# Alinex Core

**Central module for the alinex systems**

*Alexander Schilling*

# Table of contents

The core module is a base for all of my Alinex modules. It contains three parts:

1. Some helper to **manage alinex modules** by myself and keep them in-sync like setup new modules or update the documentation theme.
2. **Basic functions** needed in alinex modules. This is a collection of small functions which are not covered in separate modules.
3. **Standards definition** for Alinex modules as documentation.

It's common methods which are necessary in nearly all of my packages at the moment contains:

- exit - handler for process exits
- logo - a method to present ASCII art command line logos

## 1.1 Installation

To include this in your module install it:

```
npm install @alinex/core --save
```

Now only include it in your code and call it's methods. Or use the `bin/...` commands to use the Alinex environment helpers.

## 1.2 Statistics

The following statistics will give you a better understanding about it's complexity. The documentation has 26 pages (in A4 PDF format) and contains 30781 characters. And this package has 180 lines of code.

The latest version in the repository is Version 1.7.1. It has a size of 236,68KiB in 35 files. The development contains 17 packages (resolved) and 0 of it are from the Alinex project.

## 1.3 Download Documentation

If you want to have an offline access to the documentation, feel free to download the 26 pages PDF Documentation.

## 1.4 License

### 1.4.1 Alinex Core



> **Apache License, Version 2.0**
>
> Licensed under the Apache License, Version 2.0 (the "License"); you may not use this file except in compliance with the License. You may obtain a copy of the License at
>
> http://www.apache.org/licenses/LICENSE-2.0
>
> Unless required by applicable law or agreed to in writing, software distributed under the License is distributed on an "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied. See the License for the specific language governing permissions and limitations under the License.

## 2.1 Install

The bash script under `node_modules/@alinex/core/bin/install` will initialize a new module to the alinex structure and add a lot of settings.

## 2.2 Update

The bash script under `node_modules/@alinex/core/bin/update` will update some of the settings from the core documentation or the @alinex/core files.

> **Note**
>
> You can also call it with `node_modules/.bin/alinex-update`.

## 2.3 Stats

The bash script under `node_modules/@alinex/core/bin/stats` will show and update some statistic data which can be included in the documentation (see it on the bottom of main page).

> **Example**
>
> You should include it with the following scripts in `package.json`:
>
> ```
> "stats": "node_modules/.bin/alinex-stats",
> "postpublish": "npm run stats && git add -A && git commit -m \"Update documentation\"; git push origin --all && git push origin --tags"
> ```
>
> Now you may update the stats using `npm run stats` and it will be done after the next publication automatically.

### 2.3.1 Man Pages

Some of my NodeJS scripts are callable from CLI on Linux systems. Therefore I also support man pages, but to make it easy I write them in markdown, too. The following script will generate new man pages in the `bin` folder out of the given markdown file:

> **Example**
>
> You should include it with the following scripts in `package.json`:
>
> ```
> "man": "node_modules/.bin/alinex-man src/datastore.1.md",
> "build": "npm run man && rm -rf lib && tsc",
> ```
>
> This will automatically update the man page on each build run.

## 3.1 Alinex Core Module

The core module is a base for all of my alinex modules. It has some common methods which are necessary in nearly any of my packages.

At the moment it contains:

- exit - handler for process exits
- logo - a method to present ASCII art command line logos

### 3.1.1 Installation

To include this in your module install it:

```
npm install @alinex/core --save
```

Now only include it in your code and call it's methods.

### 3.1.2 Usage

Command line modules should import the default methods and can use the logo display from it. But for module you may also only import specific functionalities:

**TypeScript**

```
// load default methods
import core from '@alinex/core'
// load all parts
import * as core from '@alinex/core'
// load specific elements into own variables
import { logo } from '@alinex/core'
```

**JavaScript**

```
// load only the defaults
const core = require('@alinex/core').default
// load all parts
const core = require('@alinex/core')
```

## 3.2 Exit Handler

See exit codes description for a detailed information about exit codes and the standards for them.

### 3.2.1 Error display

The error output of NodeJS is optimized to display better readable and colorful error messages. You will get this by only including this module once.

### 3.2.2 Interrupt Signals

This comes in two parts. At first a handler will be set which will work on interrupt signals and exit the process with a short message. This will be automatically done by loading the module:

**TypeScript**

```
import '@alinex/core'
```

**JavaScript**

```
require('@alinex/core');
```

Supported signals are:

| Interrupt | Signal | Exit Code |
|---|---|---|
| 1 | `SIGHUP` | 129 |
| 2 | `SIGINT` | 130 |
| 3 | `SIGQUIT` | 131 |
| 6 | `SIGABRT` | 134 |
| 9 | `SIGTERM` | 143 |

Not all signals are supported here, they are mapped to 128 + signal number for the exit code.

### 3.2.3 Managing Uncaught Errors

In NodeJS there may be some errors which were thrown and neither catched. This can be from rejected promises or thrown exceptions. The following code at the start of your program will catch them all and send them to the exit handler to make a nice error message and exit.

```
import core from '@alinex/core';

process.on('uncaughtException', err => core.exit(err, 1));
process.on('unhandledRejection', (err: any) => core.exit(err, 1));
```

### 3.2.4 Controlled Exit

Use this to manually exit the process with a possible error.

**Definition**

```
core.exit(
    err: ExitError | Error | string,
    code: number | string | null
)
```

It is possible to use the `exit` method for controlled exit:

**TypeScript**

```typescript
import core from '@alinex/core'

// exit with code 100
core.exit(new Error("Command already running, can't be called in parallel!"), 100);
```

**JavaScript**

```javascript
const core = require('@alinex/core');

// exit with code 100
core.exit(new Error("Command already running, can't be called in parallel!"), 100);
```

This will:

- output the `error.message`
- output a possible `error.description` property
- auto set the code if possible
- exit the process with the given code

The alinex tools are based on the bash exit codes 0, 1 and 124-143. In addition the codes 3-6 are used for NodeJS system codes and some alinex codes are set in the range 16-120 like:

| Code | Description |
|---|---|
| 0 | OK - no error |
| 1 | General error which should not occur |
| 2 | Command parameter problem |
| 3 | File system access problem |
| 4 | Network problems |
| 5 | Service or system access problem |
| 6 | No such service or address |
| 124 | command times out |
| 125 | if a command itself fails |
| 126 | Command invoked cannot execute |
| 127 | "command not found" |
| 128 | Invalid argument to exit |
| 129 | `SIGHUP` (Signal 1) |
| 130 | `SIGINT` like through `^ Ctrl` + `C` (Signal 2) |
| 131 | `SIGQUIT` (Signal 3) |
| 134 | `SIGABRT` or `SIGIOT` (Signal 6) |
| 143 | `SIGTERM` (Signal 15) |
| 255 | Exit status out of range |

## 3.2.5 ExitError

**Definition**

```
class ExitError extends Error {
    constructor(
        message: string,
        description?: string,
        code?: number | string
    )
    description?: string
    exit?: number
    code?: string
}
```

Such errors are used to auto detect the exit code and display a more detailed error message in case of exiting the system with this. Therefore the `exit()` method explicitly supports the additional fields of this type of Error.

TypeScript

```
import { exit, ExitError } from '@alinex/core'

const err = new ExitError('Wrong Parameter', 'Only the following parameters are allowed...')
err.exit = 2
exit(err);
```

JavaScript

```
const core = require('@alinex/core');

const err = new core.ExitError('Wrong Parameter', 'Only the following parameters are allowed...')
err.exit = 2
core.exit(err);
```

The exit code will be taken from:

- the `err.exit` field
- the `err.code` name (auto detect for node errors)

This procedure gives you the possibility to define the exit code then the error occurs and throw it first but decide later on other position if you want to give it to the exit handler.

## 3.3 Logo

Display an ASCII art logo on console.

**Definition**

```
core.logo(title: string): string
```

This is often used at startup of command line calls:

TypeScript

```ts
import core from '@alinex/core'

// output logo on console
console.log(core.logo('Code Documentation Extractor'));
```

JavaScript

```js
const core = require('@alinex/core');

// output logo on console
// eslint-disable-next-line no-console
console.log(core.logo('Code Documentation Extractor'));
```

This will give a logo like:


For some special custom developments alternative logos are available, which can be selected by environment setting `LOGO`.

## 4.1 Alinex JavaScript Standards

This part describes my current standards in coding JavaScript modules. They are followed not only in this small module, but in all my modules.

> **Warning**
>
> Keep in mind that older versions are using older no longer here documented standards.
>
> Also it isn't a must but a should rule so small differences to this standards are allowed.

The basic settings used for my new modules are:

| Software | Comment |
|---|---|
| Node >= 10 | Allows to use ES2018 |
| `typescript` | TypeScript |
| `debug` + `chalk` | Debugging on demand |
| `mocha` | To run unit tests |
| `mkdocs` | Documentation |

To work with this standards some small bash tools are included here.

### 4.1.1 Install Helper

> **Attention**
>
> You first have to create your package.json if not already done, use `npm init` to do this interactively.

To use this helpers you have to install them, which will check your system and load necessary libraries:

```
npm install @alinex/core --save
node_modules/@alinex/core/bin/install
```

### 4.1.2 Update to Standard

Afterwards you can update your project anytime by calling:

```
node_modules/@alinex/core/bin/update
```

This will install missing files in an initial version and update some which should be always up to date. The processing depends on the package type detected by the first part in the repository name.

#### Code in NodeJS

For packages with name `node-...`

- load `.gitignore` if not present from @alinex/core
- load `.gitlab-ci.yml` if not present from @alinex/core
- load `.npmignore` if not present from @alinex/core
- add mocha and typescript modules
- load `tsconfig.json` if not present from @alinex/core

#### Documentation Setup

This will setup the `mkdocs` documentation to be corresponding to alinex.gitlab.io:

- install `mkdocs` if not already done
- load `docs/assets` if not present from alinex.gitlab.io
- load `docs/assets/extra.css` from alinex.gitlab.io
- load `docs/assets/pdfmathjax.js` from alinex.gitlab.io
- load `docs/assets/default.jpg` from alinex.gitlab.io
- load `docs/assets/abbreviations.txt` from alinex.gitlab.io
- load `docs/manifest.webmanifest` from @alinex/core
- load `mkdocs.yml` if not present from @alinex/core
- load `.markdownlint.json` from @alinex/core
- load `docs/policy.md` if not present from @alinex/core
- write `docs/README.md` if not present
- write `docs/CHANGELOG.md` if not present
- write `docs/roadmap.md` if not present
- load `LICENSE.md` if not present from @alinex/core

### 4.1.3 NPM Scripts

Mostly the following scripts are defined to be called:

- `npm run man` - this will create the man page file for binary
- `npm run stats` - output statistics and update include file for documentation
- `npm run build` - compile typescript
- `npm run prepare` - call build before publishing
- `npm run test` - run the unit tests
- `preversion` - run tests before bumping version
- `postversion` - publish,
- `postpublish` - update stats, push to git origin

With the above scripts publishing is as easy as calling `npm version patch`, `npm version minor` or `npm version major` and npm as well as git is updated.

## 4.2 File Structure

All modules in the Alinex namespace will use the same directory structure. This follows the general standards and is described here.

### 4.2.1 Overview

The locally installed system may be in one of the following states which presents the development cycle of the system:

1. **Source** - the real code base
2. **Development** - after installing
3. **Build** - if installed from npm or after building
4. **Productive** - after the system is configured

Each of these states may have some of the possible directories so they will be referenced in the further description.

#### Source

The developer will start with the GIT source by cloning or forking.

#### Development

While in development sometimes additional directories will be created while compiling and testing the code.

#### Build

For productive use, this is the start point. You get a ready to run compiled system.

#### Productive

In the first run the system may be configured and create some additional directories for configurations and runtime data.

### 4.2.2 Possible directories

The following list displays all directories of any state which may exist each listed with the states to which it belongs:

```
.             # source
bin           # all
data          # all
docs          # source, development
etc           # all
lib           # (source, development) build, production
node_modules  # development, ...
src           # source, development
test          # source, development
var           # production data
```

Read the further sections to get more information of what resides in which directory and how it is used and created.

### 4.2.3 Ignoring Files

To properly support the file structure in all phases two ignore files are needed:

- `.gitignore` (used to not push everything to git)
- `.npmignore` (used fro npm publishing)

## 4.2.4 Directories

#### Source Stage

The source specifies what is stored in the code repository.

This stage contains the following directories:

```
bin           # executable files
data          # base data
  locals      # i18n localization
etc           # default/example configuration
docs          # general documentation as gitbook
src           # source code
test          # test data and test suites
  data        # test data
  mocha       # mocha test suites
```

The source code resides in the `src` folder and will be copied/compiled into `lib` to run. This step is done on build of package.

#### Development Stage

Shows what the developer will find on his machine while developing and testing the system. While testing the development system will also get all the directories from productive which are not listed here.

This stage contains the following directories:

```
bin           # executable files
data          # base data
docs          # created documentation (optional)
etc           # default/example configuration
lib           # copied/compiled code
node_modules  # npm installed packages
src           # source code
test          # test data and test suites
```

#### Installed Stage

This is what you get after a fresh npm installation.

This stage contains the following directories:

```
bin           # executable files
data          # base data
etc           # configurations
lib           # copied/compiled code
var           # data and code which maybe changed in installation
node_modules  # npm installed packages
```

#### Productive Stage

And finally this shows what resides on the productive server.

This stage contains the following directories:

```
bin           # executable files
data          # base data
etc           # configurations
lib           # copied/compiled code
var           # data and code which maybe changed in installation
  config      # compiled configuration
  data        # persistent file store
  log         # log files
node_modules  # npm installed packages
```

### 4.2.5 Where what belongs to

The following list should give an overview of there to store what:

- cache files -> systems temp folder
- configuration -> system or user `alinex` folder or `etc`
- resources for binaries -> `bin/lib`
- temporary files -> systems temp folder

## 4.3 TypeScript Language

Most modules will be written in TypeScript but be compiled into pure JavaScript with additional TypeScript definitions for runtime. That means it is also usable if you are not familiar with typescript.

The decision to use typescript is based on:

- type safety for more reliable code
- it is a superset of javascript
- great IDE support

At the moment typescript is configured to:

- compile to ES2018, but it may be changed anytime
- using commonJS module syntax, maybe changing to ES2016 later
- TypeScript declarations included
- sourceMaps are generated
- comments are removed
- strict type checking is enabled
- unused locals are not allowed

### 4.3.1 package.json

To support typescript you should add the following:

**package.json**

```
"scripts": {
    "build": "rm -rf lib && tsc",
    "prepare": "npm run build",
    "test": "mocha -r ts-node/register test/mocha/**.ts --exit"
}
```

### 4.3.2 JSDoc

To further help developers in the IDE, all public elements of a package should be documented using JSDoc.

## 4.4 Executables

If your package contains executable files like bash scripts to start from CLI this is best supported as follows.

### 4.4.1 File Structure

The relevant files are:

```
bin/
    my-script       # script with x-flag calling cli.js
    my-script.1     # man file
lib/
    cli.js          # compiled start file
src/
    my-script.1.md  # source for man file
    cli.ts          # typescript source
package.js          # package dependencies
```

### 4.4.2 Usage

From the bottom up we need a transformer to create the man file from the markdown source:

```
npm install -y marked-man marked
```

> **Attention**
>
> Because `marked-man` only works with `marked` V0.7.0 don't install newer versions, yet.

Add the script to build the man pages, add them to the build script and reference the resulting file as man page:

**package.json**

```
"scripts": {
    "man": "node_modules/.bin/marked-man src/my-ascript.1.md > bin/extract.1",
    "build": "npm run man && rm -rf lib && tsc",
    "prepare": "npm run build"
},
"man": [
    "./bin/my-script.1"
]
```

Now the real binary should be called using NodeJS in the she-bang and it's only purpose is to load the `lib/cli.js` and let it work with one of those scripts:

**JavaScript**

```
#!/usr/bin/env node

require(`${__dirname}/../lib/cli.js`) // eslint-disable-line import/no-dynamic-require
```

**Bash**

```
#!/bin/bash

source_dir=$(dirname $(readlink -f "${BASH_SOURCE[0]:-$(pwd)/x}"))
/usr/bin/env node "$(dirname $source_dir)/lib/cli.js"
```

The a bit complex looking part is to detect the script directory and call `cli.js` from there.

## 4.5 Debug Output

While working in different stages like development or even production it may be useful to switch on and off debug statements on demand.

This is done through a tiny NodeJS debug utility modeled after node's core debugging technique. It can be enabled through environment settings and will printout some predefined messages. Another small helper called `chalk` to make the output messages color full.

A lot of foreign libraries are also using debug so it will work on their code also.

### 4.5.1 Installation

Because this should also stay in the productive code use:

```
npm install debug chalk --save
```

### 4.5.2 Usage

Simply create a `debug` instance by giving it the name of your module and maybe the subroutine as concatenated function. Using it's `debug` method will return a decorated version of the given message to `console.error` only if debug is enabled.

**TypeScript**

```ts
import Debug from 'debug';

// initialize debug with module name and maybe sub element
const debug = Debug('<module>')('<sub>');

debug('new instance created using %o', setup);
```

**JavaScript**

```js
const Debug = require('debug');

// initialize debug with module name and maybe sub element
const debug = Debug('<module>')('<sub>');

debug('new instance created using %o', setup);
```

You may also make the output colorful by also adding the earlier mentioned `chalk` module:

**TypeScript**

```ts
import Debug from 'debug';
import chalk from 'chalk';

// initialize debug with module name and maybe sub element
const debug = Debug('<module>')('<sub>');

debug(chalk.red('new instance created using %o'), setup);
```

**JavaScript**

```js
const Debug = require('debug');
const chalk = require('chalk');

// initialize debug with module name and maybe sub element
const debug = Debug('<module>')('<sub>');

debug(chalk.red('new instance created using %o'), setup);
```

#### Code to run only if debug enabled

```
if (debug.enabled) {
    debug(claculateMessage());
}
```

This allows to only run the message calculation if debugging is enabled for this module.

#### Formatting

Debug uses `printf`-style formatting supporting the following formatters:

| Formatter | Representation |
|---|---|
| `%O` | Pretty-print an Object on multiple lines. |
| `%o` | Pretty-print an Object all on a single line. |
| `%s` | String. |
| `%d` | Number (both integer and float). |
| `%j` | JSON. Replaced with the string '[Circular]' if the argument contains circular references. |
| `%%` | Single percent sign ('%'). This does not consume an argument. |

#### Debug names

The convention is to use the short name of the module as debug name like `config` for the `alinex-config` module. The sub names are often used after their file names which implies the functional part like `config:compile` for the compiler class in `compile.js`.

Also within the tests you may use `test` as debug module name.

### 4.5.3 Colors

The `debug` module already uses colors if possible for the namespace identifier. It will try to pick a different color for each namespace to make it easier to overflow a bigger output. But like described above you may use `chalk` yourself to make the message itself also colorful. Possible colors are:

- `red`, `bold.red`
- `green`, `bold.green`
- `yellow`, `bold.yellow`
- `blue`, `bold.blue`
- `magenta`, `bold.magenta`
- `cyan`, `bold.cyan`
- `white`, `bold.white`
- `gray`, `bold.gray`
- `black`, `bold.black`

The `bold` versions resemble the color more. But you may also combine this with `dim`, `italic`, `underline`, `inverse`, `strikethrough` and also set `bgRed` ...

### 4.5.4 Control debugging

When running the code, you can set a few environment variables that will change the behavior of the debug logging:

| Name | Purpose |
|---|---|
| `DEBUG` | Enables/disables specific debugging namespaces. |
| `DEBUG_COLORS` | Whether or not to use colors in the debug output. |
| `DEBUG_DEPTH` | Object inspection depth. |
| `DEBUG_SHOW_HIDDEN` | Shows hidden properties on inspected objects. |

For the `DEBUG` variable, you can give multiple comma separated namespaces using colon as delimiter between the namespace levels. An `*` character is also possible to select multiple or all:

| DEBUG | Usage |
| --- | --- |
| `DEBUG=*` | Show all messages. |
| `DEBUG=test*` | Show additional messages from the test suite. |
| `DEBUG=config,config:compile` | Show only the main and compile messages from the config module. |
| `DEBUG=config*,file*` | Show all config and file module messages. |

## 5.1 Version 1.7.0 (07.05.2021)

- upgrade to ES2019, Node >= 12
- added test coverage report
- Hotfix 1.7.1 (09.05.2021) - fix typescript settings

## 5.2 Version 1.6.0 (01.01.2021)

- new doc theme
- update development packages
- Hotfix 1.6.1 (01.01.2021) - fix pdf link and optimize license inclusion
- Update 1.6.2 (02.01.2021) - add man page creation helper
- Hotfix 1.6.3 (03.01.2021) - also update bash modules
- Update 1.6.4 (06.05.2021) - update docs and eslint
- Update 1.6.5 (07.05.2021) - improved update script

## 5.3 Version 1.5.0 (23.10.2020)

- Restructure documentation with abbreviations
- Add automatic generated statistics to docs
- Hotfix 1.5.1 (23.10.2020) - fix stats size calculation
- Hotfix 1.5.2 (23.10.2020) - fix counting only of different packages
- Hotfix 1.5.3 (23.10.2020) - fix alinex package counter
- Hotfix 1.5.4 (23.10.2020) - fix alinex package counter again

## 5.4 Version 1.4.0 (15.02.2020)

Updating NPM scripts to better automate publishing.

- Version 1.4.1 (15.02.2020) - update modules
- Version 1.4.2 (08.03.2020) - update mkdocs settings
- Version 1.4.3 (08.03.2020) - fix PDF creation
- Version 1.4.4 (10.07.2020) - update modules
- Version 1.4.5 (26.09.2020) - better error reporting
- Version 1.4.6 (17.10.2020) - fix updating

## 5.5 Version 1.3.0 (02.08.2019)

Remove async methods because now extracted into alinex-async.

- Version 1.3.1 (27.12.2019) - update modules

## 5.6 Version 1.2.0 (30.06.2019)

Add async helper to work with promises in an easy way supporting: delay, retry, each, map, filter, parallel.

## 5.7 Version 1.1.0 (17.04.2019)

- add ExitError as class
- Version 1.1.1 (25.04.2019) - adding optimized nodejs error output
- Version 1.1.2 (03.05.2019) - add doc comments

## 5.8 Version 1.0.0 (08.04.2019)

- transform to TypeScript
- document new Standards
- adding bash update tool
- Version 1.0.1 (08.04.2019) - fix update script - add update documentation

## 5.9 Version 0.4.1 (04.04.2019)

- add exit code handling

## 5.10 Version 0.3.0 (04.04.2019)

- Transform into pure JavaScript (flow before)
- remove exit code handling

## 5.11 Version 0.2.12 (28.06.2017)

- Fix divibib logo

## 5.12 Version 0.2.11 (28.06.2017)

- Skip tests
- Add possibility for shutdown function on exit handler
- Fix documentation to hide style comments in github view.
- Updated ignore files.
- Update Travis.

## 5.13 Version 0.2.10 (19.08.2016)

- Update alinex-builder@2.3.6, alinex-util@2.4.0, mocha@3.0.2
- Fix error code to be at least 1 if error given.
- Update docs.
- Update documentation.
- Fix logo documentation.
- Slightly updated documentation.
- Update documentation.

- Rename repository to use general alinex namespace naming.
- Add GitHub link.
- Rename index title.
- Updated documentation.
- Update docs.
- Add copyright sign.

## 5.14 Version 0.2.9 (09.06.2016)

- Fix exit without error.

## 5.15 Version 0.2.8 (09.06.2016)

- Fix exit without error.

## 5.16 Version 0.2.7 (09.06.2016)

- Upgraded util, mocha and builder package.
- Allow calling exit without error.

## 5.17 Version 0.2.6 (31.05.2016)

- Upgraded builder package.
- Optimize alinex logo.

## 5.18 Version 0.2.5 (28.04.2016)

- Added optional exit code.
- Fix error code detection for undefined code or code -1.
- Fix changelog.

## 5.19 Version 0.2.4 (27.05.2016)

- Add exit code auto detection.

## 5.20 Version 0.2.3 (27.04.2016)

test new publish task

- Fixed tests to always have chalk enabled.
- Upgraded builder.
- Added missing util package.
- Merge with origin.
- Upgraded chalk package.

## 5.21 Version 0.2.2 (22.04.2016)

- Fixed missing packages.

## 5.22 Version 0.2.0 (22.04.2016)

- Added exit handling.

## 5.23 Version 0.1.4 (29.02.2016)

- Change name because of NPM install problems.

## 5.24 Version 0.1.3 (29.02.2016)

- Fixed tests.

## 5.25 Version 0.1.2 (26.02.2016)

- Update version.
- Update version.

## 5.26 Version 0.1.1 (26.02.2016)

- Update documentation.
- Upgraded builder, `chai` and `mocha`.
- Fixed changelog.

## 5.27 Version 0.1.0 (26.02.2016)

- Added tests for logo display.
- Fixed package.json
- Restructure module to include logo code.

**Info**

I don't have a roadmap here, because this module will be extended any time a small method is seen as global to all my modules.

### 6.0.1 Bugs

- support debian/arch in update/install

This documentation is part of the **alinex.gitlab.io** site and as such please have a look on the site's privacy statement.