Vim plugin index documentation

Release 0.0.1

Nikolay Pavlov

Sep 24, 2017

Contents

1 Projects that are part of the Vim plugin index **3**

2 Database directory structure **5**

3 Plugin-info file **9**

4 Contributing to this database **13**

4.1 Adding information about a new plugin 13

4.2 Adding deprecation warning 14

4.3 Adding information about a fork 14

Contents:

CHAPTER 1

Projects that are part of the Vim plugin index

Vim plugin index Database in format described by *this documentation*.

Vim-pi legacy plugin index Database in old format. Should not be used and will eventually be removed.

Vim-pi documentation Vim plugin index documentation. Contains sources for the documentation you are currently viewing.

Vim-pi descriptions Database containing plugin descriptions. Feel free to use it to construct search indexes or do some research.

Vim-pi tools Tools used by developers. Some notes about this repository:

- Backward or forward compatibility between tools is not guaranteed.
- There is no official documentation for these tools.
- These tools are used to build Vim plugin index database and/or manipulate it.

Vim-pi private data Database containing data used by *vim-pi developer tools*. Format or existence of particular data is not guaranteed. You need to use up-to-date developer tools with up-to-date private data: compatibility is not guranteed as well.

CHAPTER 2

Database directory structure

Note: Example JSON code blocks show generic JSON structure, *not* the exact layout of data physically written to the file. E.g. plugin managers must not rely on vim-pi (not) writing the whole file as one long line.

/ Root of the *database*. Contains all of the following data.

/index.json Index of all plugins. Is a JSON file containing mapping with the following format:

```
{
    "{name}": {
        "last-update-time": "{update-time}",
        "last-release-time": "{release-time}",
        "description": "{description}",
        "author": "{author}",
        "vim-script-nr": {scriptnr},
        "alternate-names": ["{name1}", "{name2}", ..., "{nameN}"],
        "deprecated": {deprecated}
    },
    ...
}
```

Fields:

{name} Exactly the same name as *plugin directory* name.

{update-time} Field {update-time} is written in a very strict variant of ISO-8601 (described below), so unless you are preparing for 101 century you can safely use simple string comparison. This field should be checked by plugin managers when they decide whether they need to update information about plugin.

Required.

{release-time} Like above, but is only altered when either new release, new fork or new development version were added. Is not altered when description or hooks were changed or when one of the old versions was removed.

Required.

Note: Vim-PI is not tracking development version updates if they use some of the version control systems. Plugin managers are supposed to simply rely on used VCS to update such plugins.

{author} Author name. Optional.

{description} Latest description of the plugin. May be used for searches. Optional.

{scriptnr} A script number on www.vim.org. Optional.

{name1} ... {nameN} Alternate names may be not unique and are supposed to represent alternative variants of writing plugin: e.g. "VAM" for "vim-addon-manager". Optional. Plugin managers are supposed to prefer these names over fuzzy matches.

Note: Two plugins may share the same alternate name.

{deprecated} `true` if plugin was deprecated. If it was not this field is absent, but may be set to `false` as well. Optional. Plugin managers are supposed to remove deprecated plugins from search and completion unless configured otherwise.

Note: For forward compatibility plugin managers must not rely on absence of keys that are not described here.

Note: ISO-8601 is very permissive. For `index.json` there are additional restrictions:

- UTC time zone indicated by `Z` at the end,
- nanoseconds; uses comma as decimal fraction separator,
- hyphenminuses (ASCII dashes) as year, month and day separator,
- no week dates,
- `T` as date/time separator,
- `:` as hour/minute/second separator.

In this format UNIX Epoch will look like this:

```
1970-01-01T00:00:00,000000000Z
```

/update-times.dat Smaller version of the index of all plugins: contains only *plugin name* – *last update time* – *last time of a new release* 3-tuples in the format:

```
{name1}\0{last-update-time1}\0{last-release-time1}\n
...
{nameN}\0{last-update-timeN}\0{last-release-timeN}\n
```

, trailing newline is always present and all *-time fields have all separators stripped and are missing timezone: UNIX Epoch will look like `19700101000000000000000` so plugin managers still may use string comparison, but also numeric comparison (if they are written in a language with big integer support and want to waste time on creating it).

It is intended that plugin managers use this file only to check for available updates.

/plugins/ Directory, containing *plugin directories*.

/plugins/*name*/ Directory with files specific to the given plugin.

/plugins/*name*/MANIFEST.json List of the files in this directory. List format:

```
{
    "{filename1}": {"size": {size1}, "sha256": "{sha256_1}"},
    "{filename2}": {"size": {size2}, "sha256": "{sha256_2}"},
    ...
    "{filenameN}": {"size": {sizeN}, "sha256": "{sha256_N}"}
}
```

. Each filename is a path relative to `/plugins/name` directory.

Note: For forward compatibility plugin managers must not rely on presence of sha256 key (it may be replaced with other hash(es) in the future, though it is more likely that they will be just added) or absence of any keys that are not described here.

/plugins/*name*/plugin-info.json Top-level plugin-info file. Format is described in *plugin-info file documentation*.

Note: This file must not contain *repository* and *version* keys. These key must be defined in plugin-info file inside *release* or development directory.

/plugins/*name*/hooks/ Contains hoodospel hook files used by *plugin-info file*.

/plugins/*name*/hooks/*hook*.hds Contains one specific hook. You should replace *hook* with one of the stage names identical to *one of the hook keys* from *plugin-info file*. If hook should be applied at both `post-install` and `post-update` stages it should be named `post`. If identical hook should be run at two or more other stages then you should deduce a name on your own and raise an issue at vim-pi documentation issue tracker describing this name and the case in which you need identical names for both stages.

All `post*` and `pre*` are reserved for hooks that are described directly in *relevant keys* in a *plugin-info file*. Names not starting with `post` or `pre` may be used for code that is common to more then one hook.

/plugins/*name*/releases/ Directory that contains version-specific information for all plugin versions.

/plugins/*name*/releases/*version*/ Directory that contains version-specific information for one plugin version. Plugin version must not contain `@` character: it is reserved for version variants (e.g. `0.1` contains description for installing plugin version `0.1` from archive and `0.1@git` uses git) and forks (e.g. `marcweber@0.0` for version `0.0` of a fork created by Marc Weber).

/plugins/*name*/releases/*version*/plugin-info.json Main plugin-info file. Format of this file is described in *plugin-info file documentation*. Is merged with top-level *plugin-info file*.

/plugins/*name*/releases/*version*/hooks/ Same as *top-level hooks directory*, but contain hooks specific to given plugin version. These hooks will be used first.

/plugins/*name*/development/ Directory that contains all variants of development installations.

/plugins/*name*/development/*variant*/ Directory that contains version-specific information for development plugin version. Directory structure is the same as for *release directory*. *variant* should be either a name of the author of the fork, preceded with `fork@`, an upstream mirror that uses different VCS in a format `mirror@vcs` where *vcs* must be one of the *repository types* or just `upstream`.

Note: Forks must not be chosen by plugin managers by default.

/plugins/*name*/files/ Miscellaneous files that are not any of the above files.

CHAPTER 3

Plugin-info file

Plugin-info file is a regular JSON file that contains JSON dictionary with the following keys:

Required keys:

repository Repository description, must not be present in *top-level plugin-info file*. Contains a dictionary with the following keys:

type Required. Must contain one of the following strings:

archive Designates that this dictionary describes an archive downloaded from given URL.

file Designates that this dictionary describes a plain .vim file.

hg, git, svn, bzr, darcs Designates that this dictionary describes a repository controlled by the given version control system.

any name starting with _ Reserved for plugin managers. Must not be present in the database.

url Required. For `archive` and `file` repository types it determines URL of the file to download, for various VCS repository types it determines location from where it should clone the repository.

Note: Plugin managers are supposed to use this URL when updating.

---

revision Optional if *repository type* is one of VCS types, must not be present otherwise. Determines revision that should be checked out, may be a branch name.

vim-directory Describes directory where plugin `.vim` file(s) should be moved if *repository type* happens to be `file`.

unpack-sequence Required if *repository type* happens to be `archive`, optional if it happens to be `file` and must be absent in other cases. Contains a list of strings which determine the unpack sequence that should be performed by a plugin manager to unpack downloaded archive. If type is `file` then this list must contain only stream compress formats. Known formats:

Format	Description
`gz`, `xz`, `lzma`, `bz2`	Stream compress formats: gzip, xz, lzma, bzip2.
`tar`	Tape archive (tar) format.
`zip`	Zip archive format.
`rar`	RAR archive format.
`cab`	CAB (CABinet) archive format.
`arj`	ARJ archive format.
`jar`	JAR (Java ARchive) archive format.
`7z`	7z archive format (should not be used for non-.7z archives supported by 7-zip, but not listed here).
`vmb`	Vimball archives.

`file-name` Required if *repository type* happens to be `file`, optional if it happens to be `archive` and must be absent in other cases. Contains a string that is the name of the `.vim` file (when type is `file`) or downloaded archive name (when type is `archive`) and must not be used for anything but determining how to name file downloaded from *the given URL*.

Note: For archived files like `pt.vim.gz` (`repository-type` is `file`) the `file-name` key is `pt.vim.gz`, not `pt.vim`. It is up to the plugin manager to determine what will the name of the file be after unpacking.

`strip-components` Optional. Tells plugin manager to strip given number of top-level directories. Only valid for repositories with type `archive`.

Note: Plugin managers *must* strip given number of path components when this key is present. They are free to do automatic detection in case it is not present.

`name` String, name of the plugin. Value must match plugin directory name.

`version` String. In database it is required to be the same as *{version}* component of *release directory* and be absent in any other plugin-info files.

Optional keys:

`dependencies` Dictionary, description of the dependencies. Must be present in the form `dependency_name : dependency_description`: `dependency_name` is a dictionary key, `dependency_description` is a dictionary. The latter may contain keys listed below:

`version` A single string describing allowed dependency versions. String must look like the following:

```
version :: top_constraints " " version_base
top_constraints :: [<=>]? "=" | [<>]
version_base :: version_component ( "." version_component )*
version_component :: same_as_current | number | any
same_as_current :: "~"
number :: [^.~*] [^.]*
any :: "*"
```

Absense of this key works like if `==` `*` was specified.

Description of the format string:

0. Versions are supposed to have format like `1.2.3.4`. Semantic versioning is preferred, but not forced. In place of numbers any alphanumeric sequence may appear. When comparing versions only the first numeric part at the start of each version component will be taken into account: `alpha1` equals zero

because `a` is not a digit, components `123rc` and `123rc` are considered equal, as well as `1rc2` and `1rc3`.

When parsing version any non-alpha-numeric character that is not dot or ~ is stripped out, producing new component: `2014-05-16` is converted into `2014.05.16`. This applies both to `version` key in `dependencies` dictionary and `version` key of the actual dependency. Using identical to constraint disables this.

1. `top_constraints` specify which versions should be considered to be allowed. Possible variants: < (lesser then), > (greater then), <= (lesser then or equal to), >= (greater then or equal to), = (equal to), == (identical to).

   Checked version is considered matched if its first differing component matches given constraint. Missing component is strictly lesser then any other value.

   When using "indentical to" constraint corresponding string is always treated as a single component and is matched literally unless it is equal to `same_as_current` or `any`.

2. `same_as_current` is substituted with value of the component in the same position of the version of the package for which dependencies are defined. `any` means that version component may have any value, and all of the following components may have any value, including missing. It thus must be the last atom.

   All lesser and greater constraint variants automatically receive `any` as the last component, no matter whether or not it is specified.

   "Identical to" top constraint allows either a single component `same_as_current` or a random string.

Examples (assuming *version key* is `2.6.10`):

Version definition	Matched examples	Not matched examples
`>= 1.2`	1.2, 1.2.5	1.1, 0.1
`> 1.2`	1.2rc1, 1.3	1.1, 1.2
`= 1.2`	1.2, 1foo2	1.2.1, 1.1, 1.3
`= 1.2.*`	1.2, 1.2.3	1.3, 1.1, 1
`= ~.~.*`	2.6, 2.6.5	2.5, 2
`== ~`	2.6.10	2f6f10, 2.6.10.1
`== 1alpha2`	1alpha2	1.2, 1.0.2, 1foo2
`== *`	1, alpha,	{no examples}
`= *`	1.0, 1, 1.2.3	{no examples}
`< 1.2`	1.0, 1	1.2, 2.0
`<= 1.2`	1.2, 1.0, 1	1.2.3, 2.5

`optional` Boolean. Determines whether described dependency is optional. Defaults to false.

`build` Boolean. Determines whether this is build-time dependency. Defaults to true.

`homepage` String, the home page of the plugin.

`vim-script-nr` Number, script number on vim.org website.

`author` String. Describes author of the plugin in format `name <email>`.

`maintainer` String. Describes maintainer of the plugin in format `name <email>`.

`description` String. Plugin description.

`deprecation-warning` String, deprecation warning message that should be displayed when installing the plugin.

Note: Plugin managers *must* display this message when they attempt attended installation of plugins with this key in plugin-info file. They may also display this message when attempting attended update and it appears that

this key is present in new plugin-info, but did not exist in the old one.

---

replacements List of strings, plugins that are suggested to replace deprecated plugin. Must not be present if there is no *deprecation-warning* key. Contributors *must* not list plugin competitors here unless plugin was deprecated for one of the *listed* reasons.

pre-install, post-install, pre-update, post-update Tells plugin manager what to run in different cases: `*-install` hooks are run before or after plugin installation, `*-update` hooks are run before or after update. Value is a plain string that must correspond to one of the file names in *hooks directory* without an extension. Note that hook may also be located in *release-specific hooks directory*.

alternate-names List of strings: alternate names of the plugin. Used for populating *alternate-names key* in the *index.json file*.

CHAPTER 4

Contributing to this database

Contributions are accepted in a form of bitbucket pull requests. There exist the following types of contribution:

Note: One pull request must contain only one contribution type. It may contain more then one contribution with the given type though.

Note: There must be no merge commits in the pull request.

Adding information about a new plugin

To add information about a new plugin one should create a new *plugin directory tree* in `/plugins/` directory without `MANIFEST.json` file (it will be generated later). Files this tree must contain (relative to plugin directory):

Top level *plugin info file* *Plugin info* file. As mentioned in *this file documentation* it must not contain *repository* key.

***Plugin info file* in one of the releases directories** *Plugin info* file inside *release* or *development* subdirectory. Must contain at least *repository* key.

Tree also may contain hooks and files used by hooks (in *files* subdirectory).

Plugin name must consist of latin letters, digits, dashes and underscores. Only ASCII variants of these symbols are allowed. There must not be two consequent dashes or underscores (e.g. substrings `--` and `__` are not allowed, but `_-_` is). It also must be unique.

Note: Contributors should not modify *index.json file* or *update times file*. Just like MANIFEST they will be generated later.

Adding deprecation warning

Deprecation warning is added to the top-level *plugin info file* to the key *deprecation-warning*. Contributor may also add *replacements* key. First key must contain a reason for deprecating this plugin and may also contain suggestions for the user about replacing this plugin.

Valid reasons for deprecation:

- Plugin author explicitly described his plugin as deprecated. In this case deprecation warning should be added even if there are no alternatives.
- Features of this plugin were included in one of its dependencies.
- Plugin depends on missing or deprecated plugins.
- Plugin was last updated at least six months ago, contains known bugs and there is a replacement for it. One may consider contributing information about *a fork* instead.

Adding information about a fork

TODO