapped in Layout
<h1>Hello World Demo<h1>
<div>Hello Foobar</div>
```

Das verinfachte Layout würde folgendesmaßen ausshen: Der Titel wird ersetzt mit “My Page Title” und der Body wird ersetzt mit dem `index.hbs` Template

```
<!DOCTYPE html>
<html>
<head>
  <meta charset="utf-8">
  <title>{{title}}</title>
</head>
<body>
  {{{body}}}
</body>
</html>
```

Übersetzungen

Mit dem Express Server haben wir auch die Möglichkeit Übersetzungsdateien zu benutzen. Diese Müssen wie bei jedem Plugin in `templates/lang` hinterlegt werden. Anders als bei normalen plugins sind es hier `*.js` Dateien. Der Name der Datei ist gleichzeitig die Sprache also “de” für Deutsch, “en” für Englisch und so weiter. In der Datei definieren wir ein JavaScript Objekt mit Key Value Paaren und exportieren dieses. Der Key wird dann benutzt um die Übersetzung zu laden. Dazu steht in der `express.js` Datei die `translate` Funktion zur Verfügung.

Die Übersetzungen sind mit I18next realisiert und deren Dokumentation kann für Spezialfälle herbeigezogen werden.

```
module.exports = {
  key: 'value'
  anotherKey: 'another value {{name}}'
}
```

Benutzung im Express Handler

```
module.exports = ({router, logger, translate}) => {
  router.get('/', (request, response) => {
    logger.info(translate('key'))
    logger.info(translate('anotherKey', {name: 'Foo'}))
    res.send('OK')
  });

  return router;
};
```

Handlebars mit Übersetzungen

Um die Übersetzungen in einem Template zu nutzen steht der Helper `t` bereit. Mit diesem können wir wie oben Übersetzungen anfragen.

```
module.exports = {
  helloWorld: 'Hello World'
  hello: 'Hello {{name}}'
```

```
}
```

```
<h1>{{t 'helloWorld'}}</h1>
<h2>{{t 'hello' name='Foo'}}</h2>
```

Logger

Das Plugin benutzt einen benutzerdefinierten Logger für ein einheitliches Logfile. Das Logfile wird dann benutzt um im Express Plugin die Logs anzuzeigen. Die Logger sind anhand von Plugins voneinander getrennt. Die Logs sind auf de Plugin Seite via Tags gekennzeichnet. Das Express Plugin bekommt den Tag “Express” wogegen ein Plugin mit dem Namen “Mein_Cooles_Plugin” mit “Mein Cooles Plugin” geflaggt wird. Das funktioniert standardmäßig wenn der Logger benutz wird. `console.log` werden nicht wie gewohnt funktionieren.

Die Logfiles werden unter `LBHOME/logs/plugins/express/` gespeichert. Fehler werden in der Datei `express-errors.log` gespeichert.

Im Moment werden alle Logs geschrieben, aber es gibt den Plan das Loglevel zu konfigurieren um beispielsweise nur Fehler zu loggen. Aber das funktioniert noch nicht.

Log Methoden

```
* info(message: String)
* debug(message: String)
* warn(message: String)
* error(message: String, error: Exception)
```