NAME

CPAN::Meta::Spec - specification for CPAN distribution metadata

VERSION

version 2.150001

SYNOPSIS

```
my $distmeta = {
name => 'Module-Build',
abstract => 'Build and install Perl modules',
description => "Module::Build is a system for "
. "building, testing, and installing Perl modules. "
. "It is meant to ... blah blah blah ...",
version => '0.36',
release_status => 'stable',
author => [
'Ken Williams <kwilliams@cpan.org>',
'Module-Build List <module-build@perl.org>', # additional contact
],
license => [ 'perl_5' ],
prereqs => {
runtime => {
requires => {
'perl' => '5.006',
'ExtUtils::Install' => '0',
'File::Basename' => '0',
'File::Compare' => '0',
'IO::File' => '0',
},
recommends => {
'Archive::Tar' => '1.00',
'ExtUtils::Install' => '0.3',
'ExtUtils::ParseXS' => '2.02',
},
},
build => {
requires => {
'Test::More' => '0',
},
}
},
resources => {
     license => ['http://dev.perl.org/licenses/'],
},
optional_features => {
domination => {
description => 'Take over the world',
prereqs => {
develop => { requires => { 'Genius::Evil' => '1.234' } },
runtime => { requires => { 'Machine::Weather' => '2.0' } },
},
},
},
dynamic_config => 1,
keywords => [ qw/ toolchain cpan dual-life / ],
'meta-spec' => {
```

```
version => '2',
     url     => 'https://metacpan.org/pod/CPAN::Meta::Spec',
},
generated_by => 'Module::Build version 0.36',
};
```

DESCRIPTION

This document describes version 2 of the CPAN distribution metadata specification, also known as the "CPAN Meta Spec".

Revisions of this specification for typo corrections and prose clarifications may be issued as CPAN::Meta::Spec 2.*x*. These revisions will never change semantics or add or remove specified behavior.

Distribution metadata describe important properties of Perl distributions. Distribution building tools like Module::Build, Module::Install, ExtUtils::MakeMaker or Dist::Zilla should create a metadata file in accordance with this specification and include it with the distribution for use by automated tools that index, examine, package or install Perl distributions.

TERMINOLOGY

distribution
: This is the primary object described by the metadata. In the context of this document it usually refers to a collection of modules, scripts, and/or documents that are distributed together for other developers to use. Examples of distributions are `Class-Container`, `libwww-perl`, or `DBI`.

module
: This refers to a reusable library of code contained in a single file. Modules usually contain one or more packages and are often referred to by the name of a primary package that can be mapped to the file name. For example, one might refer to `File::Spec` instead of *File/Spec.pm*

package
: This refers to a namespace declared with the Perl `package` statement. In Perl, packages often have a version number property given by the `$VERSION` variable in the namespace.

consumer
: This refers to code that reads a metadata file, deserializes it into a data structure in memory, or interprets a data structure of metadata elements.

producer
: This refers to code that constructs a metadata data structure, serializes into a bytestream and/or writes it to disk.

must, should, may, etc.
: These terms are interpreted as described in IETF RFC 2119.

DATA TYPES

Fields in the "STRUCTURE" section describe data elements, each of which has an associated data type as described herein. There are four primitive types: Boolean, String, List and Map. Other types are subtypes of primitives and define compound data structures or define constraints on the values of a data element.

Boolean

A *Boolean* is used to provide a true or false value. It **must** be represented as a defined value.

String

A *String* is data element containing a non-zero length sequence of Unicode characters, such as an ordinary Perl scalar that is not a reference.

List

A *List* is an ordered collection of zero or more data elements. Elements of a List may be of mixed types.

Producers **must** represent List elements using a data structure which unambiguously indicates that multiple values are possible, such as a reference to a Perl array (an ''arrayref'').

Consumers expecting a List **must** consider a String as equivalent to a List of length 1.

Map

A *Map* is an unordered collection of zero or more data elements (''values''), indexed by associated String elements (''keys''). The Map's value elements may be of mixed types.

License String

A *License String*is a subt ype of String with a restricted set of values. Valid values are described in detail in the description of the ''license'' field.

URL

URL is a subtype of String containing a Uniform Resource Locator or Identifier. [This type is called URL and not URI for historical reasons.]

Version

A *Version* is a subtype of String containing a value that describes the version number of packages or distributions. Restrictions on format are described in detail in the ''Version Formats'' section.

Version Range

The *Version Range* type is a subtype of String. It describes a range of Versions that may be present or installed to fulfill prerequisites. It is specified in detail in the ''Version Ranges'' section.

STRUCTURE

The metadata structure is a data element of type Map. This section describes valid keys within the Map.

Any keys not described in this specification document (whether top-level or within compound data structures described herein) are considered *custom keys* and **must** begin with an ''x'' or ''X'' and be followed by an underscore; i.e. they must match the pattern: `qr{\Ax_}i`. If a custom key refers to a compound data structure, subkeys within it do not need an ''x_'' or ''X_'' prefix.

Consumers of metadata may ignore any or all custom keys. All other keys not described herein are invalid and should be ignored by consumers. Producers must not generate or output invalid keys.

For each key, an example is provided followed by a description. The description begins with the version of spec in which the key was added or in which the definition was modified, whether the key is *required* or *optional* and the data type of the corresponding data element. These items are in parentheses, brackets and braces, respectively.

If a data type is a Map or Map subtype, valid subkeys will be described as well.

Some fields are marked *Deprecated*. These are shown for historical context and must not be produced in or consumed from any metadata structure of version 2 or higher.

REQUIRED FIELDS

abstract

Example:

```
abstract => 'Build and install Perl modules'
```

(Spec 1.2) [required] {String}

This is a short description of the purpose of the distribution.

author

Example:

```
author => [ 'Ken Williams <kwilliams@cpan.org>' ]
```

(Spec 1.2) [required] {List of one or more Strings}

This List indicates the person(s) to contact concerning the distribution. The preferred form of the contact string is:

```
contact-name <email-address>
```

This field provides a general contact list independent of other structured fields provided within the ''resources'' field, such as `bugtracker`. The addressee(s) can be contacted for any purpose including but not limited to (security) problems with the distribution, questions about the distribution or bugs in the distribution.

A distribution's original author is usually the contact listed within this field. Co-maintainers, successor maintainers or mailing lists devoted to the distribution may also be listed in addition to or instead of the original author.

dynamic_config

Example:

```
dynamic_config => 1
```

(Spec 2) [required] {Boolean}

A boolean flag indicating whether a *Build.PL* or *Makefile.PL* (or similar) must be executed to determine prerequisites.

This field should be set to a true value if the distribution performs some dynamic configuration (asking questions, sensing the environment, etc.) as part of its configuration. This field should be set to a false value to indicate that prerequisites included in metadata may be considered final and valid for static analysis.

Note: when this field is true, post-configuration prerequisites are not guaranteed to bear any relation whatsoever to those stated in the metadata, and relying on them doing so is an error. See also ''Prerequisites for dynamically configured distributions'' in the implementors' notes.

This field explicitly **does not** indicate whether installation may be safely performed without using a Makefile or Build file, as there may be special files to install or custom installation targets (e.g. for dual-life modules that exist on CPAN as well as in the Perl core). This field only defines whether or not prerequisites are exactly as given in the metadata.

generated_by

Example:

```
generated_by => 'Module::Build version 0.36'
```

(Spec 1.0) [required] {String}

This field indicates the tool that was used to create this metadata. There are no defined semantics for this field, but it is traditional to use a string in the form ''Generating::Package version 1.23'' or the author's name, if the file was generated by hand.

license

Example:

```
license => [ 'perl_5' ]

license => [ 'apache_2_0', 'mozilla_1_0' ]
```

(Spec 2) [required] {List of one or more License Strings}

One or more licenses that apply to some or all of the files in the distribution. If multiple licenses are listed, the distribution documentation should be consulted to clarify the interpretation of

multiple licenses.

The following list of license strings are valid:

```
string description
------------ ---------------------------------------------
agpl_3 GNU Affero General Public License, Version 3
apache_1_1 Apache Software License, Version 1.1
apache_2_0 Apache License, Version 2.0
artistic_1 Artistic License, (Version 1)
artistic_2 Artistic License, Version 2.0
bsd BSD License (three-clause)
freebsd FreeBSD License (two-clause)
gfdl_1_2 GNU Free Documentation License, Version 1.2
gfdl_1_3 GNU Free Documentation License, Version 1.3
gpl_1 GNU General Public License, Version 1
gpl_2 GNU General Public License, Version 2
gpl_3 GNU General Public License, Version 3
lgpl_2_1 GNU Lesser General Public License, Version 2.1
lgpl_3_0 GNU Lesser General Public License, Version 3.0
mit MIT (aka X11) License
mozilla_1_0 Mozilla Public License, Version 1.0
mozilla_1_1 Mozilla Public License, Version 1.1
openssl OpenSSL License
perl_5 The Perl 5 License (Artistic 1 & GPL 1 or later)
qpl_1_0 Q Public License, Version 1.0
ssleay Original SSLeay License
sun Sun Internet Standards Source License (SISSL)
zlib zlib License
```

The following license strings are also valid and indicate other licensing not described above:

```
string description
------------ ---------------------------------------------
open_source Other Open Source Initiative (OSI) approved license
restricted Requires special permission from copyright holder
unrestricted Not an OSI approved license, but not restricted
unknown License not provided in metadata
```

All other strings are invalid in the license field.

meta-spec

Example:

```
'meta-spec' => {
version => '2',
   url     => 'http://search.cpan.org/perldoc?CPAN::Meta::Spec',
}
```

(Spec 1.2) [required] {Map}

This field indicates the version of the CPAN Meta Spec that should be used to interpret the metadata. Consumers must check this key as soon as possible and abort further metadata processing if the meta-spec version is not supported by the consumer.

The following keys are valid, but only `version` is required.

version
: This subkey gives the integer *Version* of the CPAN Meta Spec against which the document was generated.

url This is a *URL* of the metadata specification document corresponding to the given version. This is strictly for human-consumption and should not impact the interpretation of the document.

For the version 2 spec, either of these are recommended:

- https://metacpan.org/pod/CPAN::Meta::Spec
- http://search.cpan.org/perldoc?CPAN::Meta::Spec

name

Example:

```
name => 'Module-Build'
```

(Spec 1.0) [required] {String}

This field is the name of the distribution. This is often created by taking the "main package" in the distribution and changing `::` to `-`, but the name may be completely unrelated to the packages within the distribution. For example, LWP::UserAgent is distributed as part of the distribution name "libwww-perl".

release_status

Example:

```
release_status => 'stable'
```

(Spec 2) [required] {String}

This field provides the release status of this distribution. If the `version` field contains an underscore character, then `release_status` **must not** be "stable."

The `release_status` field **must** have one of the following values:

stable
: This indicates an ordinary, "final" release that should be indexed by PAUSE or other indexers.

testing
: This indicates a "beta" release that is substantially complete, but has an elevated risk of bugs and requires additional testing. The distribution should not be installed over a stable release without an explicit request or other confirmation from a user. This release status may also be used for "release candidate" versions of a distribution.

unstable
: This indicates an "alpha" release that is under active development, but has been released for early feedback or testing and may be missing features or may have serious bugs. The distribution should not be installed over a stable release without an explicit request or other confirmation from a user.

Consumers **may** use this field to determine how to index the distribution for CPAN or other repositories in addition to or in replacement of heuristics based on version number or file name.

version

Example:

```
version => '0.36'
```

(Spec 1.0) [required] {Version}

This field gives the version of the distribution to which the metadata structure refers.

OPTIONAL FIELDS

description

Example:

```
description => "Module::Build is a system for "
. "building, testing, and installing Perl modules. "
. "It is meant to ... blah blah blah ...",
```

(Spec 2) [optional] {String}

A longer, more complete description of the purpose or intended use of the distribution than the one provided by the `abstract` key.

keywords

Example:

```
keywords => [ qw/ toolchain cpan dual-life / ]
```

(Spec 1.1) [optional] {List of zero or more Strings}

A List of keywords that describe this distribution. Keywords **must not**include whitespace.

no_index

Example:

```
no_index => {
file => [ 'My/Module.pm' ],
directory => [ 'My/Private' ],
package => [ 'My::Module::Secret' ],
namespace => [ 'My::Module::Sample' ],
}
```

(Spec 1.2) [optional] {Map}

This Map describes any files, directories, packages, and namespaces that are private to the packaging or implementation of the distribution and should be ignored by indexing or search tools. Note that this is a list of exclusions, and the spec does not define what to *include* - see "Indexing distributions a la PAUSE" in the implementors notes for more information.

Valid subkeys are as follows:

file A *List* of relative paths to files. Paths **must be**specified with unix con ventions.

directory
: A *List* of relative paths to directories. Paths **must be**specified with unix con ventions.

  [Note: previous editions of the spec had `dir` instead of `directory`]

package
: A *List* of package names.

namespace
: A *List* of package namespaces, where anything below the namespace must be ignored, but *not* the namespace itself.

  In the example above for `no_index`, `My::Module::Sample::Foo` would be ignored, but `My::Module::Sample` would not.

optional_features

Example:

```
optional_features => {
sqlite => {
description => 'Provides SQLite support',
prereqs => {
runtime => {
requires => {
'DBD::SQLite' => '1.25'
}
}
}
}
}
```

(Spec 2) [optional] {Map}

This Map describes optional features with incremental prerequisites. Each key of the `optional_features` Map is a String used to identify the feature and each value is a Map with additional information about the feature. Valid subkeys include:

description
: This is a String describing the feature. Every optional feature should provide a description

prereqs
: This entry is required and has the same structure as that of the `"prereqs"` key. It provides a list of package requirements that must be satisfied for the feature to be supported or enabled.

  There is one crucial restriction: the prereqs of an optional feature **must not**include `configure` phase prereqs.

Consumers **must not**include optional features as prerequisites without explicit instruction from users (whether via interactive prompting, a function parameter or a configuration value, etc.).

If an optional feature is used by a consumer to add additional prerequisites, the consumer should merge the optional feature prerequisites into those given by the `prereqs` key using the same semantics. See ‘‘Merging and Resolving Prerequisites’’ for details on merging prerequisites.

*Suggestion for disuse:*Because there is curren tly no way for a distribution to specify a dependency on an optional feature of another dependency, the use of `optional_feature` is discouraged. Instead, create a separate, installable distribution that ensures the desired feature is available. For example, if `Foo::Bar` has a `Baz` feature, release a separate `Foo-Bar-Baz` distribution that satisfies requirements for the feature.

prereqs

Example:

```
prereqs => {
runtime => {
requires => {
'perl' => '5.006',
'File::Spec' => '0.86',
'JSON' => '2.16',
},
recommends => {
'JSON::XS' => '2.26',
},
suggests => {
'Archive::Tar' => '0',
},
},
build => {
```

```
requires => {
'Alien::SDL' => '1.00',
},
},
test => {
recommends => {
'Test::Deep' => '0.10',
},
}
}
```

(Spec 2) [optional] {Map}

This is a Map that describes all the prerequisites of the distribution. The keys are phases of activity, such as `configure`, `build`, `test` or `runtime`. Values are Maps in which the keys name the type of prerequisite relationship such as `requires`, `recommends`, or `suggests` and the value provides a set of prerequisite relations. The set of relations **must** be specified as a Map of package names to version ranges.

The full definition for this field is given in the ''Prereq Spec'' section.

provides

Example:

```
provides => {
'Foo::Bar' => {
file => 'lib/Foo/Bar.pm',
version => '0.27_02',
},
'Foo::Bar::Blah' => {
file => 'lib/Foo/Bar/Blah.pm',
},
'Foo::Bar::Baz' => {
file => 'lib/Foo/Bar/Baz.pm',
version => '0.3',
},
}
```

(Spec 1.2) [optional] {Map}

This describes all packages provided by this distribution. This information is used by distribution and automation mechanisms like PAUSE, CPAN, metacpan.org and search.cpan.org to build indexes saying in which distribution various packages can be found.

The keys of `provides` are package names that can be found within the distribution. If a package name key is provided, it must have a Map with the following valid subkeys:

file This field is required. It must contain a Unix-style relative file path from the root of the distribution directory to a file that contains or generates the package. It may be given as `META.yml` or `META.json` to claim a package for indexing without needing a `*.pm`.

version
: If it exists, this field must contains a *Version* String for the package. If the package does not have a `$VERSION`, this field must be omitted.

resources

Example:

```
 resources => {
    license     => [ 'http://dev.perl.org/licenses/'
],
    homepage    => 'http://sourceforge.net/projects/module-build',
 bugtracker => {
      web    => 'http://rt.cpan.org/Public/Dist/Display.html?Name=CPAN-Meta',
 mailto => 'meta-bugs@example.com',
 },
 repository => {
 url => 'git://github.com/dagolden/cpan-meta.git',
      web  => 'http://github.com/dagolden/cpan-meta',
 type => 'git',
 },
    x_twitter   => 'http://twitter.com/cpan_linked/',
 }
```

(Spec 2) [optional] {Map}

This field describes resources related to this distribution.

Valid subkeys include:

homepage
: The official home of this project on the web.

license
: A List of *URL*'s that relate to this distribution's license. As with the top-level `license` field, distribution documentation should be consulted to clarify the interpretation of multiple licenses provided here.

bugtracker
: This entry describes the bug tracking system for this distribution. It is a Map with the following valid keys:

```
web - a URL pointing to a web front-end for the bug tracker
mailto - an email address to which bugs can be sent
```

repository
: This entry describes the source control repository for this distribution. It is a Map with the following valid keys:

```
url - a URL pointing to the repository itself
web - a URL pointing to a web front-end for the repository
type - a lowercase string indicating the VCS used
```

Because a url like `http://myrepo.example.com/` is ambiguous as to type, producers should provide a `type` whenever a `url` key is given. The`type` field should be the name of the most common program used to work with the repository, e.g. `git`, `svn`, `cvs`, `darcs`, `bzr` or `hg`.

DEPRECATED FIELDS

build_requires

(Deprecated in Spec 2)[optional] {String}

Replaced by `prereqs`

configure_requires

(Deprecated in Spec 2)[optional] {String}

Replaced by `prereqs`

conflicts

(Deprecated in Spec 2)[optional] {String}

Replaced by `prereqs`

distribution_type

(Deprecated in Spec 2)[optional] {String}

This field indicated 'module' or 'script' but was considered meaningless, since many distributions are hybrids of several kinds of things.

license_uri

(Deprecated in Spec 1.2)[optional] {URL}

Replaced by `license` in `resources`

private

(Deprecated in Spec 1.2)[optional] {Map}

This field has been renamed to ''no_index''.

recommends

(Deprecated in Spec 2)[optional] {String}

Replaced by `prereqs`

requires

(Deprecated in Spec 2)[optional] {String}

Replaced by `prereqs`

VERSION NUMBERS

Version Formats

This section defines the Version type, used by several fields in the CPAN Meta Spec.

Version numbers must be treated as strings, not numbers. For example, `1.200` **must not** be serialized as `1.2`. Version comparison should be delegated to the Perl version module, version 0.80 or newer.

Unless otherwise specified, version numbers **must** appear in one of two formats:

Decimal versions

Decimal versions are regular ''decimal numbers'', with some limitations. They **must** be non-negative and **must** begin and end with a digit. A single underscore **may** be included, but **must** be between two digits. They **must not** use exponen tial notation (''1.23e-2'').

```
version => '1.234' # OK
version => '1.23_04' # OK

version => '1.23_04_05' # Illegal
version => '1.' # Illegal
version => '.1' # Illegal
```

Dotted-integer versions

Dotted-integer (also known as dotted-decimal) versions consist of positive integers separated by full stop characters (i.e. ''dots'', ''periods'' or ''decimal points''). This are equivalent in format to Perl ''v-strings'', with some additional restrictions on form. They must be given in ''normal'' form, which has a leading ''v'' character and at least three integer components. To retain a one-to-one mapping with decimal versions, all components after the first **should** be restricted to the range 0 to 999. The final component **may** be separated by an underscore character instead of a period.

```
version => 'v1.2.3' # OK
version => 'v1.2_3' # OK
version => 'v1.2.3.4' # OK
version => 'v1.2.3_4' # OK
version => 'v2009.10.31' # OK

version => 'v1.2' # Illegal
version => '1.2.3' # Illegal
version => 'v1.2_3_4' # Illegal
version => 'v1.2009.10.31' # Not recommended
```

Version Ranges

Some fields (prereq, optional_features) indicate the particular version(s) of some other module that may be required as a prerequisite. This section details the Version Range type used to provide this information.

The simplest format for a Version Range is just the version number itself, e.g. `2.4`. This means that **at least** version 2.4 must be present. To indicate that **any** version of a prerequisite is okay, even if the prerequisite doesn't define a version at all, use the version `0`.

Alternatively, a version range **may** use the operators < (less than), <= (less than or equal), > (greater than), >= (greater than or equal), == (equal), and != (not equal). For example, the specification `< 2.0` means that any version of the prerequisite less than 2.0 is suitable.

For more complicated situations, version specifications **may** be AND-ed together using commas. The specification `>= 1.2, != 1.5, < 2.0` indicates a version that must be **at least** 1.2, **less than** 2.0, and **not equal to** 1.5.

PREREQUISITES

Prereq Spec

The `prereqs` key in the top-level metadata and within `optional_features` define the relationship between a distribution and other packages. The prereq spec structure is a hierarchical data structure which divides prerequisites into *Phases* of activity in the installation process and *Relationships* that indicate how prerequisites should be resolved.

For example, to specify that `Data::Dumper` is `required` during the `test` phase, this entry would appear in the distribution metadata:

```
prereqs => {
test => {
requires => {
'Data::Dumper' => '2.00'
}
}
}
```

Phases

Requirements for regular use must be listed in the `runtime` phase. Other requirements should be listed in the earliest stage in which they are required and consumers must accumulate and satisfy requirements across phases before executing the activity. For example, `build` requirements must also be available during the `test` phase.

```
before action requirements that must be met
---------------- --------------------------------
perl Build.PL configure
perl Makefile.PL

make configure, runtime, build
Build
```

```
make test configure, runtime, build, test
Build test
```

Consumers that install the distribution must ensure that *runtime* requirements are also installed and may install dependencies from other phases.

```
after action requirements that must be met
---------------- --------------------------------
make install runtime
Build install
```

configure
: The configure phase occurs before any dynamic configuration has been attempted. Libraries required by the configure phase **must** be available for use before the distribution building tool has been executed.

build
: The build phase is when the distribution's source code is compiled (if necessary) and otherwise made ready for installation.

test
: The test phase is when the distribution's automated test suite is run. Any library that is needed only for testing and not for subsequent use should be listed here.

runtime
: The runtime phase refers not only to when the distribution's contents are installed, but also to its continued use. Any library that is a prerequisite for regular use of this distribution should be indicated here.

develop
: The develop phase's prereqs are libraries needed to work on the distribution's source code as its author does. These tools might be needed to build a release tarball, to run author-only tests, or to perform other tasks related to developing new versions of the distribution.

Relationships

requires
: These dependencies **must** be installed for proper completion of the phase.

recommends
: Recommended dependencies are *strongly* encouraged and should be satisfied except in resource constrained environments.

suggests
: These dependencies are optional, but are suggested for enhanced operation of the described distribution.

conflicts
: These libraries cannot be installed when the phase is in operation. This is a very rare situation, and the `conflicts` relationship should be used with great caution, or not at all.

Merging and Resolving Prerequisites

Whenever metadata consumers merge prerequisites, either from different phases or from `optional_features`, they should merged in a way which preserves the intended semantics of the prerequisite structure. Generally, this means concatenating the version specifications using commas, as described in the ''Version Ranges'' section.

Another subtle error that can occur in resolving prerequisites comes from the way that modules in prerequisites are indexed to distribution files on CPAN. When a module is deleted from a distribution, prerequisites calling for that module could indicate an older distribution should be installed, potentially overwriting files from a newer distribution.

For example, as of Oct 31, 2009, the CPAN index file contained these module-distribution mappings:

```
Class::MOP 0.94 D/DR/DROLSKY/Class-MOP-0.94.tar.gz
Class::MOP::Class 0.94 D/DR/DROLSKY/Class-MOP-0.94.tar.gz
Class::MOP::Class::Immutable 0.04 S/ST/STEVAN/Class-MOP-0.36.tar.gz
```

Consider the case where ''Class::MOP'' 0.94 is installed. If a distribution specified ''Class::MOP::Class::Immutable'' as a prerequisite, it could result in Class-MOP-0.36.tar.gz being installed, overwriting any files from Class-MOP-0.94.tar.gz.

Consumers of metadata **should** test whether prerequisites would result in installed module files being ''downgraded'' to an older version and **may** warn users or ignore the prerequisite that would cause such a result.

SERIALIZATION

Distribution metadata should be serialized (as a hashref) as JSON-encoded data and packaged with distributions as the file *META.json*.

In the past, the distribution metadata structure had been packed with distributions as *META.yml*, a file in the YAML Tiny format (for which, see YAML::Tiny). Tools that consume distribution metadata from disk should be capable of loading *META.yml*, but should prefer *META.json* if both are found.

NOTES FOR IMPLEMENTORS

Extracting Version Numbers from Perl Modules

To get the version number from a Perl module, consumers should use the `MM->parse_version($file)` method provided by ExtUtils::MakeMaker or Module::Metadata. For example, for the module given by `$mod`, the version may be retrieved in one of the following ways:

```
# via ExtUtils::MakeMaker
my $file = MM->_installed_file_for_module($mod);
my $version = MM->parse_version($file)
```

The private `_installed_file_for_module` method may be replaced with other methods for locating a module in `@INC`.

```
# via Module::Metadata
my $info = Module::Metadata->new_from_module($mod);
my $version = $info->version;
```

If only a filename is available, the following approach may be used:

```
# via Module::Build
my $info = Module::Metadata->new_from_file($file);
my $version = $info->version;
```

Comparing Version Numbers

The version module provides the most reliable way to compare version numbers in all the various ways they might be provided or might exist within modules. Given two strings containing version numbers, `$v1` and `$v2`, they should be converted to `version` objects before using ordinary comparison operators. For example:

```
use version;
if ( version->new($v1) <=> version->new($v2) ) {
print "Versions are not equal\n";
}
```

If the only comparison needed is whether an installed module is of a sufficiently high version, a direct test may be done using the string form of `eval` and the `use` function. For example, for module `$mod` and version prerequisite `$prereq`:

```
if ( eval "use $mod $prereq (); 1" ) {
print "Module $mod version is OK.\n";
}
```

If the values of `$mod` and `$prereq` have not been scrubbed, however, this presents security implications.

Prerequisites for dynamically configured distributions

When `dynamic_config` is true, it is an error to presume that the prerequisites given in distribution metadata will have any relationship whatsoever to the actual prerequisites of the distribution.

In practice, however, one can generally expect such prerequisites to be one of two things:

- The minimum prerequisites for the distribution, to which dynamic configuration will only add items
- Whatever the distribution configured with on the releaser's machine at release time

The second case often turns out to have identical results to the first case, albeit only by accident.

As such, consumers may use this data for informational analysis, but presenting it to the user as canonical or relying on it as such is invariably the height of folly.

Indexing distributions a la PAUSE

While no_index tells you what must be ignored when indexing, this spec holds no opinion on how you should get your initial candidate list of things to possibly index. For ''normal'' distributions you might consider simply indexing the contents of lib/, but there are many fascinating oddities on CPAN and many dists from the days when it was normal to put the main .pm file in the root of the distribution archive - so PAUSE currently indexes all .pm and .PL files that are not either (a) specifically excluded by no_index (b) in `inc`, `xt`, or `t` directories, or common 'mistake' directories such as `perl5(1)`

Or: If you're trying to be PAUSE-like, make sure you skip `inc`, `xt` and `t` as well as anything marked as no_index.

Also remember: If the META file contains a provides field, you shouldn't be indexing anything in the first place - just use that.

SEE ALSO

- CPAN, <http://www.cpan.org/>
- JSON, <http://json.org/>
- YAML, <http://www.yaml.org/>
- CPAN
- CPANPLUS
- ExtUtils::MakeMaker
- Module::Build
- Module::Install

HISTORY

Ken Williams wrote the original CPAN Meta Spec (also known as the ''META.yml spec'') in 2003 and maintained it through several revisions with input from various members of the community. In 2005, Randy Sims redrafted it from HTML to POD for the version 1.2 release. Ken continued to maintain the spec through version 1.4.

In late 2009, David Golden organized the version 2 proposal review process. David and Ricardo Signes drafted the final version 2 spec in April 2010 based on the version 1.4 spec and patches contributed during the proposal process.

AUTHORS

- David Golden <dagolden@cpan.org>
- Ricardo Signes <rjbs@cpan.org>

COPYRIGHT AND LICENSE