Checkmk extension packages (MKPs)

Create an issue on GitHub ✏️

1. Introduction

Checkmk has a very modular structure, and those with a knowledge of Python programming can extend this structure in many places. Among other things it is possible to extend Checkmk with the following elements:

- Own checks and agent plug-ins, including input masks for the configuration environment.
- Own plug-ins for the Checkmk HW/SW inventory
- Extensions for the GUI (views, dashboards, columns, icons, etc.).
- Definitions of graphs or Perf-O-Meters
- Notification and alert handler scripts (also in shell or other scripting languages).

All of these these extensions can be implemented by placing additional files in the `~/local` directory within the Checkmk site. To manage these extensions, to roll them out in distributed environments and also share them with other users, Checkmk provides its own package format the **Checkmk extension package** — in short **MKP**.

An MKP can include any desired set of extensions — for example, a set of check plug-ins including associated manual pages, threshold configuration environments and associated metric definitions. It can furthermore contain settings for distribution via the Agent Bakery. An MKP has a name, a version number and can be installed or removed with a simple action.

Use a test site to create and customize MKPs, and copy the MKPs to the production-use site for deployment. This will save you from two main potential problems that arise when modified files are not packaged to MKPs in a timely manner:

- During the Checkmk update, locally changed files are overwritten by the latest MKP state (this is exactly what happened to the author of this sentence).
- In the distributed monitoring with central setup ⓘ, you wonder because plug-ins on remote sites behave differently than on the central site, because the remote sites still get the last packaged state.

1.1. The Checkmk Exchange

On the Checkmk Exchange, programmers of plug-ins can provide packages for other Checkmk users and exchange these amongst themselves. From the Exchange you can download and use extensions for free. Please note that packages from the Exchange are shared voluntarily by other users and are without any warranty.

Improperly programmed plug-ins can lead to increased CPU/system loads and memory requirements. In addition, it is possible that an MKP was developed for older versions of Checkmk and thus may not be fully compatible (e.g., with the update from version 1.6.0 to version 2.0.0 Checkmk changed from Python 2 to Python 3). In extreme cases there can be a risk of data loss. We therefore strongly recommend that before using third-party MKPs in a production environment, they should first be installed on a test site.

1.2. Tools for MKPs

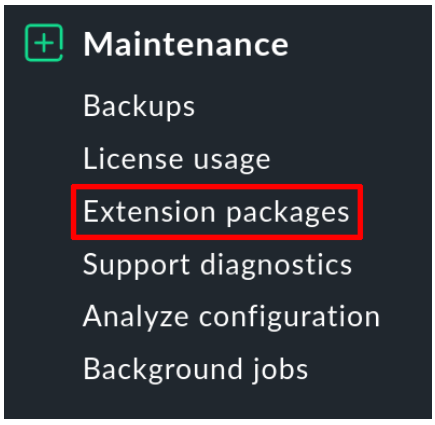
There are two tools for managing MKPs:

- The `mkp` command
- In the Setup menu, the *Extension Packages* item (commercial editions only)

We will now introduce both of these management tools in more detail. They are compatible with each other so that you can use both the command and *Extension Packages* without 'messing anything up'.

2. Managing extension packages via the Setup Menu

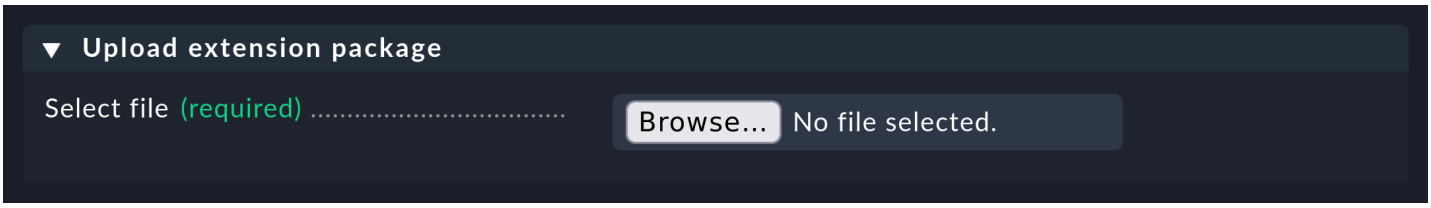
The facility to manage MKPs via the GUI exists exclusively in the commercial editions of Checkmk. In the *Setup* menu you enter the administration of MKPs via *Setup* > *Maintenance* > *Extension packages*. Here you can install, modify or create MKPs:



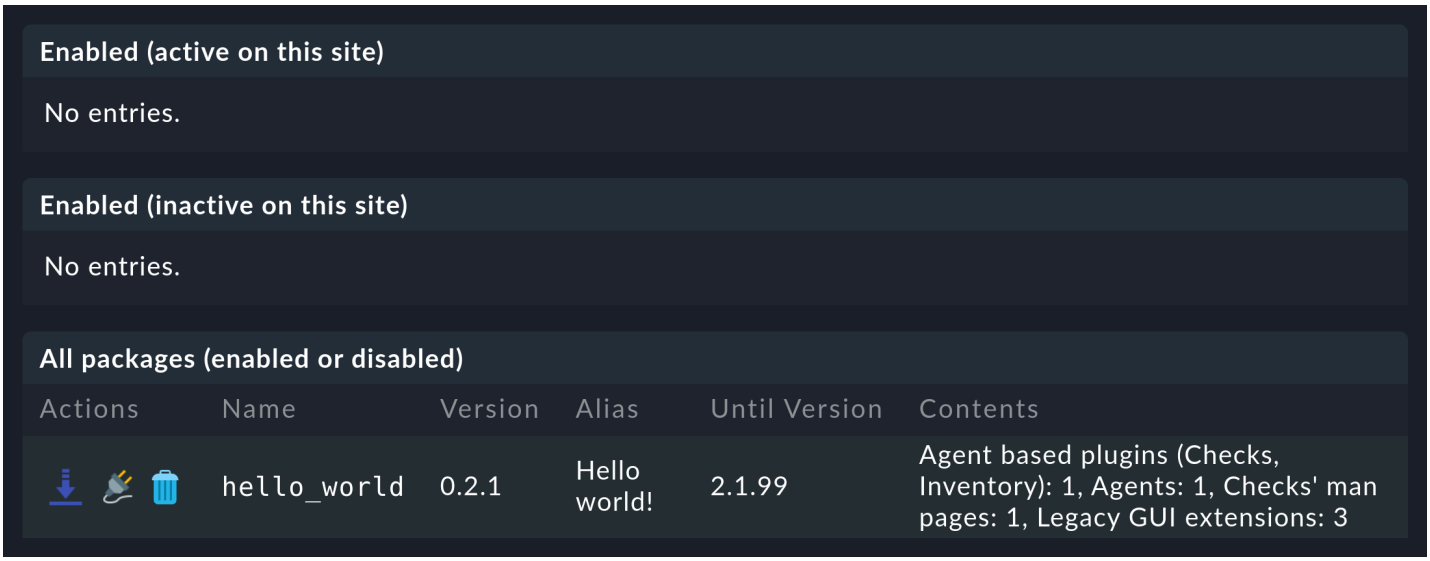
Outdated MKPs can only be installed via the command line. From the *Extension packages*, you can only install (enable **and** activate) MKPs that meet the version requirements. Other MKPs will be enabled but not installed (a corresponding error message will be displayed).

2.1. Adding an MKP

An MKP that you have downloaded from the Exchange, for example, can be uploaded to Checkmk by clicking the *Upload package* button and will then be made available for installation. To do this, the file must be present on the machine that is also running your web browser. The file name of the package must include the `.mkp` extension.

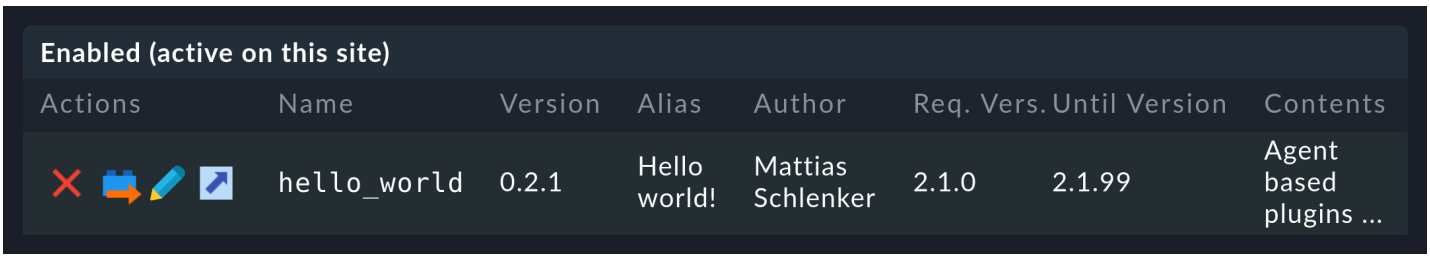


Following an installation, the extension package is initially *available*. but *not active*. It is located under *All packages (enabled or disabled)*:



2.2. Activating an MKP

Only with a click on the plug icon will an available package also be activated. During activation, the files are installed in a folder hierarchy under `~/local/`. The package description file is also placed in `~/var/check_mk/packages/`. After activation, the package will also appear in the list of *enabled and active* MKPs — *Enabled (active on this site)*:



Now perform an activation of changes, after which all functions from the package will be anchored in the system and ready for use.

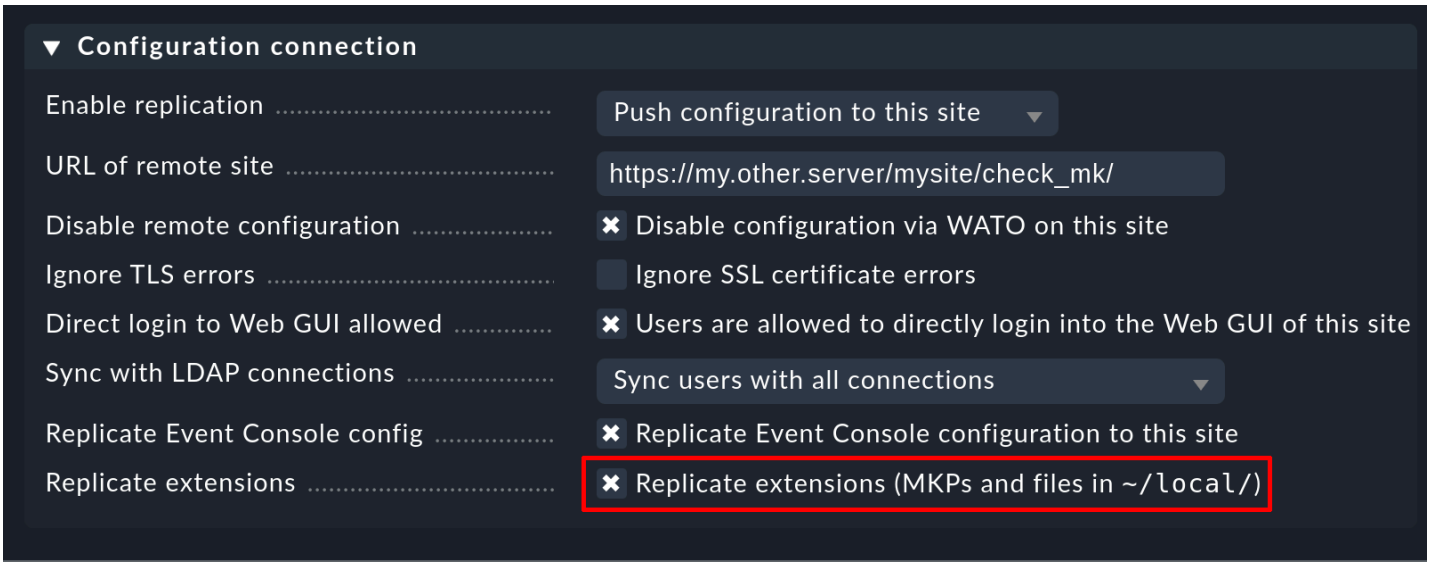
2.3. Deactivating and removing packages

The complete deletion of a package is also done in two stages. With the ✕ button you first deactivate a package in the list of active packages. In this step the installed files are removed, but the MKP is still kept — this step only reverses the *activation*.

Using the 🗑 icon in the list of all packages, you can again delete installed and unused packages. When deleting, the package is deleted and with it the extension is completely removed — i.e. the opposite of *adding a package*.

2.4. MKPs in distributed environments

In the case of a distributed monitoring with central setup ⓘ, it is sufficient to make the packages available on the central site. For each remote site associated with the central site, you can then separately determine whether the customizations should be propagated to that remote site. All you have to do is activate the *Replicate extensions* option. After that, the MKPs and all other changes within the `~/local` directory will also be transferred during a synchronization.

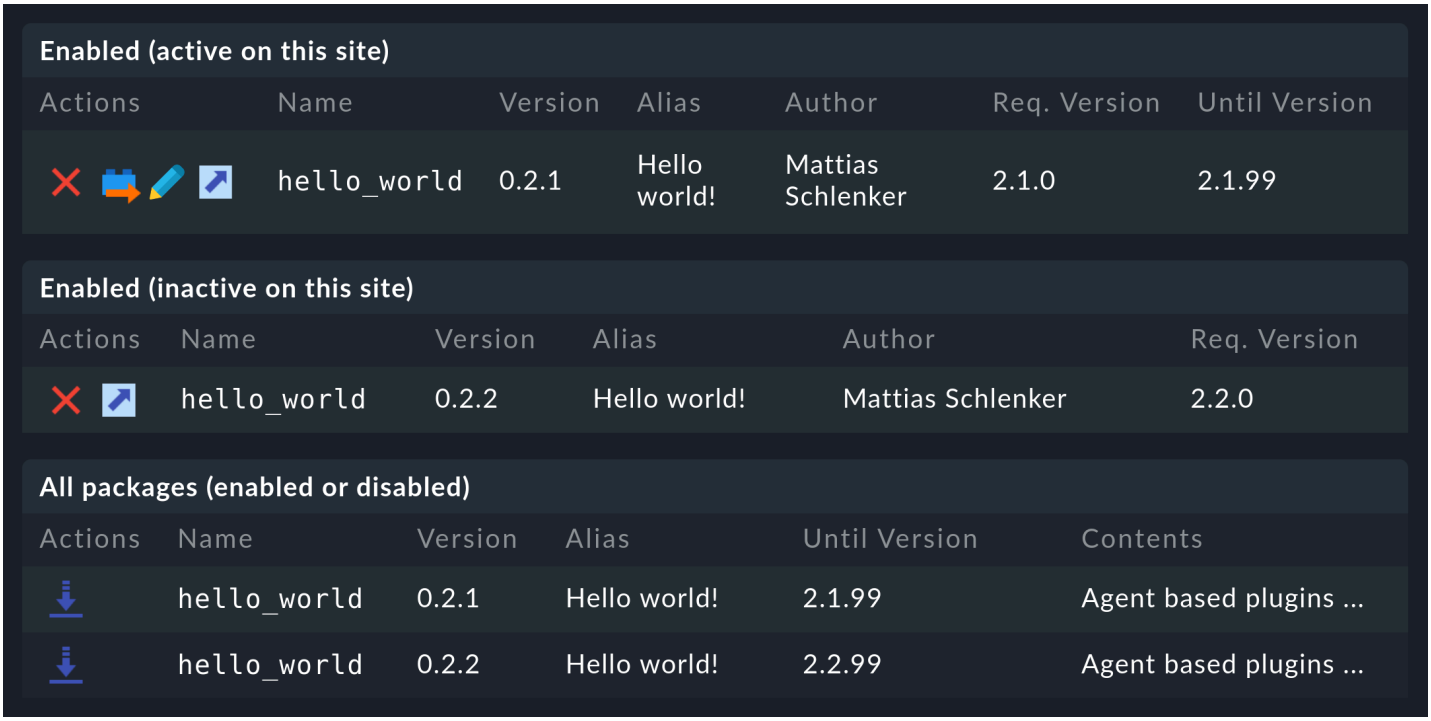


If a particular transfer is not desired, simply turn off the option for this or all sites.

Important: The customizations for the central setup will only be transferred if the *Enable replication* option is set to *Push configuration to this site*.

2.5. A special case: enabled but inactive packages

A special situation is the attempted activation of a package that does not match the Checkmk version used. Such a package that is enabled, but whose activation fails because of an incompatible Checkmk version, will end up in the *Enabled (inactive on this site)* list.



But why install packages that do not match the Checkmk version you are using? There are two good possible reasons:

1. An update of the Checkmk version: You have the possibility of storing packages for both the old and the new versions — when you next perform an update, the newer package will be activated automatically.

2. Distributed monitoring: To facilitate updates, the Checkmk major version of remote sites may be one higher than that of the central site. However, this previously made it difficult to distribute MKPs because these had to be compatible with both major versions. With the ability to unlock mismatched packages, you can keep packages on the central site that match both the source and target versions. The newer version will then be automatically activated during an update.

From the version numbers shown in the above screenshot, you can see that it is a Checkmk 2.1.0 central site that provides packages for remote sites that have already been upgraded to 2.2.0.

3. Managing extension packages via the command line

You can also perform all of the above actions on the command line. The `mkp` command is used for this purpose. If you call it without a subcommand, it shows hints on how to use it. We have abbreviated the output, which is about 50 lines long, to less than half here for clarity:

```
OMD[mysite]:~$ mkp
usage: mkp [-h] [--debug] [--verbose] {find,inspect,show,show-all,files,l

Command line interface for the Checkmk Extension Packages
```

```
options:
  -h, --help            show this help message and exit
  --debug, -d
  --verbose, -v         Be more verbose

available commands:
  {find,inspect,show,show-all,files,list,add,...}
    find                Show information about local files.
    inspect             Show manifest of an MKP file.
    show                Show manifest of a stored package.
    show-all            Show all manifests.
    files               Show all files beloning to a package.
    list                Show a table of all known files, including the dep
    add                 Add an MKP to the collection of managed MKPs.
[...]
```

In the following sections, we will present the most important commands for managing MKPs. A useful command reference can be found as a table at the end of this article.

3.1. Adding an MKP

Adding a package is performed with `mkp add`. To do this, of course, you must first bring the MKP file to the Checkmk server (e.g., with `scp`). Then run the following command:

```
OMD[mysite]:~$ mkp add /tmp/hello_world-0.2.5.mkp
```

You request a list of the available packages with `mkp list`. Following an installation, the extension package is initially *available*, but *not active* — in the list it will have the *Disabled* state:

```
OMD[mysite]:~$ mkp list
Name        Version Title        Author                Req. Version Unti
----------- ------- ------------ --------------------- ------------ -----
hello_world 0.2.5   Hello world! Checkmk knowledge team 2.3.0b1      2.5.9
```

3.2. Activating an MKP

Only with the `enable` subcommand will an available package also be activated. Specifying the version number is only required in the event that the name alone is not unique:

```
OMD[mysite]:~$ mkp enable hello_world 0.2.5
```

In principle, you can only activate MKPs that match the version of your Checkmk installation, even on the command line. If you want to bypass the version restrictions and install (enable **and** activate) the MKP under all conditions, use `--force-install`. This is primarily relevant for developers who need to gradually adapt packages to new Checkmk versions in distributed environments.

```
OMD[mysite]:~$ mkp enable --force-install hello_world 0.2.5
```

Once activated — regardless of whether `force-install` is used — , the files are installed in a directory hierarchy within `~/local/` and the package description file is placed in `~/var/check_mk/packages/`. This results in the package getting the *Enabled (active on this site)* state:

```
OMD[mysite]:~$ mkp list
Name        Version Title        Author                Req. Version Unti
----------- ------- ------------ --------------------- ------------ -----
hello_world 0.2.5   Hello world! Checkmk knowledge team 2.3.0b1      2.5.9
```

Details on an individual package can be obtained with `mkp show`, its actual activation status does not matter:

```
OMD[mysite]:~$ mkp show hello_world 0.2.5
Name:                           hello_world
Version:                        0.2.5
Packaged on Checkmk Version:    2.4.0b1
Required Checkmk Version:       2.3.0b1
Valid until Checkmk version:    2.5.99
Title:                          Hello world!
Author:                         Checkmk knowledge team
Download-URL:                   https://docs.checkmk.com/latest/en/devel_ch
Files:
  Agents
    plugins/hello_world
    windows/plugins/hello_world.cmd
  Additional Checkmk plug-ins by third parties
    hello_world/agent_based/hello_world.py
    hello_world/checkman/hello_world
    hello_world/graphing/helloworld_perfometer_graphing.py
    hello_world/rulesets/ruleset_hello_world.py
    hello_world/rulesets/ruleset_hello_world_bakery.py
  Libraries
    python3/cmk/base/cee/plugins/bakery/hello_world.py
Description:
  This is a very basic plugin with the sole purpose to be used as templat
```

3.3. Deactivating and removing packages

Uninstalling a package is done in two stages. First, the package is disabled with `mkp disable`. This deletes installed files, but still keeps the package — for a possible later reactivation, for example. Again, specifying the version number is only necessary in the event that the package's name alone is not unique:

```
OMD[mysite]:~$ mkp disable hello_world 0.2.5
```

In the package list you will now see the *Disabled* state when you call `mkp list` again:

```
OMD[mysite]:~$ mkp list
Name        Version Title        Author                Req. Version Unti
----------- ------- ------------ --------------------- ------------ -----
hello_world 0.2.5   Hello world! Checkmk knowledge team 2.3.0b1      2.5.9
```

Only `mkp remove` will delete the package irrevocably:

```
OMD[mysite]:~$ mkp remove hello_world 0.2.5
```

3.4. A special case: enabled but inactive packages

A special situation is when a package is installed that does not match the Checkmk version being used:

```
OMD[mysite]:~$ mkp install hello_world-0.3.0.mkp
The package requires Checkmk version 2.5.0, but you have 2.3.0p23 installe
```

You can activate such a package, but the activation will fail because of the incompatible Checkmk version, and the package will get the *Enabled (inactive on this site)* state.

```
OMD[mysite]:~$ mkp list
Name        Version Title        Author                 Req. Version Unti
----------- ------- ------------ ---------------------- ------------ -----
hello_world 0.3.0   Hello world! Checkmk knowledge team 2.5.0b1      2.6.9
hello_world 0.2.5   Hello world! Checkmk knowledge team 2.3.0b1      2.5.9
```

We explained the possible circumstances for choosing to install incompatible packages — i.e. with updates in distributed environments — earlier above in the corresponding Setup section. Similarly to the Setup procedure, use `mkp enable packagename version` to enable a package, or `mkp disable packagename version` to disable an existing enable.

4. MKPs for developers

Most of us who know or learn programming *are "like dwarfs standing on the shoulders of giants to be able to see more and more distant than them"*: It is in Open Source that we can really benefit from the earlier work of others. In the case of Checkmk, this is especially true for extensions, which, in the context of the GPL are works derivative of Checkmk itself, which in turn is subject to the GPL (version 2.0). Specifically, this means that you can customize packages downloaded from the Checkmk Exchange to your heart's content (or simply for current needs).

In the following sections we show — starting from repackaging with minor changes, to resolving an existing (example) package, to compiling unpackaged files — all of the relevant steps presented in the typical sequence in which they are performed.

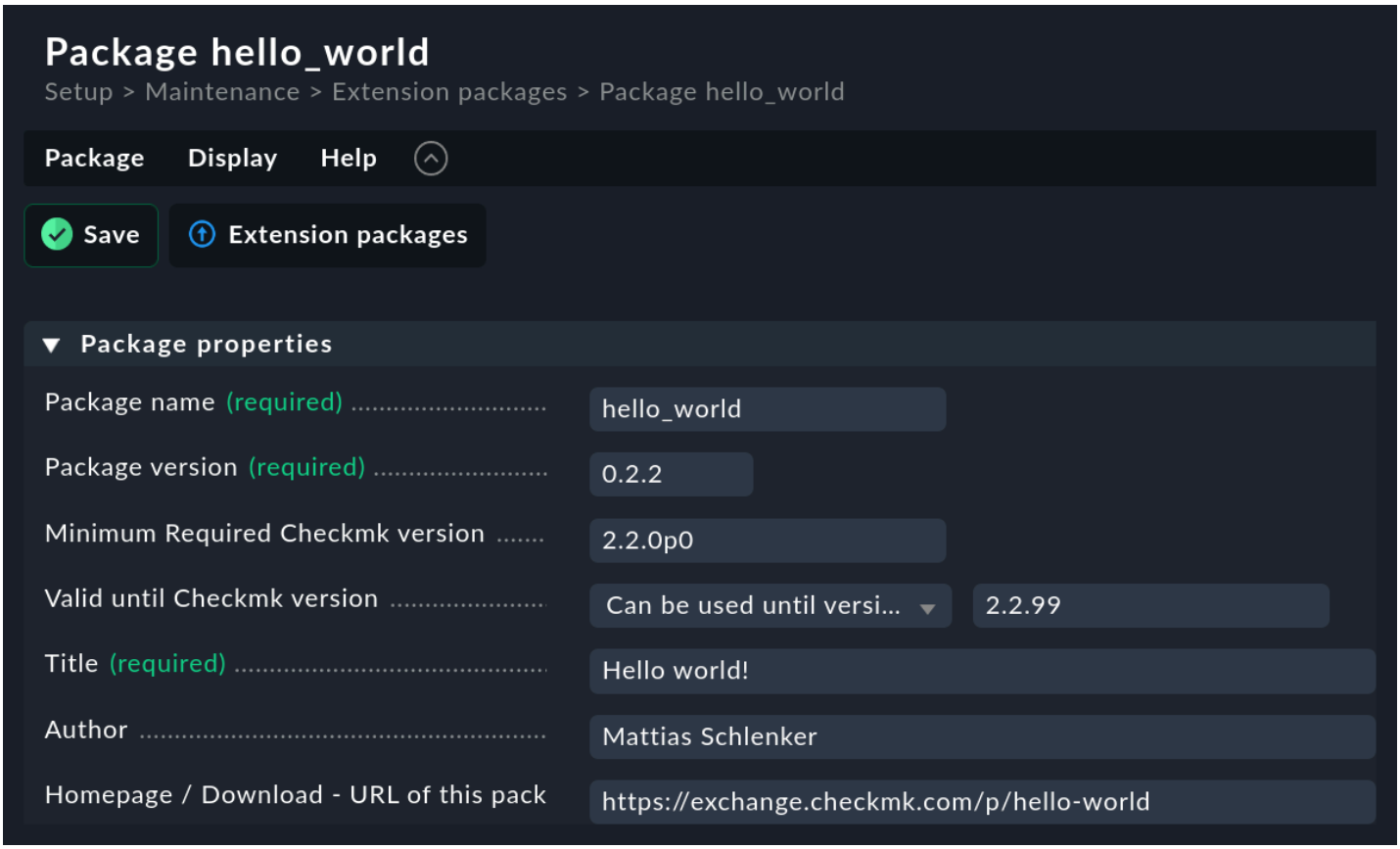
If you are programming or modifying your own plug-ins for Checkmk, see the articles on the existing programming interfaces and the integration into the Agent Bakery.

4.1. Editing packages

The correction of minor errors often makes it necessary to adapt an existing package without changing its structure or name. In this case, it is advisable not only to adapt the existing files stored in the file system, but

also to at least update the package's version number. If changes to Checkmk's APIs require modifications to a package, also adjust the version numbers stored in the package for the minimum and maximum supported versions. In addition, when using the Agent Bakery, the presence of new MKPs triggers the rebuilding of the agent packages.

In the commercial editions use the icon to get to the modifications dialog.



Users of **Checkmk Community** instead take the following two steps via resolve and recreate.

4.2. Unpacking packages

The Setup menu

The unpacking of a package 'frees' the packaged files within `~/local/`, so to speak, and removes only the package description. As a result, the files will be unpackaged and the extensions will remain active. This is the opposite of creating a package from previously unpackaged files.

In practice, you will most likely need to unpackage when you want to customize an extension and later repackage it to include any modifications. For example, you can get started with our Hello world! example, which does nothing actually useful but can serve as a template for your first custom package.

The command line

On the command line, you can release a package with the `mkp release` command. The package to be unpacked must have the *Enabled (active on this site)* state for this to work. The actual extension files are

retained and only the package description is deleted:

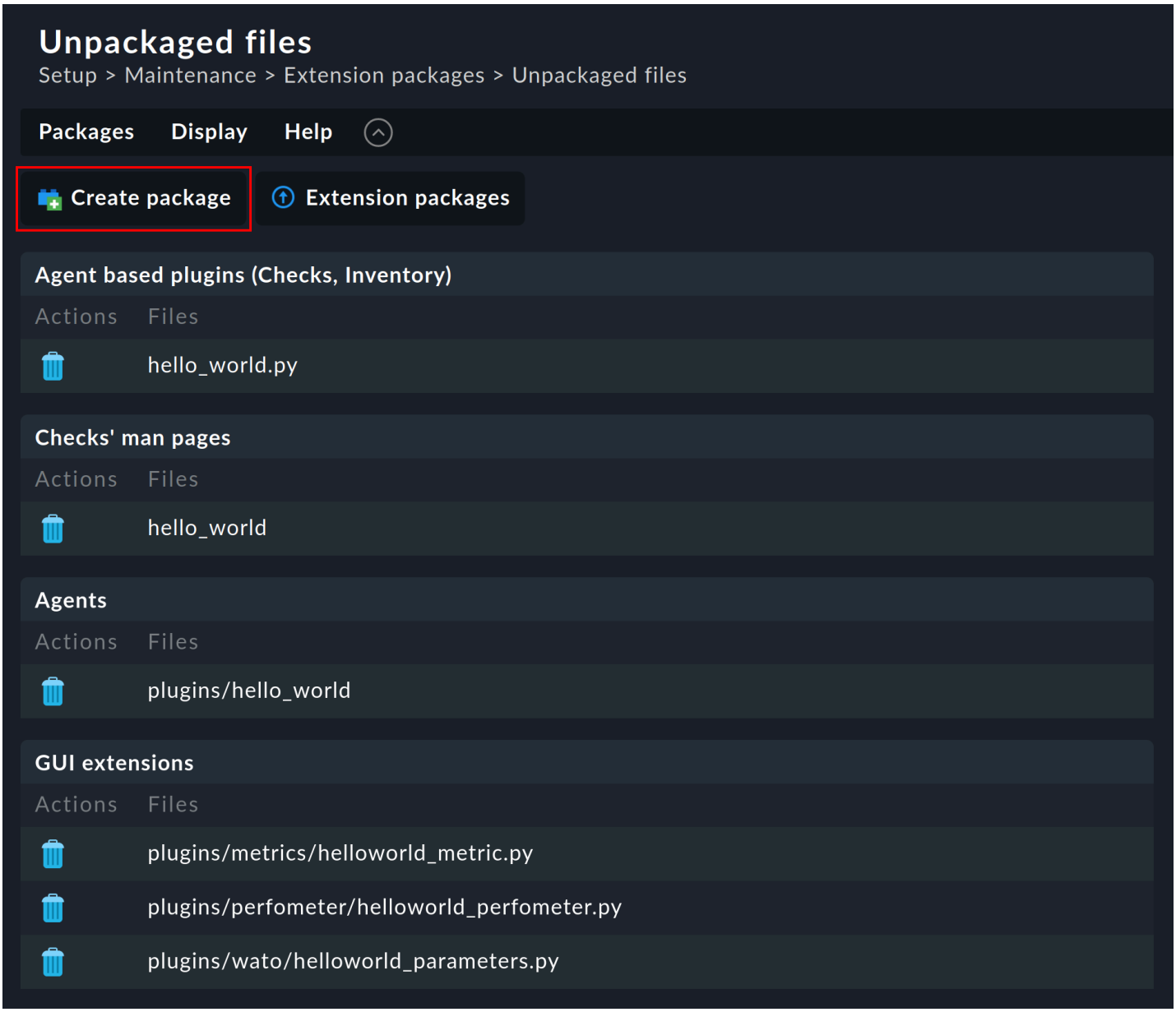
```
OMD[mysite]:~$ mkp release hello_world
```

The original package remains intact and changes its state to *Enabled (inactive on this site)*. It can thus also serve as a backup in case something goes wrong during customization. Then simply delete the 'redundant' files, re-enable the package and start over.

4.3. Finding unpackaged files

The Setup menu

Once the programming or customization work has been completed, it will be necessary to find the existing and added files again. Since these files do not currently belong to any package, they are listed in the Setup under *Unpackaged files*:



List of *Unpackaged files* and the *Create package* button

The command line

The command line equivalent is `mkp find`:

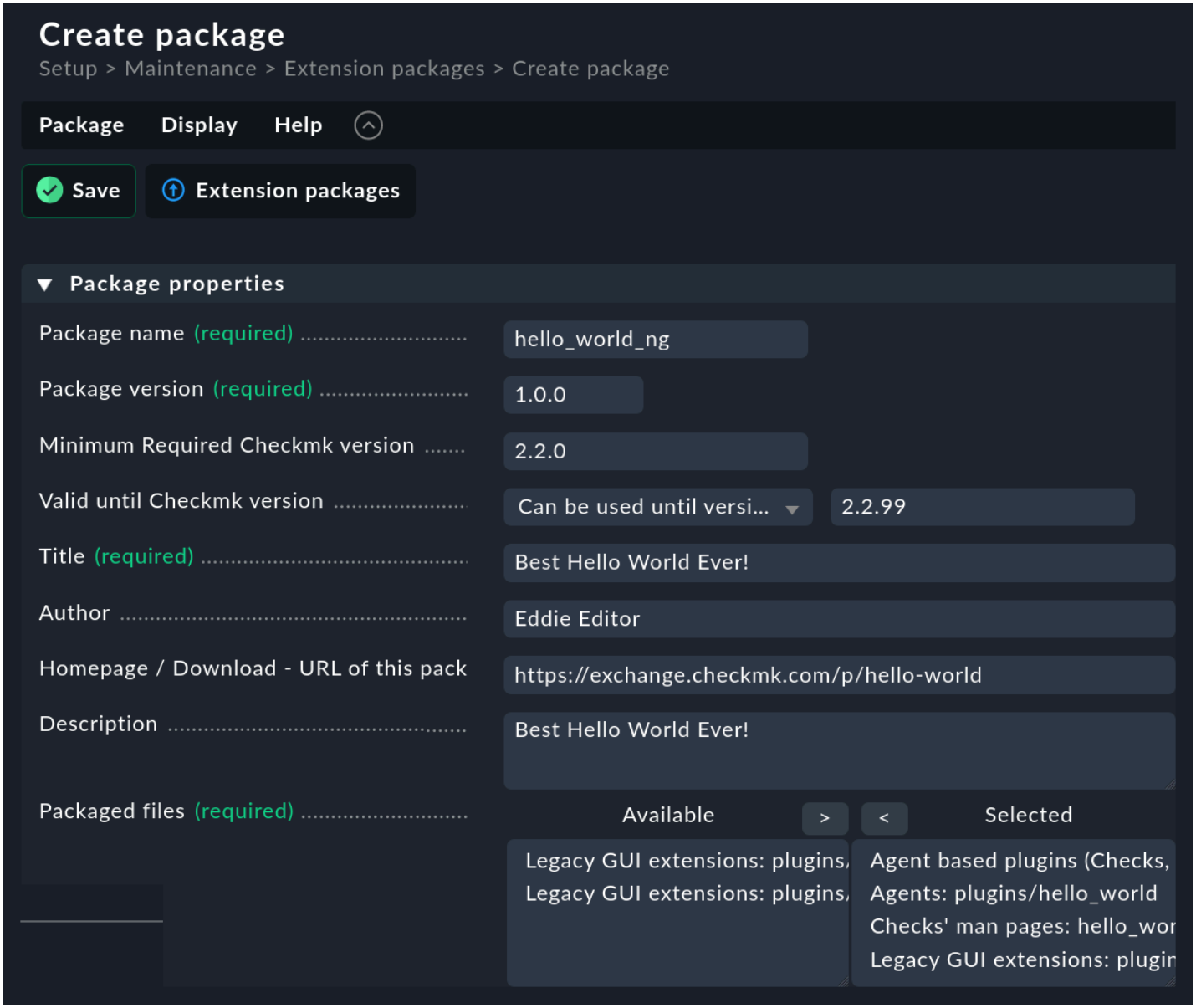
```
OMD[mysite]:~$ mkp find
File                                                    Package Version Pa
------------------------------------------------------- ------- ------- --
hello_world/rulesets/ruleset_hello_world_bakery.py                      Ac
hello_world/agent_based/hello_world.py                                  Ac
hello_world/checkman/hello_world                                        Ac
hello_world/rulesets/ruleset_hello_world.py                             Ac
hello_world/graphing/helloworld_perfometer_graphing.py                  Ac
plugins/hello_world                                                     Ag
windows/plugins/hello_world.cmd                                         Ag
python3/cmk/base/cee/plugins/bakery/hello_world.py                      Lt
```

Delete files that are not needed, or note which files should not be included in the new package. In the next step, the unpackaged files will then be (again) combined into a package.

4.4. Creating packages

The Setup menu

The *Create package* button in the unpackaged files overview takes you to the dialog for creating a new package:



In addition to the obvious details, it is important that you select at least one file to be packaged. Creating a package also creates a package description under `~/var/check_mk/packages/`, which contains general information as well as the list of the included files. The maximum supported Checkmk version is of course difficult to predict without a crystal ball.

Extensions that use the new APIs introduced in Checkmk 2.3.0 are future-proof and will also work up to Checkmk 2.5.0 without any adjustments. You can enter 2.5.99 for *Valid until Checkmk version* as the maximum supported Checkmk version in this case. No statement can be made for the future after that at the time of revision of this article.

You can now download this newly created package as an MKP file via the package list with the  icon — to transfer it to another system or to upload it to the Exchange, for example.

The command line

The procedure for creating MKPs on the command line is analogous to that in the Setup menu. First, use `mkp template` to create a package configuration which (for now) contains all of these files. Specify the desired name for the new package as a parameter:

```
OMD[mysite]:~$ mkp template hello_world_ng
Created 'tmp/check_mk/hello_world_ng.manifest.temp'.
You may now edit it.
Create the package using `mkp package tmp/check_mk/hello_world_ng.manifes
```

You now edit the properties of the package with a text editor:

tmp/check_mk/hello_world_ng.manifest.temp

```
{'author': 'Add your name here',
 'description': 'Please add a description here',
 'download_url': 'https://example.com/hello_world_ng/',
 'files': {'agents': ['plugins/hello_world', 'windows/plugins/hello_world
          'cmk_addons_plugins': ['hello_world/agent_based/hello_world.py
                                 'hello_world/checkman/hello_world',
                                 'hello_world/graphing/helloworld_perfome
                                 'hello_world/rulesets/ruleset_hello_wor
                                 'hello_world/rulesets/ruleset_hello_wor
          'lib': ['python3/cmk/base/cee/plugins/bakery/hello_world.py']}
 'name': 'hello_world_ng',
 'title': 'Title of hello_world_ng',
 'version': '1.0.0',
 'version.min_required': '2.3.0p27',
 'version.packaged': 'cmk-mkp-tool 0.2.0',
 'version.usable_until': None}
```

Edit this file according to your requirements. Pay attention to correct Python syntax — Unicode strings (texts containing non-ASCII characters such as umlauts) must be prefixed with a small `u` for example.

Under the `files` entry, you can remove files that should not be packaged. At `version.min_required` enter the minimum version of Checkmk that is required to be able to use the package.

When done, you can create an MKP file with `mkp package`:

```
OMD[mysite]:~$ mkp package tmp/check_mk/hello_world_ng.manifest.temp
Successfully created hello_world_ng 1.0.0
Successfully wrote package file
Removing packaged files before reinstalling...
```

```
[hello_world_ng 1.0.0]: Removed file local/share/check_mk/agents/plugins/
[hello_world_ng 1.0.0]: Removed file local/share/check_mk/agents/windows/
[hello_world_ng 1.0.0]: Removed file local/lib/python3/cmk_addons/plugins/
[hello_world_ng 1.0.0]: Removed file local/lib/python3/cmk_addons/plugins/
[hello_world_ng 1.0.0]: Removed file local/lib/python3/cmk_addons/plugins/
[hello_world_ng 1.0.0]: Removed file local/lib/python3/cmk_addons/plugins/
[hello_world_ng 1.0.0]: Removed file local/lib/python3/cmk_addons/plugins/
[hello_world_ng 1.0.0]: Removed file local/lib/python3/cmk/base/cee/plugin
[hello_world_ng 1.0.0]: Installing
Successfully installed hello_world_ng 1.0.0
```

Packages are stored under `~/var/check_mk/packages_local`:

```
OMD[mysite]:~$ ll ~/var/check_mk/packages_local/*.mkp
-rw-rw---- 2 mysite mysite 4197 Mar 15 13:37 hello_world_ng-1.0.0.mkp
```

5. The MKP package format

You may want to program and package new extension packages on a development machine, and then transfer the finished package to the Checkmk server for testing. This is quite easy to do since the MKP format is simply a `.tar.gz` file, which in turn contains `.tar` files and manifest files.

Examining the downloaded `hello_world-0.2.5.mkp` reveals the first level of its structure:

```
user@host:~$ tar tvf hello_world-0.2.5.mkp
-rw-r--r-- 0/0            1715 2025-03-07 16:19 info
-rw-r--r-- 0/0            1311 2025-03-07 16:19 info.json
-rw-r--r-- 0/0           10240 2025-03-07 16:19 agents.tar
-rw-r--r-- 0/0           20480 2025-03-07 16:19 cmk_addons_plugins.tar
-rw-r--r-- 0/0           10240 2025-03-07 16:19 lib.tar
```

Unpack the package into a temporary directory, and there you can view the contents of the included tar archives. The file paths are relative to the directory that contains their respective components:

```
user@host:~$ tar tvf cmk_addons_plugins.tar
-rw------- mysite/mysite 3711 2025-03-07 10:59 hello_world/agent_based/he
-rw------- mysite/mysite 1079 2025-03-07 10:59 hello_world/checkman/hello_
-rw------- mysite/mysite 1179 2025-03-07 10:59 hello_world/graphing/hello
-rw------- mysite/mysite 3373 2025-03-07 10:59 hello_world/rulesets/rules
-rw------- mysite/mysite 2634 2025-03-07 10:59 hello_world/rulesets/rules
```

And what about the two manifest files `info` and `info.json`? You have already seen the `info` file and its fields contained in the Python Dict format above. The JSON equivalent `info.json` contains exactly the same fields and values, but has been serialized in the JSON format. If you want to build the package as part of a script, you should input the Python dict file `info` and generate the JSON `info.json` file from this before packaging.

When you repack the archives, be careful not to include file paths that are not part of the folder hierarchy under `~/local`. The top level must contain only the manifests and tar files.

6. Command reference

6.1. Management

Subcommand	Parameter	Function
`add`	File name of the package to be added	Makes a package available, but does not activate it yet.
`enable`	Name of the package (and version number, if applicable)	Activates a package for local use or for distribution to remote sites, depending on version compatibility.
`enable --force-install`	Name of the package (and version number, if applicable)	Activates a package even if there is no version compatibility.
`disable`	Name of the package and version number	Disables a package, which remains available in the file system.
`remove`	Package name and version number	Removes a previously disabled package completely.
`install`	File name of the package to add	This subcommand is deprecated and will be removed soon!
`list`	*none*	Lists all available packages and their activation state.
`inspect`	File name of the package to inspect	Shows information about an uninstalled MKP.

Subcommand	Parameter	Function
`show`	Name of the package (and version number if applicable)	Displays information about an available MKP.
`show-all`	*none*	Displays information about all available MKPs.
`files`	Package name (and version number if applicable)	Lists all files belonging to a package.

6.2. Development

Subcommand	Parameter	Function
`release`	Name of package	Resolves an active package.
`find`	*none*	Lists all files not belonging to any package.
`template`	Name of new package to create	Creates a manifest file as the base for a new package.
`package`	Path to manifest file	Creates an MKP based on the contents of a manifest file.

6.3. Updates

Subcommand	Parameter	Function
`disable-outdated`	*none*	Disables packages that no longer match the Checkmk version after an update.
`update-active`	*none*	Activate packages matching the Checkmk version after an update.

Last modified: Thu, 05 Feb 2026 10:15:33 GMT via commit a64b5bae9