File 1 - browser_check.py:

```
"""
This module is meant to run JupyterLab in a headless browser, making sure
the application launches and starts up without errors.
"""
import asyncio
import inspect
import logging
import os
import shutil
import subprocess
import sys
import time
from concurrent.futures import ThreadPoolExecutor
from os import path as osp
from jupyter_server.serverapp import aliases, flags
from jupyter_server.utils import pathname2url, urljoin
from tornado.ioloop import IOLoop
from tornado.iostream import StreamClosedError
from tornado.websocket import WebSocketClosedError
from traitlets import Bool, Unicode
from .labapp import LabApp, get_app_dir
from .tests.test_app import TestEnv
here = osp.abspath(osp.dirname(__file__))
test_flags = dict(flags)
test_flags["core-mode"] = ({"BrowserApp": {"core_mode": True}}, "Start the app in core mode.")
test_flags["dev-mode"] = ({"BrowserApp": {"dev_mode": True}}, "Start the app in dev mode.")
test_flags["watch"] = ({"BrowserApp": {"watch": True}}, "Start the app in watch mode.")
test_aliases = dict(aliases)
test_aliases["app-dir"] = "BrowserApp.app_dir"
class LogErrorHandler(logging.Handler):
    """A handler that exits with 1 on a logged error."""
    def __init__(self):
        super().__init__(level=logging.ERROR)
        self.errored = False
    def filter(self, record):
        if (
            hasattr(record, "exc_info")
            and record.exc_info is not None
            and isinstance(record.exc_info[1], (StreamClosedError, WebSocketClosedError))
        ):
            return
        return super().filter(record)
    def emit(self, record):
        print(record.msg, file=sys.stderr)
        self.errored = True
def run_test(app, func):
    """Synchronous entry point to run a test function.
    func is a function that accepts an app url as a parameter and returns a result.
    func can be synchronous or asynchronous.  If it is synchronous, it will be run
    in a thread, so asynchronous is preferred.
    """
    IOLoop.current().spawn_callback(run_test_async, app, func)
async def run_test_async(app, func):
    """Run a test against the application.
    func is a function that accepts an app url as a parameter and returns a result.
    func can be synchronous or asynchronous.  If it is synchronous, it will be run
    in a thread, so asynchronous is preferred.
    """
    handler = LogErrorHandler()
```

```
60: (4)                 app.log.addHandler(handler)
61: (4)                 env_patch = TestEnv()
62: (4)                 env_patch.start()
63: (4)                 app.log.info("Running async test")
64: (4)                 if hasattr(app, "browser_open_file"):
65: (8)                     url = urljoin("file:", pathname2url(app.browser_open_file))
66: (4)                 else:
67: (8)                     url = app.display_url
68: (4)                 if inspect.iscoroutinefunction(func):
69: (8)                     test = func(url)
70: (4)                 else:
71: (8)                     app.log.info("Using thread pool executor to run test")
72: (8)                     loop = asyncio.get_event_loop()
73: (8)                     executor = ThreadPoolExecutor()
74: (8)                     task = loop.run_in_executor(executor, func, url)
75: (8)                     test = asyncio.wait([task])
76: (4)                 try:
77: (8)                     await test
78: (4)                 except Exception as e:
79: (8)                     app.log.critical("Caught exception during the test:")
80: (8)                     app.log.error(str(e))
81: (4)                 app.log.info("Test Complete")
82: (4)                 result = 0
83: (4)                 if handler.errored:
84: (8)                     result = 1
85: (8)                     app.log.critical("Exiting with 1 due to errors")
86: (4)                 else:
87: (8)                     app.log.info("Exiting normally")
88: (4)                 app.log.info("Stopping server...")
89: (4)                 try:
90: (8)                     app.http_server.stop()
91: (8)                     app.io_loop.stop()
92: (8)                     env_patch.stop()
93: (4)                 except Exception as e:
94: (8)                     app.log.error(str(e))
95: (8)                     result = 1
96: (4)                 finally:
97: (8)                     time.sleep(2)
98: (8)                     os._exit(result)
99: (0)             async def run_async_process(cmd, **kwargs):
100: (4)                """Run an asynchronous command"""
101: (4)                proc = await asyncio.create_subprocess_exec(*cmd, **kwargs)
102: (4)                stdout, stderr = await proc.communicate()
103: (4)                if proc.returncode != 0:
104: (8)                    raise RuntimeError(str(cmd) + " exited with " + str(proc.returncode))
105: (4)                return stdout, stderr
106: (0)            async def run_browser(url):
107: (4)                """Run the browser test and return an exit code."""
108: (4)                target = osp.join(get_app_dir(), "browser_test")
109: (4)                if not osp.exists(osp.join(target, "node_modules")):
110: (8)                    if not osp.exists(target):
111: (12)                       os.makedirs(osp.join(target))
112: (8)                    await run_async_process(["npm", "init", "-y"], cwd=target)
113: (8)                    await run_async_process(["npm", "install", "playwright@^1.9.2"],
cwd=target)
114: (4)                await run_async_process(["npx", "playwright", "install"], cwd=target)
115: (4)                shutil.copy(osp.join(here, "browser-test.js"), osp.join(target, "browser-
test.js"))
116: (4)                await run_async_process(["node", "browser-test.js", url], cwd=target)
117: (0)            def run_browser_sync(url):
118: (4)                """Run the browser test and return an exit code."""
119: (4)                target = osp.join(get_app_dir(), "browser_test")
120: (4)                if not osp.exists(osp.join(target, "node_modules")):
121: (8)                    os.makedirs(target)
122: (8)                    subprocess.call(["npm", "init", "-y"], cwd=target)  # noqa S603 S607
123: (8)                    subprocess.call(["npm", "install", "playwright@^1.9.2"], cwd=target)
# noqa S603 S607
124: (4)                subprocess.call(["npx", "playwright", "install"], cwd=target)  # noqa S603
S607
```

```
125: (4)                shutil.copy(osp.join(here, "browser-test.js"), osp.join(target, "browser-
test.js"))
126: (4)                return subprocess.check_call(["node", "browser-test.js", url], cwd=target)
# noqa S603 S607
127: (0)            class BrowserApp(LabApp):
128: (4)                """An app the launches JupyterLab and waits for it to start up, checking
for
129: (4)                JS console errors, JS errors, and Python logged errors.
130: (4)                """
131: (4)                name = __name__
132: (4)                open_browser = False
133: (4)                serverapp_config = {"base_url": "/foo/"}
134: (4)                default_url = Unicode("/lab?reset", config=True, help="The default URL to
redirect to from `/`")
135: (4)                ip = "127.0.0.1"
136: (4)                flags = test_flags
137: (4)                aliases = test_aliases
138: (4)                test_browser = Bool(True)
139: (4)                def initialize_settings(self):
140: (8)                    self.settings.setdefault("page_config_data", {})
141: (8)                    self.settings["page_config_data"]["browserTest"] = True
142: (8)                    self.settings["page_config_data"]["buildAvailable"] = False
143: (8)                    self.settings["page_config_data"]["exposeAppInBrowser"] = True
144: (8)                    super().initialize_settings()
145: (4)                def initialize_handlers(self):
146: (8)                    func = run_browser if self.test_browser else lambda url: 0
147: (8)                    if os.name == "nt" and func == run_browser:
148: (12)                       func = run_browser_sync
149: (8)                    run_test(self.serverapp, func)
150: (8)                    super().initialize_handlers()
151: (0)            def _jupyter_server_extension_points():
152: (4)                return [{"module": __name__, "app": BrowserApp}]
153: (0)            def _jupyter_server_extension_paths():
154: (4)                return [{"module": "jupyterlab.browser_check"}]
155: (0)            if __name__ == "__main__":
156: (4)                skip_options = ["--no-browser-test", "--no-chrome-test"]
157: (4)                for option in skip_options:
158: (8)                    if option in sys.argv:
159: (12)                       BrowserApp.test_browser = False
160: (12)                       sys.argv.remove(option)
161: (4)                BrowserApp.launch_instance()

----------------------------------------

File 2 - commands.py:

1: (0)              """JupyterLab command handler"""
2: (0)              import contextlib
3: (0)              import errno
4: (0)              import hashlib
5: (0)              import itertools
6: (0)              import json
7: (0)              import logging
8: (0)              import os
9: (0)              import os.path as osp
10: (0)             import re
11: (0)             import shutil
12: (0)             import site
13: (0)             import stat
14: (0)             import subprocess
15: (0)             import sys
16: (0)             import tarfile
17: (0)             from copy import deepcopy
18: (0)             from dataclasses import dataclass
19: (0)             from glob import glob
20: (0)             from pathlib import Path
21: (0)             from tempfile import TemporaryDirectory
22: (0)             from threading import Event
23: (0)             from typing import FrozenSet, Optional
```

```
from urllib.error import URLError
from urllib.request import Request, quote, urljoin, urlopen
from jupyter_core.paths import jupyter_config_dir
from jupyter_server.extension.serverextension import GREEN_ENABLED, GREEN_OK, RED_DISABLED, RED_X
from jupyterlab_server.config import (
    get_allowed_levels,
    get_federated_extensions,
    get_package_url,
    get_page_config,
    get_static_page_config,
    write_page_config,
)
from jupyterlab_server.process import Process, WatchHelper, list2cmdline, which
from packaging.version import Version
from traitlets import Bool, HasTraits, Instance, List, Unicode, default
from jupyterlab._version import __version__
from jupyterlab.coreconfig import CoreConfig
from jupyterlab.jlpmapp import HERE, YARN_PATH
from jupyterlab.semver import Range, gt, gte, lt, lte, make_semver
WEBPACK_EXPECT = re.compile(r".*theme-light-extension/style/theme.css")
REPO_ROOT = osp.abspath(osp.join(HERE, ".."))
DEV_DIR = osp.join(REPO_ROOT, "dev_mode")
PIN_PREFIX = "pin@"
YARN_DEFAULT_REGISTRY = "https://registry.yarnpkg.com"
class ProgressProcess(Process):
    def __init__(self, cmd, logger=None, cwd=None, kill_event=None, env=None):
        """Start a subprocess that can be run asynchronously.
        Parameters
        ----------
        cmd: list
            The command to run.
        logger: :class:`~logger.Logger`, optional
            The logger instance.
        cwd: string, optional
            The cwd of the process.
        kill_event: :class:`~threading.Event`, optional
            An event used to kill the process operation.
        env: dict, optional
            The environment for the process.
        """
        if not isinstance(cmd, (list, tuple)):
            msg = "Command must be given as a list"
            raise ValueError(msg)
        if kill_event and kill_event.is_set():
            msg = "Process aborted"
            raise ValueError(msg)
        self.logger = _ensure_logger(logger)
        self._last_line = ""
        self.cmd = cmd
        self.logger.debug(f"> {list2cmdline(cmd)}")
        self.proc = self._create_process(
            cwd=cwd,
            env=env,
            stderr=subprocess.STDOUT,
            stdout=subprocess.PIPE,
            universal_newlines=True,
            encoding="utf-8",
        )
        self._kill_event = kill_event or Event()
        Process._procs.add(self)
    def wait(self):
        cache = []
        proc = self.proc
        kill_event = self._kill_event
        spinner = itertools.cycle(["-", "\\", "|", "/"])
        while proc.poll() is None:
            sys.stdout.write(next(spinner))  # write the next character
            sys.stdout.flush()  # flush stdout buffer (actual character display)
            sys.stdout.write("\b")
            if kill_event.is_set():
                self.terminate()
                msg = "Process was aborted"
                raise ValueError(msg)
            try:
                out, _ = proc.communicate(timeout=0.1)
                cache.append(out)
            except subprocess.TimeoutExpired:
                continue
        self.logger.debug("\n".join(cache))
        sys.stdout.flush()
        return self.terminate()
def pjoin(*args):
    """Join paths to create a real path."""
    return osp.abspath(osp.join(*args))
def get_user_settings_dir():
    """Get the configured JupyterLab user settings directory."""
    settings_dir = os.environ.get("JUPYTERLAB_SETTINGS_DIR")
    settings_dir = settings_dir or pjoin(jupyter_config_dir(), "lab", "user-settings")
    return osp.abspath(settings_dir)
def get_workspaces_dir():
    """Get the configured JupyterLab workspaces directory."""
    workspaces_dir = os.environ.get("JUPYTERLAB_WORKSPACES_DIR")
    workspaces_dir = workspaces_dir or pjoin(jupyter_config_dir(), "lab", "workspaces")
    return osp.abspath(workspaces_dir)
def get_app_dir():
    """Get the configured JupyterLab app directory."""
    if os.environ.get("JUPYTERLAB_DIR"):
        return str(Path(os.environ["JUPYTERLAB_DIR"]).resolve())
    app_dir = pjoin(sys.prefix, "share", "jupyter", "lab")
    if hasattr(site, "getuserbase"):
        site.getuserbase()
    userbase = getattr(site, "USER_BASE", None)
    if HERE.startswith(userbase) and not app_dir.startswith(userbase):
        app_dir = pjoin(userbase, "share", "jupyter", "lab")
    elif (
        sys.prefix.startswith("/usr")
        and not osp.exists(app_dir)
        and osp.exists("/usr/local/share/jupyter/lab")
    ):
        app_dir = "/usr/local/share/jupyter/lab"
    return str(Path(app_dir).resolve())
def dedupe_yarn(path, logger=None):
    """`yarn-deduplicate` with the `fewer` strategy to minimize total
    packages installed in a given staging directory
    This means a extension (or dependency) _could_ cause a downgrade of an
    version expected at publication time, but core should aggressively set
    pins above, for example, known-bad versions
    """
    had_dupes = (
        ProgressProcess(
            [
                "node",
                YARN_PATH,
                "dlx",
                "yarn-berry-deduplicate",
                "-s",
                "fewerHighest",
                "--fail",
            ],
            cwd=path,
            logger=logger,
        ).wait()
        != 0
(4)                 )
(4)                 if had_dupes:
(8)                     yarn_proc = ProgressProcess(["node", YARN_PATH], cwd=path, logger=logger)
(8)                     yarn_proc.wait()
(0)             def ensure_node_modules(cwd, logger=None):
(4)                 """Ensure that node_modules is up to date.
(4)                 Returns true if the node_modules was updated.
(4)                 """
(4)                 logger = _ensure_logger(logger)
(4)                 yarn_proc = ProgressProcess(
(8)                     ["node", YARN_PATH, "--immutable", "--immutable-cache"], cwd=cwd, logger=logger
(4)                 )
(4)                 ret = yarn_proc.wait()
(4)                 if ret != 0:
(8)                     yarn_proc = ProgressProcess(["node", YARN_PATH], cwd=cwd, logger=logger)
(8)                     yarn_proc.wait()
(8)                     dedupe_yarn(REPO_ROOT, logger)
(4)                 return ret != 0
(0)             def ensure_dev(logger=None):
(4)                 """Ensure that the dev assets are available."""
(4)                 logger = _ensure_logger(logger)
(4)                 target = pjoin(DEV_DIR, "static")
(4)                 if ensure_node_modules(REPO_ROOT, logger) or not osp.exists(target):
(8)                     yarn_proc = ProgressProcess(["node", YARN_PATH, "build"], cwd=REPO_ROOT, logger=logger)
(8)                     yarn_proc.wait()
(0)             def ensure_core(logger=None):
(4)                 """Ensure that the core assets are available."""
(4)                 staging = pjoin(HERE, "staging")
(4)                 logger = _ensure_logger(logger)
(4)                 target = pjoin(HERE, "static", "index.html")
(4)                 if not osp.exists(target):
(8)                     ensure_node_modules(staging, logger)
(8)                     yarn_proc = ProgressProcess(["node", YARN_PATH, "build"], cwd=staging, logger=logger)
(8)                     yarn_proc.wait()
(0)             def ensure_app(app_dir):
(4)                 """Ensure that an application directory is available.
(4)                 If it does not exist, return a list of messages to prompt the user.
(4)                 """
(4)                 if osp.exists(pjoin(app_dir, "static", "index.html")):
(8)                     return
(4)                 msgs = [
(8)                     'JupyterLab application assets not found in "%s"' % app_dir,
(8)                     "Please run `jupyter lab build` or use a different app directory",
(4)                 ]
(4)                 return msgs
(0)             def watch_packages(logger=None):
(4)                 """Run watch mode for the source packages.
(4)                 Parameters
(4)                 ----------
(4)                 logger: :class:`~logger.Logger`, optional
(8)                     The logger instance.
(4)                 Returns
(4)                 -------
(4)                 A list of `WatchHelper` objects.
(4)                 """
(4)                 logger = _ensure_logger(logger)
(4)                 ensure_node_modules(REPO_ROOT, logger)
(4)                 ts_dir = osp.abspath(osp.join(REPO_ROOT, "packages", "metapackage"))
(4)                 ts_regex = r".* Found 0 errors\. Watching for file changes\."
(4)                 ts_proc = WatchHelper(
(8)                     ["node", YARN_PATH, "run", "watch"], cwd=ts_dir, logger=logger, startup_regex=ts_regex
(4)                 )
(4)                 return [ts_proc]
```

```
def watch_dev(logger=None):
    """Run watch mode in a given directory.
    Parameters
    ----------
    logger: :class:`~logger.Logger`, optional
        The logger instance.
    Returns
    -------
    A list of `WatchHelper` objects.
    """
    logger = _ensure_logger(logger)
    package_procs = watch_packages(logger)
    wp_proc = WatchHelper(
        ["node", YARN_PATH, "run", "watch"],
        cwd=DEV_DIR,
        logger=logger,
        startup_regex=WEBPACK_EXPECT,
    )
    return [*package_procs, wp_proc]
class AppOptions(HasTraits):
    """Options object for build system"""
    def __init__(self, logger=None, core_config=None, **kwargs):
        if core_config is not None:
            kwargs["core_config"] = core_config
        if logger is not None:
            kwargs["logger"] = logger
        if "app_dir" in kwargs and not kwargs["app_dir"]:
            kwargs.pop("app_dir")
        super().__init__(**kwargs)
    app_dir = Unicode(help="The application directory")
    use_sys_dir = Bool(
        True,
        help=("Whether to shadow the default app_dir if that is set to a non-default value"),
    )
    logger = Instance(logging.Logger, help="The logger to use")
    core_config = Instance(CoreConfig, help="Configuration for core data")
    kill_event = Instance(Event, args=(), help="Event for aborting call")
    labextensions_path = List(
        Unicode(), help="The paths to look in for prebuilt JupyterLab extensions"
    )
    registry = Unicode(help="NPM packages registry URL")
    splice_source = Bool(False, help="Splice source packages into app directory.")
    skip_full_build_check = Bool(
        False,
        help=(
            "If true, perform only a quick check that the lab build is up to date."
            " If false, perform a thorough check, which verifies extension contents."
        ),
    )
    verbose = Bool(False, help="Increase verbosity level.")
    @default("logger")
    def _default_logger(self):
        return logging.getLogger("jupyterlab")
    @default("app_dir")
    def _default_app_dir(self):
        return get_app_dir()
    @default("core_config")
    def _default_core_config(self):
        return CoreConfig()
    @default("registry")
    def _default_registry(self):
        config = _yarn_config(self.logger)["yarn config"]
        return config.get("registry", YARN_DEFAULT_REGISTRY)
def _ensure_options(options):
```

```
284: (4)                """Helper to use deprecated kwargs for AppOption"""
285: (4)                if options is None:
286: (8)                    return AppOptions()
287: (4)                elif issubclass(options.__class__, AppOptions):
288: (8)                    return options
289: (4)                else:
290: (8)                    return AppOptions(**options)
291: (0)            def watch(app_options=None):
292: (4)                """Watch the application.
293: (4)                Parameters
294: (4)                ----------
295: (4)                app_options: :class:`AppOptions`, optional
296: (8)                    The application options.
297: (4)                Returns
298: (4)                -------
299: (4)                A list of processes to run asynchronously.
300: (4)                """
301: (4)                app_options = _ensure_options(app_options)
302: (4)                _node_check(app_options.logger)
303: (4)                handler = _AppHandler(app_options)
304: (4)                package_procs = watch_packages(app_options.logger) if
app_options.splice_source else []
305: (4)                return package_procs + handler.watch()
306: (0)            def install_extension(extension, app_options=None, pin=None):
307: (4)                """Install an extension package into JupyterLab.
308: (4)                The extension is first validated.
309: (4)                Returns `True` if a rebuild is recommended, `False` otherwise.
310: (4)                """
311: (4)                app_options = _ensure_options(app_options)
312: (4)                _node_check(app_options.logger)
313: (4)                handler = _AppHandler(app_options)
314: (4)                return handler.install_extension(extension, pin=pin)
315: (0)            def uninstall_extension(name=None, app_options=None, all_=False):
316: (4)                """Uninstall an extension by name or path.
317: (4)                Returns `True` if a rebuild is recommended, `False` otherwise.
318: (4)                """
319: (4)                app_options = _ensure_options(app_options)
320: (4)                _node_check(app_options.logger)
321: (4)                handler = _AppHandler(app_options)
322: (4)                if all_ is True:
323: (8)                    return handler.uninstall_all_extensions()
324: (4)                return handler.uninstall_extension(name)
325: (0)            def update_extension(name=None, all_=False, app_dir=None, app_options=None):
326: (4)                """Update an extension by name, or all extensions.
327: (4)                Either `name` must be given as a string, or `all_` must be `True`.
328: (4)                If `all_` is `True`, the value of `name` is ignored.
329: (4)                Returns `True` if a rebuild is recommended, `False` otherwise.
330: (4)                """
331: (4)                app_options = _ensure_options(app_options)
332: (4)                _node_check(app_options.logger)
333: (4)                handler = _AppHandler(app_options)
334: (4)                if all_ is True:
335: (8)                    return handler.update_all_extensions()
336: (4)                return handler.update_extension(name)
337: (0)            def clean(app_options=None):
338: (4)                """Clean the JupyterLab application directory."""
339: (4)                app_options = _ensure_options(app_options)
340: (4)                logger = app_options.logger
341: (4)                app_dir = app_options.app_dir
342: (4)                logger.info("Cleaning %s...", app_dir)
343: (4)                if app_dir == pjoin(HERE, "dev"):
344: (8)                    msg = "Cannot clean the dev app"
345: (8)                    raise ValueError(msg)
346: (4)                if app_dir == pjoin(HERE, "core"):
347: (8)                    msg = "Cannot clean the core app"
348: (8)                    raise ValueError(msg)
349: (4)                if getattr(app_options, "all", False):
350: (8)                    logger.info("Removing everything in %s...", app_dir)
351: (8)                    _rmtree_star(app_dir, logger)
```

```
352: (4)                    else:
353: (8)                        possible_targets = ["extensions", "settings", "staging", "static"]
354: (8)                        targets = [t for t in possible_targets if getattr(app_options, t)]
355: (8)                        for name in targets:
356: (12)                           target = pjoin(app_dir, name)
357: (12)                           if osp.exists(target):
358: (16)                               logger.info("Removing %s...", name)
359: (16)                               _rmtree(target, logger)
360: (12)                           else:
361: (16)                               logger.info("%s not present, skipping...", name)
362: (4)                    logger.info("Success!")
363: (4)                    if getattr(app_options, "all", False) or getattr(app_options, "extensions", False):
364: (8)                        logger.info("All of your extensions have been removed, and will need to be reinstalled")
365: (0)                def build(
366: (4)                    name=None,
367: (4)                    version=None,
368: (4)                    static_url=None,
369: (4)                    kill_event=None,
370: (4)                    clean_staging=False,
371: (4)                    app_options=None,
372: (4)                    production=True,
373: (4)                    minimize=True,
374: (0)                ):
375: (4)                    """Build the JupyterLab application."""
376: (4)                    app_options = _ensure_options(app_options)
377: (4)                    _node_check(app_options.logger)
378: (4)                    handler = _AppHandler(app_options)
379: (4)                    return handler.build(
380: (8)                        name=name,
381: (8)                        version=version,
382: (8)                        static_url=static_url,
383: (8)                        production=production,
384: (8)                        minimize=minimize,
385: (8)                        clean_staging=clean_staging,
386: (4)                    )
387: (0)                def get_app_info(app_options=None):
388: (4)                    """Get a dictionary of information about the app."""
389: (4)                    handler = _AppHandler(app_options)
390: (4)                    handler._ensure_disabled_info()
391: (4)                    return handler.info
392: (0)                def enable_extension(extension, app_options=None, level="sys_prefix"):
393: (4)                    """Enable a JupyterLab extension/plugin.
394: (4)                    Returns `True` if a rebuild is recommended, `False` otherwise.
395: (4)                    """
396: (4)                    handler = _AppHandler(app_options)
397: (4)                    return handler.toggle_extension(extension, False, level=level)
398: (0)                def disable_extension(extension, app_options=None, level="sys_prefix"):
399: (4)                    """Disable a JupyterLab extension/plugin.
400: (4)                    Returns `True` if a rebuild is recommended, `False` otherwise.
401: (4)                    """
402: (4)                    handler = _AppHandler(app_options)
403: (4)                    return handler.toggle_extension(extension, True, level=level)
404: (0)                def check_extension(extension, installed=False, app_options=None):
405: (4)                    """Check if a JupyterLab extension is enabled or disabled."""
406: (4)                    handler = _AppHandler(app_options)
407: (4)                    return handler.check_extension(extension, installed)
408: (0)                def lock_extension(extension, app_options=None, level="sys_prefix"):
409: (4)                    """Lock a JupyterLab extension/plugin."""
410: (4)                    handler = _AppHandler(app_options)
411: (4)                    return handler.toggle_extension_lock(extension, True, level=level)
412: (0)                def unlock_extension(extension, app_options=None, level="sys_prefix"):
413: (4)                    """Unlock a JupyterLab extension/plugin."""
414: (4)                    handler = _AppHandler(app_options)
415: (4)                    return handler.toggle_extension_lock(extension, False, level=level)
416: (0)                def build_check(app_options=None):
417: (4)                    """Determine whether JupyterLab should be built.
418: (4)                    Returns a list of messages.
```

```
419: (4)                """
420: (4)                app_options = _ensure_options(app_options)
421: (4)                _node_check(app_options.logger)
422: (4)                handler = _AppHandler(app_options)
423: (4)                return handler.build_check()
424: (0)            def list_extensions(app_options=None):
425: (4)                """List the extensions."""
426: (4)                handler = _AppHandler(app_options)
427: (4)                return handler.list_extensions()
428: (0)            def link_package(path, app_options=None):
429: (4)                """Link a package against the JupyterLab build.
430: (4)                Returns `True` if a rebuild is recommended, `False` otherwise.
431: (4)                """
432: (4)                handler = _AppHandler(app_options)
433: (4)                return handler.link_package(path)
434: (0)            def unlink_package(package, app_options=None):
435: (4)                """Unlink a package from JupyterLab by path or name.
436: (4)                Returns `True` if a rebuild is recommended, `False` otherwise.
437: (4)                """
438: (4)                handler = _AppHandler(app_options)
439: (4)                return handler.unlink_package(package)
440: (0)            def get_app_version(app_options=None):
441: (4)                """Get the application version."""
442: (4)                handler = _AppHandler(app_options)
443: (4)                return handler.info["version"]
444: (0)            def get_latest_compatible_package_versions(names, app_options=None):
445: (4)                """Get the latest compatible version of a list of packages."""
446: (4)                handler = _AppHandler(app_options)
447: (4)                return handler.latest_compatible_package_versions(names)
448: (0)            def read_package(target):
449: (4)                """Read the package data in a given target tarball."""
450: (4)                tar = tarfile.open(target, "r")
451: (4)                f = tar.extractfile("package/package.json")
452: (4)                data = json.loads(f.read().decode("utf8"))
453: (4)                data["jupyterlab_extracted_files"] = [f.path[len("package/") :] for f in
tar.getmembers()]
454: (4)                tar.close()
455: (4)                return data
456: (0)            class _AppHandler:
457: (4)                def __init__(self, options):
458: (8)                    """Create a new _AppHandler object"""
459: (8)                    options = _ensure_options(options)
460: (8)                    self._options = options
461: (8)                    self.app_dir = options.app_dir
462: (8)                    self.sys_dir = get_app_dir() if options.use_sys_dir else self.app_dir
463: (8)                    self.logger = options.logger
464: (8)                    self.core_data = deepcopy(options.core_config._data)
465: (8)                    self.labextensions_path = options.labextensions_path
466: (8)                    self.verbose = options.verbose
467: (8)                    self.kill_event = options.kill_event
468: (8)                    self.registry = options.registry
469: (8)                    self.skip_full_build_check = options.skip_full_build_check
470: (8)                    self.info = self._get_app_info()
471: (8)                    try:
472: (12)                       self._maybe_mirror_disabled_in_locked(level="sys_prefix")
473: (8)                    except (PermissionError, OSError):
474: (12)                       try:
475: (16)                           self.logger.info(
476: (20)                               "`sys_prefix` level settings are read-only, using `user`
level for migration to `lockedExtensions`"
477: (16)                           )
478: (16)                           self._maybe_mirror_disabled_in_locked(level="user")
479: (12)                       except (PermissionError, OSError):
480: (16)                           self.logger.warning(
481: (20)                               "Both `sys_prefix` and `user` level settings are read-
only, cannot auto-migrate `disabledExtensions` to `lockedExtensions`"
482: (16)                           )
483: (4)                def install_extension(self, extension, existing=None, pin=None):
484: (8)                    """Install an extension package into JupyterLab.
```

```
(8)                         The extension is first validated.
(8)                         Returns `True` if a rebuild is recommended, `False` otherwise.
(8)                         """
(8)                         extension = _normalize_path(extension)
(8)                         extensions = self.info["extensions"]
(8)                         if extension in self.info["core_extensions"]:
(12)                            config = self._read_build_config()
(12)                            uninstalled = config.get("uninstalled_core_extensions", [])
(12)                            if extension in uninstalled:
(16)                                self.logger.info("Installing core extension %s" % extension)
(16)                                uninstalled.remove(extension)
(16)                                config["uninstalled_core_extensions"] = uninstalled
(16)                                self._write_build_config(config)
(16)                                return True
(12)                            return False
(8)                         self._ensure_app_dirs()
(8)                         with TemporaryDirectory() as tempdir:
(12)                            info = self._install_extension(extension, tempdir, pin=pin)
(8)                         name = info["name"]
(8)                         if info["is_dir"]:
(12)                            config = self._read_build_config()
(12)                            local = config.setdefault("local_extensions", {})
(12)                            local[name] = info["source"]
(12)                            self._write_build_config(config)
(8)                         if name in extensions:
(12)                            other = extensions[name]
(12)                            if other["path"] != info["path"] and other["location"] == "app":
(16)                                os.remove(other["path"])
(8)                         return True
(4)                     def build(
(8)                         self,
(8)                         name=None,
(8)                         version=None,
(8)                         static_url=None,
(8)                         clean_staging=False,
(8)                         production=True,
(8)                         minimize=True,
(4)                     ):
(8)                         """Build the application."""
(8)                         if production is None:
(12)                            production = not (self.info["linked_packages"] or
self.info["local_extensions"])
(8)                         if not production:
(12)                            minimize = False
(8)                         if self._options.splice_source:
(12)                            ensure_node_modules(REPO_ROOT, logger=self.logger)
(12)                            self._run(["node", YARN_PATH, "build:packages"], cwd=REPO_ROOT)
(8)                         info = ["production" if production else "development"]
(8)                         if production:
(12)                            info.append("minimized" if minimize else "not minimized")
(8)                         self.logger.info(f'Building jupyterlab assets ({", ".join(info)})')
(8)                         app_dir = self.app_dir
(8)                         self._populate_staging(
(12)                            name=name, version=version, static_url=static_url,
clean=clean_staging
(8)                         )
(8)                         staging = pjoin(app_dir, "staging")
(8)                         ret = self._run(["node", YARN_PATH, "install"], cwd=staging)
(8)                         if ret != 0:
(12)                            msg = "npm dependencies failed to install"
(12)                            self.logger.debug(msg)
(12)                            raise RuntimeError(msg)
(8)                         dedupe_yarn(staging, self.logger)
(8)                         command = f'build:{"prod" if production else "dev"}{":minimize" if
minimize else ""}'
(8)                         ret = self._run(["node", YARN_PATH, "run", command], cwd=staging)
(8)                         if ret != 0:
(12)                            msg = "JupyterLab failed to build"
(12)                            self.logger.debug(msg)
```

```
(12)              raise RuntimeError(msg)
(4)       def watch(self):
(8)           """Start the application watcher and then run the watch in
(8)           the background.
(8)           """
(8)           staging = pjoin(self.app_dir, "staging")
(8)           self._populate_staging()
(8)           self._run(["node", YARN_PATH, "install"], cwd=staging)
(8)           dedupe_yarn(staging, self.logger)
(8)           proc = WatchHelper(
(12)              ["node", YARN_PATH, "run", "watch"],
(12)              cwd=pjoin(self.app_dir, "staging"),
(12)              startup_regex=WEBPACK_EXPECT,
(12)              logger=self.logger,
(8)           )
(8)           return [proc]
(4)       def list_extensions(self):  # noqa
(8)           """Print an output of the extensions."""
(8)           self._ensure_disabled_info()
(8)           logger = self.logger
(8)           info = self.info
(8)           logger.info("JupyterLab v%s" % info["version"])
(8)           if info["federated_extensions"] or info["extensions"]:
(12)              info["compat_errors"] = self._get_extension_compat()
(8)           if info["federated_extensions"]:
(12)              self._list_federated_extensions()
(8)           if info["extensions"]:
(12)              logger.info("Other labextensions (built into JupyterLab)")
(12)              self._list_extensions(info, "app")
(12)              self._list_extensions(info, "sys")
(8)           local = info["local_extensions"]
(8)           if local:
(12)              logger.info("\n   local extensions:")
(12)              for name in sorted(local):
(16)                  logger.info(f"        {name}: {local[name]}")
(8)           linked_packages = info["linked_packages"]
(8)           if linked_packages:
(12)              logger.info("\n   linked packages:")
(12)              for key in sorted(linked_packages):
(16)                  source = linked_packages[key]["source"]
(16)                  logger.info(f"        {key}: {source}")
(8)           uninstalled_core = info["uninstalled_core"]
(8)           if uninstalled_core:
(12)              logger.info("\nUninstalled core extensions:")
(12)              [logger.info("    %s" % item) for item in
sorted(uninstalled_core)]
(8)           all_exts = (
(12)              list(info["federated_extensions"])
(12)              + list(info["extensions"])
(12)              + list(info["core_extensions"])
(8)           )
(8)           disabled = [i for i in info["disabled"] if i.partition(":")[0] in
all_exts]
(8)           if disabled:
(12)              logger.info("\nDisabled extensions:")
(12)              for item in sorted(disabled):
(16)                  if item in all_exts:
(20)                      item += " (all plugins)"  # noqa PLW2901
(16)                  logger.info("    %s" % item)
(8)           improper_shadowed = []
(8)           for ext_name in self.info["shadowed_exts"]:
(12)              source_version = self.info["extensions"][ext_name]["version"]
(12)              prebuilt_version = self.info["federated_extensions"][ext_name]
["version"]
(12)              if not gte(prebuilt_version, source_version, True):
(16)                  improper_shadowed.append(ext_name)
(8)           if improper_shadowed:
(12)              logger.info(
(16)                  "\nThe following source extensions are overshadowed by older
prebuilt extensions:"
(12)                            )
(12)                            [logger.info("    %s" % name) for name in
sorted(improper_shadowed)]
(8)                         messages = self.build_check(fast=True)
(8)                         if messages:
(12)                            logger.info("\nBuild recommended, please run `jupyter lab
build`:")
(12)                            [logger.info("    %s" % item) for item in messages]
(4)                     def build_check(self, fast=None):  # noqa
(8)                         """Determine whether JupyterLab should be built.
(8)                         Returns a list of messages.
(8)                         """
(8)                         if fast is None:
(12)                            fast = self.skip_full_build_check
(8)                         app_dir = self.app_dir
(8)                         local = self.info["local_extensions"]
(8)                         linked = self.info["linked_packages"]
(8)                         messages = []
(8)                         pkg_path = pjoin(app_dir, "static", "package.json")
(8)                         if not osp.exists(pkg_path):
(12)                            return ["No built application"]
(8)                         static_data = self.info["static_data"]
(8)                         old_jlab = static_data["jupyterlab"]
(8)                         old_deps = static_data.get("dependencies", {})
(8)                         static_version = old_jlab.get("version", "")
(8)                         if not static_version.endswith("-spliced"):
(12)                            core_version = old_jlab["version"]
(12)                            if Version(static_version) != Version(core_version):
(16)                                msg = "Version mismatch: %s (built), %s (current)"
(16)                                return [msg % (static_version, core_version)]
(8)                         shadowed_exts = self.info["shadowed_exts"]
(8)                         new_package = self._get_package_template(silent=fast)
(8)                         new_jlab = new_package["jupyterlab"]
(8)                         new_deps = new_package.get("dependencies", {})
(8)                         for ext_type in ["extensions", "mimeExtensions"]:
(12)                            for ext in new_jlab[ext_type]:
(16)                                if ext in shadowed_exts:
(20)                                    continue
(16)                                if ext not in old_jlab[ext_type]:
(20)                                    messages.append("%s needs to be included in build" % ext)
(12)                            for ext in old_jlab[ext_type]:
(16)                                if ext in shadowed_exts:
(20)                                    continue
(16)                                if ext not in new_jlab[ext_type]:
(20)                                    messages.append("%s needs to be removed from build" % ext)
(8)                         src_pkg_dir = pjoin(REPO_ROOT, "packages")
(8)                         for pkg, dep in new_deps.items():
(12)                            if old_deps.get(pkg, "").startswith(src_pkg_dir):
(16)                                continue
(12)                            if pkg not in old_deps:
(16)                                continue
(12)                            if pkg in local or pkg in linked:
(16)                                continue
(12)                            if old_deps[pkg] != dep:
(16)                                msg = "%s changed from %s to %s"
(16)                                messages.append(msg % (pkg, old_deps[pkg], new_deps[pkg]))
(8)                         for name, source in local.items():
(12)                            if fast or name in shadowed_exts:
(16)                                continue
(12)                            dname = pjoin(app_dir, "extensions")
(12)                            if self._check_local(name, source, dname):
(16)                                messages.append("%s content changed" % name)
(8)                         for name, item in linked.items():
(12)                            if fast or name in shadowed_exts:
(16)                                continue
(12)                            dname = pjoin(app_dir, "staging", "linked_packages")
(12)                            if self._check_local(name, item["source"], dname):
(16)                                messages.append("%s content changed" % name)
```

```
(8)                         return messages
(4)                     def uninstall_extension(self, name):
(8)                         """Uninstall an extension by name.
(8)                         Returns `True` if a rebuild is recommended, `False` otherwise.
(8)                         """
(8)                         info = self.info
(8)                         logger = self.logger
(8)                         if name in info["federated_extensions"]:
(12)                            if (
(16)                                info["federated_extensions"][name]
(16)                                .get("install", {})
(16)                                .get("uninstallInstructions", None)
(12)                            ):
(16)                                logger.error(
(20)                                    "JupyterLab cannot uninstall this extension. %s"
(20)                                    % info["federated_extensions"][name]["install"]["uninstallInstructions"]
(16)                                )
(12)                            else:
(16)                                logger.error(
(20)                                    "JupyterLab cannot uninstall %s since it was installed outside of JupyterLab. Use the same method used to install this extension to uninstall this extension."
(20)                                    % name
(16)                                )
(12)                            return False
(8)                         if name in info["core_extensions"]:
(12)                            config = self._read_build_config()
(12)                            uninstalled = config.get("uninstalled_core_extensions", [])
(12)                            if name not in uninstalled:
(16)                                logger.info("Uninstalling core extension %s" % name)
(16)                                uninstalled.append(name)
(16)                                config["uninstalled_core_extensions"] = uninstalled
(16)                                self._write_build_config(config)
(16)                                return True
(12)                            return False
(8)                         local = info["local_extensions"]
(8)                         for extname, data in info["extensions"].items():
(12)                            path = data["path"]
(12)                            if extname == name:
(16)                                msg = f"Uninstalling {name} from {osp.dirname(path)}"
(16)                                logger.info(msg)
(16)                                os.remove(path)
(16)                                if extname in local:
(20)                                    config = self._read_build_config()
(20)                                    data = config.setdefault("local_extensions", {})  # noqa PLW2901
(20)                                    del data[extname]
(20)                                    self._write_build_config(config)
(16)                                return True
(8)                         logger.warning('No labextension named "%s" installed' % name)
(8)                         return False
(4)                     def uninstall_all_extensions(self):
(8)                         """Uninstalls all extensions
(8)                         Returns `True` if a rebuild is recommended, `False` otherwise
(8)                         """
(8)                         should_rebuild = False
(8)                         for extname, _ in self.info["extensions"].items():
(12)                            uninstalled = self.uninstall_extension(extname)
(12)                            should_rebuild = should_rebuild or uninstalled
(8)                         return should_rebuild
(4)                     def update_all_extensions(self):
(8)                         """Update all non-local extensions.
(8)                         Returns `True` if a rebuild is recommended, `False` otherwise.
(8)                         """
(8)                         should_rebuild = False
(8)                         for extname, _ in self.info["extensions"].items():
(12)                            if extname in self.info["local_extensions"]:
(16)                                continue
```

```
(12)                        updated = self._update_extension(extname)
(12)                        should_rebuild = should_rebuild or updated
(8)                     return should_rebuild
(4)                 def update_extension(self, name):
(8)                     """Update an extension by name.
(8)                     Returns `True` if a rebuild is recommended, `False` otherwise.
(8)                     """
(8)                     if name not in self.info["extensions"]:
(12)                        self.logger.warning('No labextension named "%s" installed' % name)
(12)                        return False
(8)                     return self._update_extension(name)
(4)                 def _update_extension(self, name):
(8)                     """Update an extension by name.
(8)                     Returns `True` if a rebuild is recommended, `False` otherwise.
(8)                     """
(8)                     data = self.info["extensions"][name]
(8)                     if data["alias_package_source"]:
(12)                        self.logger.warning("Skipping updating pinned extension '%s'." %
name)
(12)                        return False
(8)                     try:
(12)                        latest = self._latest_compatible_package_version(name)
(8)                     except URLError:
(12)                        return False
(8)                     if latest is None:
(12)                        self.logger.warning(f"No compatible version found for {name}!")
(12)                        return False
(8)                     if latest == data["version"]:
(12)                        self.logger.info("Extension %r already up to date" % name)
(12)                        return False
(8)                     self.logger.info(f"Updating {name} to version {latest}")
(8)                     return self.install_extension(f"{name}@{latest}")
(4)                 def link_package(self, path):
(8)                     """Link a package at the given path.
(8)                     Returns `True` if a rebuild is recommended, `False` otherwise.
(8)                     """
(8)                     path = _normalize_path(path)
(8)                     if not osp.exists(path) or not osp.isdir(path):
(12)                        msg = 'Cannot install "%s" only link local directories'
(12)                        raise ValueError(msg % path)
(8)                     with TemporaryDirectory() as tempdir:
(12)                        info = self._extract_package(path, tempdir)
(8)                     messages = _validate_extension(info["data"])
(8)                     if not messages:
(12)                        return self.install_extension(path)
(8)                     self.logger.warning(
(12)                        "Installing %s as a linked package because it does not have
extension metadata:", path
(8)                     )
(8)                     [self.logger.warning("   %s" % m) for m in messages]
(8)                     config = self._read_build_config()
(8)                     linked = config.setdefault("linked_packages", {})
(8)                     linked[info["name"]] = info["source"]
(8)                     self._write_build_config(config)
(8)                     return True
(4)                 def unlink_package(self, path):
(8)                     """Unlink a package by name or at the given path.
(8)                     A ValueError is raised if the path is not an unlinkable package.
(8)                     Returns `True` if a rebuild is recommended, `False` otherwise.
(8)                     """
(8)                     path = _normalize_path(path)
(8)                     config = self._read_build_config()
(8)                     linked = config.setdefault("linked_packages", {})
(8)                     found = None
(8)                     for name, source in linked.items():
(12)                        if path in {name, source}:
(16)                            found = name
(8)                     if found:
(12)                        del linked[found]
```

```
815: (8)                        else:
816: (12)                           local = config.setdefault("local_extensions", {})
817: (12)                           for name, source in local.items():
818: (16)                               if path in {name, source}:
819: (20)                                   found = name
820: (12)                           if found:
821: (16)                               del local[found]
822: (16)                               path = self.info["extensions"][found]["path"]
823: (16)                               os.remove(path)
824: (8)                        if not found:
825: (12)                           raise ValueError("No linked package for %s" % path)
826: (8)                        self._write_build_config(config)
827: (8)                        return True
828: (4)                    def _is_extension_locked(self, extension, level="sys_prefix",
include_higher_levels=True):
829: (8)                        app_settings_dir = osp.join(self.app_dir, "settings")
830: (8)                        page_config = get_static_page_config(
831: (12)                           app_settings_dir=app_settings_dir,
832: (12)                           logger=self.logger,
833: (12)                           level=level,
834: (12)                           include_higher_levels=True,
835: (8)                        )
836: (8)                        locked = page_config.get("lockedExtensions", {})
837: (8)                        return locked.get(extension, False)
838: (4)                    def toggle_extension(self, extension, value, level="sys_prefix"):
839: (8)                        """Enable or disable a lab extension.
840: (8)                        Returns `True` if a rebuild is recommended, `False` otherwise.
841: (8)                        """
842: (8)                        app_settings_dir = osp.join(self.app_dir, "settings")
843: (8)                        if level != "system":
844: (12)                           allowed = get_allowed_levels()
845: (12)                           if self._is_extension_locked(
846: (16)                               extension, level=allowed[allowed.index(level) + 1],
include_higher_levels=True
847: (12)                           ):
848: (16)                               self.logger.info("Extension locked at a higher level, cannot
toggle status")
849: (16)                               return False
850: (8)                        complete_page_config = get_static_page_config(
851: (12)                           app_settings_dir=app_settings_dir, logger=self.logger, level="all"
852: (8)                        )
853: (8)                        level_page_config = get_static_page_config(
854: (12)                           app_settings_dir=app_settings_dir, logger=self.logger, level=level
855: (8)                        )
856: (8)                        disabled = complete_page_config.get("disabledExtensions", {})
857: (8)                        disabled_at_level = level_page_config.get("disabledExtensions", {})
858: (8)                        did_something = False
859: (8)                        is_disabled = disabled.get(extension, False)
860: (8)                        if value and not is_disabled:
861: (12)                           disabled_at_level[extension] = True
862: (12)                           did_something = True
863: (8)                        elif not value and is_disabled:
864: (12)                           disabled_at_level[extension] = False
865: (12)                           did_something = True
866: (8)                        if did_something:
867: (12)                           level_page_config["disabledExtensions"] = disabled_at_level
868: (12)                           write_page_config(level_page_config, level=level)
869: (8)                        return did_something
870: (4)                    def _maybe_mirror_disabled_in_locked(self, level="sys_prefix"):
871: (8)                        """Lock all extensions that were previously disabled.
872: (8)                        This exists to facilitate migration from 4.0 (which did not include
lock
873: (8)                        function) to 4.1 which exposes the plugin management to users in UI.
874: (8)                        Returns `True` if migration happened, `False` otherwise.
875: (8)                        """
876: (8)                        app_settings_dir = osp.join(self.app_dir, "settings")
877: (8)                        page_config = get_static_page_config(
878: (12)                           app_settings_dir=app_settings_dir, logger=self.logger, level=level
879: (8)                        )
```

```
(8)                     if "lockedExtensions" in page_config:
(12)                        return False
(8)                     disabled = page_config.get("disabledExtensions", {})
(8)                     if isinstance(disabled, list):
(12)                        disabled = {extension: True for extension in disabled}
(8)                     page_config["lockedExtensions"] = disabled
(8)                     write_page_config(page_config, level=level)
(8)                     return True
(4)                 def toggle_extension_lock(self, extension, value, level="sys_prefix"):
(8)                     """Lock or unlock a lab extension (/plugin)."""
(8)                     app_settings_dir = osp.join(self.app_dir, "settings")
(8)                     if level != "system":
(12)                        allowed = get_allowed_levels()
(12)                        if self._is_extension_locked(
(16)                            extension, level=allowed[allowed.index(level) + 1], include_higher_levels=True
(12)                        ):
(16)                            self.logger.info("Extension locked at a higher level, cannot toggle")
(16)                            return False
(8)                     page_config = get_static_page_config(
(12)                        app_settings_dir=app_settings_dir, logger=self.logger, level=level
(8)                     )
(8)                     locked = page_config.get("lockedExtensions", {})
(8)                     locked[extension] = value
(8)                     page_config["lockedExtensions"] = locked
(8)                     write_page_config(page_config, level=level)
(4)                 def check_extension(self, extension, check_installed_only=False):
(8)                     """Check if a lab extension is enabled or disabled"""
(8)                     self._ensure_disabled_info()
(8)                     info = self.info
(8)                     if extension in info["core_extensions"]:
(12)                        return self._check_core_extension(extension, info, check_installed_only)
(8)                     if extension in info["linked_packages"]:
(12)                        self.logger.info(f"{extension}:{GREEN_ENABLED}")
(12)                        return True
(8)                     return self._check_common_extension(extension, info, check_installed_only)
(4)                 def _check_core_extension(self, extension, info, check_installed_only):
(8)                     """Check if a core extension is enabled or disabled"""
(8)                     if extension in info["uninstalled_core"]:
(12)                        self.logger.info(f"{extension}:{RED_X}")
(12)                        return False
(8)                     if check_installed_only:
(12)                        self.logger.info(f"{extension}: {GREEN_OK}")
(12)                        return True
(8)                     if extension in info["disabled_core"]:
(12)                        self.logger.info(f"{extension}: {RED_DISABLED}")
(12)                        return False
(8)                     self.logger.info(f"{extension}:{GREEN_ENABLED}")
(8)                     return True
(4)                 def _check_common_extension(self, extension, info, check_installed_only):
(8)                     """Check if a common (non-core) extension is enabled or disabled"""
(8)                     if extension not in info["extensions"]:
(12)                        self.logger.info(f"{extension}:{RED_X}")
(12)                        return False
(8)                     errors = self._get_extension_compat()[extension]
(8)                     if errors:
(12)                        self.logger.info(f"{extension}:{RED_X} (compatibility errors)")
(12)                        return False
(8)                     if check_installed_only:
(12)                        self.logger.info(f"{extension}: {GREEN_OK}")
(12)                        return True
(8)                     if _is_disabled(extension, info["disabled"]):
(12)                        self.logger.info(f"{extension}: {RED_DISABLED}")
(12)                        return False
(8)                     self.logger.info(f"{extension}:{GREEN_ENABLED}")
(8)                     return True
```

```
(4)      def _get_app_info(self):
(8)          """Get information about the app."""
(8)          info = {}
(8)          info["core_data"] = core_data = self.core_data
(8)          info["extensions"] = extensions = self._get_extensions(core_data)
(8)          info["local_extensions"] = self._get_local_extensions()
(8)          info["linked_packages"] = self._get_linked_packages()
(8)          info["app_extensions"] = app = []
(8)          info["sys_extensions"] = sys = []
(8)          for name, data in extensions.items():
(12)             data["is_local"] = name in info["local_extensions"]
(12)             if data["location"] == "app":
(16)                 app.append(name)
(12)             else:
(16)                 sys.append(name)
(8)          info["uninstalled_core"] = self._get_uninstalled_core_extensions()
(8)          info["static_data"] = _get_static_data(self.app_dir)
(8)          app_data = info["static_data"] or core_data
(8)          info["version"] = app_data["jupyterlab"]["version"]
(8)          info["staticUrl"] = app_data["jupyterlab"].get("staticUrl", "")
(8)          info["sys_dir"] = self.sys_dir
(8)          info["app_dir"] = self.app_dir
(8)          info["core_extensions"] = _get_core_extensions(self.core_data)
(8)          info["federated_extensions"] = get_federated_extensions(self.labextensions_path)
(8)          info["shadowed_exts"] = [
(12)             ext for ext in info["extensions"] if ext in info["federated_extensions"]
(8)          ]
(8)          return info
(4)      def _ensure_disabled_info(self):
(8)          info = self.info
(8)          if "disabled" in info:
(12)             return
(8)          labextensions_path = self.labextensions_path
(8)          app_settings_dir = osp.join(self.app_dir, "settings")
(8)          page_config = get_page_config(
(12)             labextensions_path, app_settings_dir=app_settings_dir, logger=self.logger
(8)          )
(8)          disabled = page_config.get("disabledExtensions", {})
(8)          if isinstance(disabled, list):
(12)             disabled = {extension: True for extension in disabled}
(8)          info["disabled"] = disabled
(8)          locked = page_config.get("lockedExtensions", {})
(8)          if isinstance(locked, list):
(12)             locked = {extension: True for extension in locked}
(8)          info["locked"] = locked
(8)          disabled_core = []
(8)          for key in info["core_extensions"]:
(12)             if key in info["disabled"]:
(16)                 disabled_core.append(key)
(8)          info["disabled_core"] = disabled_core
(4)      def _populate_staging(self, name=None, version=None, static_url=None, clean=False):  # noqa
(8)          """Set up the assets in the staging directory."""
(8)          app_dir = self.app_dir
(8)          staging = pjoin(app_dir, "staging")
(8)          if clean and osp.exists(staging):
(12)             self.logger.info("Cleaning %s", staging)
(12)             _rmtree(staging, self.logger)
(8)          self._ensure_app_dirs()
(8)          if not version:
(12)             version = self.info["core_data"]["jupyterlab"]["version"]
(8)          splice_source = self._options.splice_source
(8)          if splice_source:
(12)             self.logger.debug("Splicing dev packages into app directory.")
(12)             source_dir = DEV_DIR
(12)             version = __version__ + "-spliced"
```

```
1010: (8)                   else:
1011: (12)                      source_dir = pjoin(HERE, "staging")
1012: (8)                   pkg_path = pjoin(staging, "package.json")
1013: (8)                   if osp.exists(pkg_path):
1014: (12)                      with open(pkg_path) as fid:
1015: (16)                          data = json.load(fid)
1016: (12)                      if data["jupyterlab"].get("version", "") != version:
1017: (16)                          _rmtree(staging, self.logger)
1018: (16)                          os.makedirs(staging)
1019: (8)                   for fname in [
1020: (12)                      "index.js",
1021: (12)                      "bootstrap.js",
1022: (12)                      "publicpath.js",
1023: (12)                      "webpack.config.js",
1024: (12)                      "webpack.prod.config.js",
1025: (12)                      "webpack.prod.minimize.config.js",
1026: (8)                   ]:
1027: (12)                      target = pjoin(staging, fname)
1028: (12)                      shutil.copy(pjoin(source_dir, fname), target)
1029: (8)                   for fname in [".yarnrc.yml", "yarn.js"]:
1030: (12)                      target = pjoin(staging, fname)
1031: (12)                      shutil.copy(pjoin(HERE, "staging", fname), target)
1032: (8)                   templates = pjoin(staging, "templates")
1033: (8)                   if osp.exists(templates):
1034: (12)                      _rmtree(templates, self.logger)
1035: (8)                   try:
1036: (12)                      shutil.copytree(pjoin(source_dir, "templates"), templates)
1037: (8)                   except shutil.Error as error:
1038: (12)                      real_error = "[Errno 22]" not in str(error) and "[Errno 5]" not in
str(error)
1039: (12)                      if real_error or not osp.exists(templates):
1040: (16)                          raise
1041: (8)                   linked_dir = pjoin(staging, "linked_packages")
1042: (8)                   if osp.exists(linked_dir):
1043: (12)                      _rmtree(linked_dir, self.logger)
1044: (8)                   os.makedirs(linked_dir)
1045: (8)                   extensions = self.info["extensions"]
1046: (8)                   removed = False
1047: (8)                   for key, source in self.info["local_extensions"].items():
1048: (12)                      if key not in extensions:
1049: (16)                          config = self._read_build_config()
1050: (16)                          data = config.setdefault("local_extensions", {})
1051: (16)                          del data[key]
1052: (16)                          self._write_build_config(config)
1053: (16)                          removed = True
1054: (16)                          continue
1055: (12)                      dname = pjoin(app_dir, "extensions")
1056: (12)                      self._update_local(key, source, dname, extensions[key],
"local_extensions")
1057: (8)                   if removed:
1058: (12)                      self.info["local_extensions"] = self._get_local_extensions()
1059: (8)                   linked = self.info["linked_packages"]
1060: (8)                   for key, item in linked.items():
1061: (12)                      dname = pjoin(staging, "linked_packages")
1062: (12)                      self._update_local(key, item["source"], dname, item,
"linked_packages")
1063: (8)                   data = self._get_package_template()
1064: (8)                   jlab = data["jupyterlab"]
1065: (8)                   if version:
1066: (12)                      jlab["version"] = version
1067: (8)                   if name:
1068: (12)                      jlab["name"] = name
1069: (8)                   if static_url:
1070: (12)                      jlab["staticUrl"] = static_url
1071: (8)                   if splice_source:
1072: (12)                      for path in glob(pjoin(REPO_ROOT, "packages", "*",
"package.json")):
1073: (16)                          local_path = osp.dirname(osp.abspath(path))
1074: (16)                          pkg_data = json.loads(Path(path).read_text(encoding="utf-8"))
```

```
1075: (16)                          name = pkg_data["name"]
1076: (16)                          if name in data["dependencies"]:
1077: (20)                              data["dependencies"][name] = local_path
1078: (20)                              jlab["linkedPackages"][name] = local_path
1079: (16)                          if name in data["resolutions"]:
1080: (20)                              data["resolutions"][name] = local_path
1081: (12)                      local_path = osp.abspath(pjoin(REPO_ROOT, "builder"))
1082: (12)                      data["devDependencies"]["@jupyterlab/builder"] = local_path
1083: (12)                      target = osp.join(staging, "node_modules", "@jupyterlab",
"builder")
1084: (12)                      node_modules = pjoin(staging, "node_modules")
1085: (12)                      if osp.exists(node_modules):
1086: (16)                          shutil.rmtree(node_modules, ignore_errors=True)
1087: (8)                   pkg_path = pjoin(staging, "package.json")
1088: (8)                   with open(pkg_path, "w") as fid:
1089: (12)                      json.dump(data, fid, indent=4)
1090: (8)                   lock_path = pjoin(staging, "yarn.lock")
1091: (8)                   lock_template = pjoin(HERE, "staging", "yarn.lock")
1092: (8)                   if not osp.exists(lock_path):
1093: (12)                      shutil.copy(lock_template, lock_path)
1094: (12)                      os.chmod(lock_path, stat.S_IWRITE | stat.S_IREAD)
1095: (4)               def _get_package_template(self, silent=False):  # noqa
1096: (8)                   """Get the template the for staging package.json file."""
1097: (8)                   logger = self.logger
1098: (8)                   data = deepcopy(self.info["core_data"])
1099: (8)                   local = self.info["local_extensions"]
1100: (8)                   linked = self.info["linked_packages"]
1101: (8)                   extensions = self.info["extensions"]
1102: (8)                   shadowed_exts = self.info["shadowed_exts"]
1103: (8)                   jlab = data["jupyterlab"]
1104: (8)                   def format_path(path):
1105: (12)                      path = osp.relpath(path,
osp.abspath(osp.realpath(pjoin(self.app_dir, "staging"))))
1106: (12)                      path = "file:" + path.replace(os.sep, "/")
1107: (12)                      if os.name == "nt":
1108: (16)                          path = path.lower()
1109: (12)                      return path
1110: (8)                   jlab["linkedPackages"] = {}
1111: (8)                   for key, source in local.items():
1112: (12)                      if key in shadowed_exts:
1113: (16)                          continue
1114: (12)                      jlab["linkedPackages"][key] = source
1115: (12)                      data["resolutions"][key] = "file:" + self.info["extensions"][key]
["path"]
1116: (8)                   for key, item in linked.items():
1117: (12)                      if key in shadowed_exts:
1118: (16)                          continue
1119: (12)                      path = pjoin(self.app_dir, "staging", "linked_packages")
1120: (12)                      path = pjoin(path, item["filename"])
1121: (12)                      data["dependencies"][key] = format_path(path)
1122: (12)                      jlab["linkedPackages"][key] = item["source"]
1123: (12)                      data["resolutions"][key] = format_path(path)
1124: (8)                   data["jupyterlab"]["extensionMetadata"] = {}
1125: (8)                   compat_errors = self._get_extension_compat()
1126: (8)                   for key, value in extensions.items():
1127: (12)                      errors = compat_errors[key]
1128: (12)                      if errors:
1129: (16)                          if not silent:
1130: (20)                              _log_single_compat_errors(logger, key, value["version"],
errors)
1131: (16)                          continue
1132: (12)                      data["dependencies"][key] = format_path(value["path"])
1133: (12)                      jlab_data = value["jupyterlab"]
1134: (12)                      for item in ["extension", "mimeExtension"]:
1135: (16)                          ext = jlab_data.get(item, False)
1136: (16)                          if not ext:
1137: (20)                              continue
1138: (16)                          if ext is True:
1139: (20)                              ext = ""
```

```
(16)                jlab[item + "s"][key] = ext
(16)                data["jupyterlab"]["extensionMetadata"][key] = jlab_data
(8)         for item in self.info["uninstalled_core"]:
(12)            if item in jlab["extensions"]:
(16)                data["jupyterlab"]["extensions"].pop(item)
(12)            elif item in jlab["mimeExtensions"]:
(16)                data["jupyterlab"]["mimeExtensions"].pop(item)
(12)            if item in data["dependencies"]:
(16)                data["dependencies"].pop(item)
(8)         return data
(4)     def _check_local(self, name, source, dname):
(8)         """Check if a local package has changed.
(8)         `dname` is the directory name of existing package tar archives.
(8)         """
(8)         with TemporaryDirectory() as tempdir:
(12)            info = self._extract_package(source, tempdir)
(12)            target = pjoin(dname, info["filename"])
(12)            return not osp.exists(target)
(4)     def _update_local(self, name, source, dname, data, dtype):
(8)         """Update a local dependency.  Return `True` if changed."""
(8)         existing = data["filename"]
(8)         if not osp.exists(pjoin(dname, existing)):
(12)            existing = ""
(8)         with TemporaryDirectory() as tempdir:
(12)            info = self._extract_package(source, tempdir)
(12)            if info["filename"] == existing:
(16)                return existing
(12)            shutil.move(info["path"], pjoin(dname, info["filename"]))
(8)         if existing:
(12)            os.remove(pjoin(dname, existing))
(8)         data["filename"] = info["filename"]
(8)         data["path"] = pjoin(data["tar_dir"], data["filename"])
(8)         return info["filename"]
(4)     def _get_extensions(self, core_data):
(8)         """Get the extensions for the application."""
(8)         app_dir = self.app_dir
(8)         extensions = {}
(8)         sys_path = pjoin(self.sys_dir, "extensions")
(8)         app_path = pjoin(self.app_dir, "extensions")
(8)         extensions = self._get_extensions_in_dir(self.sys_dir, core_data)
(8)         app_path = pjoin(app_dir, "extensions")
(8)         if app_path == sys_path or not osp.exists(app_path):
(12)            return extensions
(8)         extensions.update(self._get_extensions_in_dir(app_dir, core_data))
(8)         return extensions
(4)     def _get_extensions_in_dir(self, dname, core_data):
(8)         """Get the extensions in a given directory."""
(8)         extensions = {}
(8)         location = "app" if dname == self.app_dir else "sys"
(8)         for target in glob(pjoin(dname, "extensions", "*.tgz")):
(12)            data = read_package(target)
(12)            deps = data.get("dependencies", {})
(12)            name = data["name"]
(12)            jlab = data.get("jupyterlab", {})
(12)            path = osp.abspath(target)
(12)            filename = osp.basename(target)
(12)            if filename.startswith(PIN_PREFIX):
(16)                alias = filename[len(PIN_PREFIX) : -len(".tgz")]
(12)            else:
(16)                alias = None
(12)            url = get_package_url(data)
(12)            extensions[alias or name] = {
(16)                "description": data.get("description", ""),
(16)                "path": path,
(16)                "filename": osp.basename(path),
(16)                "url": url,
(16)                "version": data["version"],
(16)                "alias_package_source": name if alias else None,
(16)                "jupyterlab": jlab,
(16)                    "dependencies": deps,
(16)                    "tar_dir": osp.dirname(path),
(16)                    "location": location,
(12)                }
(8)             return extensions
(4)         def _get_extension_compat(self):
(8)             """Get the extension compatibility info."""
(8)             compat = {}
(8)             core_data = self.info["core_data"]
(8)             seen = set()
(8)             for name, data in self.info["federated_extensions"].items():
(12)                deps = data["dependencies"]
(12)                compat[name] = _validate_compatibility(name, deps, core_data)
(12)                seen.add(name)
(8)             for name, data in self.info["extensions"].items():
(12)                if name in seen:
(16)                    continue
(12)                deps = data["dependencies"]
(12)                compat[name] = _validate_compatibility(name, deps, core_data)
(8)             return compat
(4)         def _get_local_extensions(self):
(8)             """Get the locally installed extensions."""
(8)             return self._get_local_data("local_extensions")
(4)         def _get_linked_packages(self):
(8)             """Get the linked packages."""
(8)             info = self._get_local_data("linked_packages")
(8)             dname = pjoin(self.app_dir, "staging", "linked_packages")
(8)             for name, source in info.items():
(12)                info[name] = {"source": source, "filename": "", "tar_dir": dname}
(8)             if not osp.exists(dname):
(12)                return info
(8)             for path in glob(pjoin(dname, "*.tgz")):
(12)                path = osp.abspath(path)  # noqa PLW2901
(12)                data = read_package(path)
(12)                name = data["name"]
(12)                if name not in info:
(16)                    self.logger.warning("Removing orphaned linked package %s" % name)
(16)                    os.remove(path)
(16)                    continue
(12)                item = info[name]
(12)                item["filename"] = osp.basename(path)
(12)                item["path"] = path
(12)                item["version"] = data["version"]
(12)                item["data"] = data
(8)             return info
(4)         def _get_uninstalled_core_extensions(self):
(8)             """Get the uninstalled core extensions."""
(8)             config = self._read_build_config()
(8)             return config.get("uninstalled_core_extensions", [])
(4)         def _ensure_app_dirs(self):
(8)             """Ensure that the application directories exist"""
(8)             dirs = ["extensions", "settings", "staging", "schemas", "themes"]
(8)             for dname in dirs:
(12)                path = pjoin(self.app_dir, dname)
(12)                if not osp.exists(path):
(16)                    try:
(20)                        os.makedirs(path)
(16)                    except OSError as e:
(20)                        if e.errno != errno.EEXIST:
(24)                            raise
(4)         def _list_extensions(self, info, ext_type):
(8)             """List the extensions of a given type."""
(8)             self._ensure_disabled_info()
(8)             logger = self.logger
(8)             names = info["%s_extensions" % ext_type]
(8)             if not names:
(12)                return
(8)             dname = info["%s_dir" % ext_type]
```

```
1277: (8)                   error_accumulator = {}
1278: (8)                   logger.info(f"   {ext_type} dir: {dname}")
1279: (8)                   for name in sorted(names):
1280: (12)                      if name in info["federated_extensions"]:
1281: (16)                          continue
1282: (12)                      data = info["extensions"][name]
1283: (12)                      version = data["version"]
1284: (12)                      errors = info["compat_errors"][name]
1285: (12)                      extra = self._compose_extra_status(name, info, data, errors)
1286: (12)                      alias_package_source = data["alias_package_source"]
1287: (12)                      if alias_package_source:
1288: (16)                          logger.info(f"        {name} {alias_package_source} v{version}
{extra}")
1289: (12)                      else:
1290: (16)                          logger.info(f"        {name} v{version}{extra}")
1291: (12)                      if errors:
1292: (16)                          error_accumulator[name] = (version, errors)
1293: (8)                   _log_multiple_compat_errors(logger, error_accumulator, self.verbose)
1294: (8)                   logger.info("")
1295: (4)               def _list_federated_extensions(self):
1296: (8)                   self._ensure_disabled_info()
1297: (8)                   info = self.info
1298: (8)                   logger = self.logger
1299: (8)                   error_accumulator = {}
1300: (8)                   ext_dirs = {p: False for p in self.labextensions_path}
1301: (8)                   for value in info["federated_extensions"].values():
1302: (12)                      ext_dirs[value["ext_dir"]] = True
1303: (8)                   for ext_dir, has_exts in ext_dirs.items():
1304: (12)                      if not has_exts:
1305: (16)                          continue
1306: (12)                      logger.info(ext_dir)
1307: (12)                      for name in info["federated_extensions"]:
1308: (16)                          data = info["federated_extensions"][name]
1309: (16)                          if data["ext_dir"] != ext_dir:
1310: (20)                              continue
1311: (16)                          version = data["version"]
1312: (16)                          errors = info["compat_errors"][name]
1313: (16)                          extra = self._compose_extra_status(name, info, data, errors)
1314: (16)                          install = data.get("install")
1315: (16)                          if install:
1316: (20)                              extra += " ({}, {})".format(install["packageManager"],
install["packageName"])
1317: (16)                          logger.info(f"        {name} v{version}{extra}")
1318: (16)                          if errors:
1319: (20)                              error_accumulator[name] = (version, errors)
1320: (12)                      logger.info("")
1321: (8)                   _log_multiple_compat_errors(logger, error_accumulator, self.verbose)
1322: (4)               def _compose_extra_status(self, name: str, info: dict, data: dict, errors)
-> str:
1323: (8)                   extra = ""
1324: (8)                   if _is_disabled(name, info["disabled"]):
1325: (12)                      extra += " %s" % RED_DISABLED
1326: (8)                   else:
1327: (12)                      extra += " %s" % GREEN_ENABLED
1328: (8)                   if errors:
1329: (12)                      extra += " %s" % RED_X
1330: (8)                   else:
1331: (12)                      extra += " %s" % GREEN_OK
1332: (8)                   if data["is_local"]:
1333: (12)                      extra += "*"
1334: (8)                   lock_status = _is_locked(name, info["locked"])
1335: (8)                   if lock_status.entire_extension_locked:
1336: (12)                      extra += " 🔒 (all plugins locked)"
1337: (8)                   elif lock_status.locked_plugins:
1338: (12)                      plugin_list = ", ".join(sorted(lock_status.locked_plugins))
1339: (12)                      extra += " 🔒 (plugins: %s locked)" % plugin_list
1340: (8)                   return extra
1341: (4)               def _read_build_config(self):
1342: (8)                   """Get the build config data for the app dir."""
```

```
1343: (8)                   target = pjoin(self.app_dir, "settings", "build_config.json")
1344: (8)                   if not osp.exists(target):
1345: (12)                      return {}
1346: (8)                   else:
1347: (12)                      with open(target) as fid:
1348: (16)                          return json.load(fid)
1349: (4)               def _write_build_config(self, config):
1350: (8)                   """Write the build config to the app dir."""
1351: (8)                   self._ensure_app_dirs()
1352: (8)                   target = pjoin(self.app_dir, "settings", "build_config.json")
1353: (8)                   with open(target, "w") as fid:
1354: (12)                      json.dump(config, fid, indent=4)
1355: (4)               def _get_local_data(self, source):
1356: (8)                   """Get the local data for extensions or linked packages."""
1357: (8)                   config = self._read_build_config()
1358: (8)                   data = config.setdefault(source, {})
1359: (8)                   dead = []
1360: (8)                   for name, source in data.items():
1361: (12)                      if not osp.exists(source):
1362: (16)                          dead.append(name)
1363: (8)                   for name in dead:
1364: (12)                      link_type = source.replace("_", " ")
1365: (12)                      msg = f'**Note: Removing dead {link_type} "{name}"'
1366: (12)                      self.logger.warning(msg)
1367: (12)                      del data[name]
1368: (8)                   if dead:
1369: (12)                      self._write_build_config(config)
1370: (8)                   return data
1371: (4)               def _install_extension(self, extension, tempdir, pin=None):
1372: (8)                   """Install an extension with validation and return the name and
path."""
1373: (8)                   info = self._extract_package(extension, tempdir, pin=pin)
1374: (8)                   data = info["data"]
1375: (8)                   allow_fallback = "@" not in extension[1:] and not info["is_dir"]
1376: (8)                   name = info["name"]
1377: (8)                   messages = _validate_extension(data)
1378: (8)                   if messages:
1379: (12)                      msg = '"%s" is not a valid extension:\n%s'
1380: (12)                      msg = msg % (extension, "\n".join(messages))
1381: (12)                      if allow_fallback:
1382: (16)                          try:
1383: (20)                              version = self._latest_compatible_package_version(name)
1384: (16)                          except URLError:
1385: (20)                              raise ValueError(msg) from None
1386: (12)                      else:
1387: (16)                          raise ValueError(msg)
1388: (8)                   deps = data.get("dependencies", {})
1389: (8)                   errors = _validate_compatibility(extension, deps, self.core_data)
1390: (8)                   if errors:
1391: (12)                      msg = _format_compatibility_errors(data["name"], data["version"],
errors)
1392: (12)                      if allow_fallback:
1393: (16)                          try:
1394: (20)                              version = self._latest_compatible_package_version(name)
1395: (16)                          except URLError:
1396: (20)                              raise ValueError(msg) from None
1397: (16)                          if version and name:
1398: (20)                              self.logger.debug("Incompatible extension:\n%s", name)
1399: (20)                              self.logger.debug("Found compatible version: %s", version)
1400: (20)                              with TemporaryDirectory() as tempdir2:
1401: (24)                                  return self._install_extension(f"{name}@{version}",
tempdir2)
1402: (16)                          conflicts = "\n".join(msg.splitlines()[2:])
1403: (16)                          msg =
"".join((self._format_no_compatible_package_version(name), "\n\n", conflicts))
1404: (12)                      raise ValueError(msg)
1405: (8)                   target = pjoin(self.app_dir, "extensions", info["filename"])
1406: (8)                   if osp.exists(target):
1407: (12)                      os.remove(target)
```

```
(8)                     shutil.move(info["path"], target)
(8)                     info["path"] = target
(8)                     return info
(4)                 def _extract_package(self, source, tempdir, pin=None):
(8)                     """Call `npm pack` for an extension.
(8)                     The pack command will download the package tar if `source` is
(8)                     a package name, or run `npm pack` locally if `source` is a
(8)                     directory.
(8)                     """
(8)                     is_dir = osp.exists(source) and osp.isdir(source)
(8)                     if is_dir and not osp.exists(pjoin(source, "node_modules")):
(12)                        self._run(["node", YARN_PATH, "install"], cwd=source)
(8)                     info = {"source": source, "is_dir": is_dir}
(8)                     ret = self._run([which("npm"), "pack", source], cwd=tempdir)
(8)                     if ret != 0:
(12)                        msg = '"%s" is not a valid npm package'
(12)                        raise ValueError(msg % source)
(8)                     path = glob(pjoin(tempdir, "*.tgz"))[0]
(8)                     info["data"] = read_package(path)
(8)                     if is_dir:
(12)                        info["sha"] = sha = _tarsum(path)
(12)                        target = path.replace(".tgz", "-%s.tgz" % sha)
(12)                        shutil.move(path, target)
(12)                        info["path"] = target
(8)                     else:
(12)                        info["path"] = path
(8)                     if pin:
(12)                        old_path = info["path"]
(12)                        new_path = pjoin(osp.dirname(old_path), f"{PIN_PREFIX}{pin}.tgz")
(12)                        shutil.move(old_path, new_path)
(12)                        info["path"] = new_path
(8)                     info["filename"] = osp.basename(info["path"])
(8)                     info["name"] = info["data"]["name"]
(8)                     info["version"] = info["data"]["version"]
(8)                     return info
(4)                 def _latest_compatible_package_version(self, name):
(8)                     """Get the latest compatible version of a package"""
(8)                     core_data = self.info["core_data"]
(8)                     try:
(12)                        metadata = _fetch_package_metadata(self.registry, name, self.logger)
(8)                     except URLError:
(12)                        return
(8)                     versions = metadata.get("versions", {})
(8)                     def sort_key(key_value):
(12)                        return _semver_key(key_value[0], prerelease_first=True)
(8)                     for version, data in sorted(versions.items(), key=sort_key, reverse=True):
(12)                        deps = data.get("dependencies", {})
(12)                        errors = _validate_compatibility(name, deps, core_data)
(12)                        if not errors:
(16)                            if "deprecated" in data:
(20)                                self.logger.debug(
(24)                                    f"Disregarding compatible version of package as it is deprecated: {name}@{version}"
(20)                                )
(20)                                continue
(16)                            with TemporaryDirectory() as tempdir:
(20)                                info = self._extract_package(f"{name}@{version}", tempdir)
(16)                            if _validate_extension(info["data"]):
(20)                                return
(16)                            return version
(4)                 def latest_compatible_package_versions(self, names):
(8)                     """Get the latest compatible versions of several packages
(8)                     Like _latest_compatible_package_version, but optimized for
(8)                     retrieving the latest version for several packages in one go.
(8)                     """
(8)                     core_data = self.info["core_data"]
(8)                     keys = []
```

```
1474: (8)                   for name in names:
1475: (12)                      try:
1476: (16)                          metadata = _fetch_package_metadata(self.registry, name,
self.logger)
1477: (12)                      except URLError:
1478: (16)                          continue
1479: (12)                      versions = metadata.get("versions", {})
1480: (12)                      def sort_key(key_value):
1481: (16)                          return _semver_key(key_value[0], prerelease_first=True)
1482: (12)                      for version, data in sorted(versions.items(), key=sort_key,
reverse=True):
1483: (16)                          if "deprecated" in data:
1484: (20)                              continue
1485: (16)                          deps = data.get("dependencies", {})
1486: (16)                          errors = _validate_compatibility(name, deps, core_data)
1487: (16)                          if not errors:
1488: (20)                              keys.append(f"{name}@{version}")
1489: (20)                              break  # break inner for
1490: (8)                   versions = {}
1491: (8)                   if not keys:
1492: (12)                      return versions
1493: (8)                   with TemporaryDirectory() as tempdir:
1494: (12)                      ret = self._run([which("npm"), "pack", *keys], cwd=tempdir)
1495: (12)                      if ret != 0:
1496: (16)                          msg = '"%s" is not a valid npm package'
1497: (16)                          raise ValueError(msg % keys)
1498: (12)                      for key in keys:
1499: (16)                          fname = (
1500: (20)                              key[0].replace("@", "") + key[1:].replace("@", "-
").replace("/", "-") + ".tgz"
1501: (16)                          )
1502: (16)                          data = read_package(osp.join(tempdir, fname))
1503: (16)                          if not _validate_extension(data):
1504: (20)                              versions[data["name"]] = data["version"]
1505: (8)                   return versions
1506: (4)               def _format_no_compatible_package_version(self, name):
1507: (8)                   """Get the latest compatible version of a package"""
1508: (8)                   core_data = self.info["core_data"]
1509: (8)                   lab_newer_than_latest = False
1510: (8)                   latest_newer_than_lab = False
1511: (8)                   try:
1512: (12)                      metadata = _fetch_package_metadata(self.registry, name,
self.logger)
1513: (8)                   except URLError:
1514: (12)                      pass
1515: (8)                   else:
1516: (12)                      versions = metadata.get("versions", {})
1517: (12)                      def sort_key(key_value):
1518: (16)                          return _semver_key(key_value[0], prerelease_first=True)
1519: (12)                      store = tuple(sorted(versions.items(), key=sort_key,
reverse=True))
1520: (12)                      latest_deps = store[0][1].get("dependencies", {})
1521: (12)                      core_deps = core_data["resolutions"]
1522: (12)                      singletons = core_data["jupyterlab"]["singletonPackages"]
1523: (12)                      for key, value in latest_deps.items():
1524: (16)                          if key in singletons:
1525: (20)                              c = _compare_ranges(core_deps[key], value,
drop_prerelease1=True)
1526: (20)                              lab_newer_than_latest = lab_newer_than_latest or c < 0
1527: (20)                              latest_newer_than_lab = latest_newer_than_lab or c > 0
1528: (8)                   if lab_newer_than_latest:
1529: (12)                      return (
1530: (16)                          'The extension "%s" does not yet support the current version
of '
1531: (16)                          "JupyterLab.\n" % name
1532: (12)                      )
1533: (8)                   parts = [
1534: (12)                      "No version of {extension} could be found that is compatible with
"
(12)                    "the current version of JupyterLab."
(8)                 ]
(8)                 if latest_newer_than_lab:
(12)                    parts.extend(
(16)                        (
(20)                            "However, it seems to support a new version of
JupyterLab.",
(20)                            "Consider upgrading JupyterLab.",
(16)                        )
(12)                    )
(8)                 return " ".join(parts).format(extension=name)
(4)             def _run(self, cmd, **kwargs):
(8)                 """Run the command using our logger and abort callback.
(8)                 Returns the exit code.
(8)                 """
(8)                 if self.kill_event.is_set():
(12)                    msg = "Command was killed"
(12)                    raise ValueError(msg)
(8)                 kwargs["logger"] = self.logger
(8)                 kwargs["kill_event"] = self.kill_event
(8)                 proc = ProgressProcess(cmd, **kwargs)
(8)                 return proc.wait()
(0)         def _node_check(logger):
(4)             """Check for the existence of nodejs with the correct version."""
(4)             node = which("node")
(4)             try:
(8)                 output = subprocess.check_output([node, "node-version-check.js"],
cwd=HERE)  # noqa S603
(8)                 logger.debug(output.decode("utf-8"))
(4)             except Exception:
(8)                 data = CoreConfig()._data
(8)                 ver = data["engines"]["node"]
(8)                 msg = (
(12)                    "Please install nodejs %s before continuing. nodejs may be
installed using conda or directly from the nodejs website."
(12)                    % ver
(8)                 )
(8)                 raise ValueError(msg) from None
(0)         def _yarn_config(logger):
(4)             """Get the yarn configuration.
(4)             Returns
(4)             -------
(4)             {"yarn config": dict, "npm config": dict} if unsuccessfull the
subdictionary are empty
(4)             """
(4)             configuration = {"yarn config": {}, "npm config": {}}
(4)             try:
(8)                 node = which("node")
(4)             except ValueError:  # Node not found == user with no need for building
jupyterlab
(8)                 logger.debug("NodeJS was not found. Yarn user configuration is
ignored.")
(8)                 return configuration
(4)             try:
(8)                 output_binary = subprocess.check_output(
(12)                    [node, YARN_PATH, "config", "--json"],  # noqa S603
(12)                    stderr=subprocess.PIPE,
(12)                    cwd=HERE,
(8)                 )
(8)                 output = output_binary.decode("utf-8")
(8)                 lines = iter(output.splitlines())
(8)                 try:
(12)                    for line in lines:
(16)                        info = json.loads(line)
(16)                        if info["type"] == "info":
(20)                            key = info["data"]
(20)                            inspect = json.loads(next(lines))
(20)                            if inspect["type"] == "inspect":
(24)                                configuration[key] = inspect["data"]
```

```
1598: (8)                   except StopIteration:
1599: (12)                      pass
1600: (8)                   logger.debug("Yarn configuration loaded.")
1601: (4)               except subprocess.CalledProcessError as e:
1602: (8)                   logger.error(
1603: (12)                      "Fail to get yarn configuration. {!s}{!s}".format(
1604: (16)                          e.stderr.decode("utf-8"), e.output.decode("utf-8")
1605: (12)                      )
1606: (8)                   )
1607: (4)               except Exception as e:
1608: (8)                   logger.error(f"Fail to get yarn configuration. {e!s}")
1609: (4)               return configuration
1610: (0)           def _ensure_logger(logger=None):
1611: (4)               """Ensure that we have a logger"""
1612: (4)               return logger or logging.getLogger("jupyterlab")
1613: (0)           def _normalize_path(extension):
1614: (4)               """Normalize a given extension if it is a path."""
1615: (4)               extension = osp.expanduser(extension)
1616: (4)               if osp.exists(extension):
1617: (8)                   extension = osp.abspath(extension)
1618: (4)               return extension
1619: (0)           def _rmtree(path, logger):
1620: (4)               """Remove a tree, logging errors"""
1621: (4)               def onerror(*exc_info):
1622: (8)                   logger.debug("Error in shutil.rmtree", exc_info=exc_info)
1623: (4)               shutil.rmtree(path, onerror=onerror)
1624: (0)           def _unlink(path, logger):
1625: (4)               """Remove a file, logging errors"""
1626: (4)               try:
1627: (8)                   os.unlink(path)
1628: (4)               except Exception:
1629: (8)                   logger.debug("Error in os.unlink", exc_info=sys.exc_info())
1630: (0)           def _rmtree_star(path, logger):
1631: (4)               """Remove all files/trees within a dir, logging errors"""
1632: (4)               for filename in os.listdir(path):
1633: (8)                   file_path = osp.join(path, filename)
1634: (8)                   if osp.isfile(file_path) or osp.islink(file_path):
1635: (12)                      _unlink(file_path, logger)
1636: (8)                   elif osp.isdir(file_path):
1637: (12)                      _rmtree(file_path, logger)
1638: (0)           def _validate_extension(data):  # noqa
1639: (4)               """Detect if a package is an extension using its metadata.
1640: (4)               Returns any problems it finds.
1641: (4)               """
1642: (4)               jlab = data.get("jupyterlab", None)
1643: (4)               if jlab is None:
1644: (8)                   return ["No `jupyterlab` key"]
1645: (4)               if not isinstance(jlab, dict):
1646: (8)                   return ["The `jupyterlab` key must be a JSON object"]
1647: (4)               extension = jlab.get("extension", False)
1648: (4)               mime_extension = jlab.get("mimeExtension", False)
1649: (4)               theme_path = jlab.get("themePath", "")
1650: (4)               schema_dir = jlab.get("schemaDir", "")
1651: (4)               messages = []
1652: (4)               if not extension and not mime_extension:
1653: (8)                   messages.append("No `extension` or `mimeExtension` key present")
1654: (4)               if extension == mime_extension:
1655: (8)                   msg = "`mimeExtension` and `extension` must point to different
modules"
1656: (8)                   messages.append(msg)
1657: (4)               files = data["jupyterlab_extracted_files"]
1658: (4)               main = data.get("main", "index.js")
1659: (4)               if not main.endswith(".js"):
1660: (8)                   main += ".js"
1661: (4)               if extension is True:
1662: (8)                   extension = main
1663: (4)               elif extension and not extension.endswith(".js"):
1664: (8)                   extension += ".js"
1665: (4)               if mime_extension is True:
```

```
1666: (8)                   mime_extension = main
1667: (4)               elif mime_extension and not mime_extension.endswith(".js"):
1668: (8)                   mime_extension += ".js"
1669: (4)               if extension and extension not in files:
1670: (8)                   messages.append('Missing extension module "%s"' % extension)
1671: (4)               if mime_extension and mime_extension not in files:
1672: (8)                   messages.append('Missing mimeExtension module "%s"' % mime_extension)
1673: (4)               if theme_path and not any(f.startswith(str(Path(theme_path))) for f in
files):
1674: (8)                   messages.append('themePath is empty: "%s"' % theme_path)
1675: (4)               if schema_dir and not any(f.startswith(str(Path(schema_dir))) for f in
files):
1676: (8)                   messages.append('schemaDir is empty: "%s"' % schema_dir)
1677: (4)               return messages
1678: (0)           def _tarsum(input_file):
1679: (4)               """
1680: (4)               Compute the recursive sha sum of a tar file.
1681: (4)               """
1682: (4)               tar = tarfile.open(input_file, "r")
1683: (4)               chunk_size = 100 * 1024
1684: (4)               h = hashlib.new("sha1")  # noqa: S324
1685: (4)               for member in tar:
1686: (8)                   if not member.isfile():
1687: (12)                      continue
1688: (8)                   f = tar.extractfile(member)
1689: (8)                   data = f.read(chunk_size)
1690: (8)                   while data:
1691: (12)                      h.update(data)
1692: (12)                      data = f.read(chunk_size)
1693: (4)               return h.hexdigest()
1694: (0)           def _get_static_data(app_dir):
1695: (4)               """Get the data for the app static dir."""
1696: (4)               target = pjoin(app_dir, "static", "package.json")
1697: (4)               if osp.exists(target):
1698: (8)                   with open(target) as fid:
1699: (12)                      return json.load(fid)
1700: (4)               else:
1701: (8)                   return None
1702: (0)           def _validate_compatibility(extension, deps, core_data):
1703: (4)               """Validate the compatibility of an extension."""
1704: (4)               core_deps = core_data["resolutions"]
1705: (4)               singletons = core_data["jupyterlab"]["singletonPackages"]
1706: (4)               errors = []
1707: (4)               for key, value in deps.items():
1708: (8)                   if key in singletons:
1709: (12)                      overlap = _test_overlap(core_deps[key], value,
drop_prerelease1=True)
1710: (12)                      if overlap is False:
1711: (16)                          errors.append((key, core_deps[key], value))
1712: (4)               return errors
1713: (0)           def _test_overlap(spec1, spec2, drop_prerelease1=False,
drop_prerelease2=False):
1714: (4)               """Test whether two version specs overlap.
1715: (4)               Returns `None` if we cannot determine compatibility,
1716: (4)               otherwise whether there is an overlap
1717: (4)               """
1718: (4)               cmp = _compare_ranges(
1719: (8)                   spec1, spec2, drop_prerelease1=drop_prerelease1,
drop_prerelease2=drop_prerelease2
1720: (4)               )
1721: (4)               if cmp is None:
1722: (8)                   return
1723: (4)               return cmp == 0
1724: (0)           def _compare_ranges(spec1, spec2, drop_prerelease1=False,
drop_prerelease2=False):  # noqa
1725: (4)               """Test whether two version specs overlap.
1726: (4)               Returns `None` if we cannot determine compatibility,
1727: (4)               otherwise return 0 if there is an overlap, 1 if
1728: (4)               spec1 is lower/older than spec2, and -1 if spec1
```

```
    is higher/newer than spec2.
    """
    r1 = Range(spec1, True)
    r2 = Range(spec2, True)
    if not r1.range or not r2.range:
        return
    return_value = False
    for r1set, r2set in itertools.product(r1.set, r2.set):
        x1 = r1set[0].semver
        x2 = r1set[-1].semver
        y1 = r2set[0].semver
        y2 = r2set[-1].semver
        if x1.prerelease and drop_prerelease1:
            x1 = x1.inc("patch")
        if y1.prerelease and drop_prerelease2:
            y1 = y1.inc("patch")
        o1 = r1set[0].operator
        o2 = r2set[0].operator
        if o1.startswith("<") or o2.startswith("<"):
            continue
        lx = lte if x1 == x2 else lt
        ly = lte if y1 == y2 else lt
        gx = gte if x1 == x2 else gt
        gy = gte if x1 == x2 else gt
        def noop(x, y, z):
            return True
        if x1 == x2 and o1.startswith(">"):
            lx = noop
        if y1 == y2 and o2.startswith(">"):
            ly = noop
        if (
            gte(x1, y1, True)
            and ly(x1, y2, True)
            or gy(x2, y1, True)
            and ly(x2, y2, True)
            or gte(y1, x1, True)
            and lx(y1, x2, True)
            or gx(y2, x1, True)
            and lx(y2, x2, True)
        ):
            return 0
        if gte(y1, x2, True):
            if return_value is False:
                return_value = 1
            elif return_value == -1:
                return_value = None
            continue
        if gte(x1, y2, True):
            if return_value is False:
                return_value = -1
            elif return_value == 1:
                return_value = None
            continue
        msg = "Unexpected case comparing version ranges"
        raise AssertionError(msg)
    if return_value is False:
        return_value = None
    return return_value
def _is_disabled(name, disabled=None):
    """Test whether the package is disabled."""
    disabled = disabled or {}
    for pattern, value in disabled.items():
        if value is False:
            continue
        if name == pattern:
            return True
        if re.compile(pattern).match(name) is not None:
            return True
    return False
```

```
(0)     @dataclass(frozen=True)
(0)     class LockStatus:
(4)         entire_extension_locked: bool
(4)         locked_plugins: Optional[FrozenSet[str]] = None
(0)     def _is_locked(name, locked=None) -> LockStatus:
(4)         """Test whether the package is locked.
(4)         If only a subset of extension plugins is locked return them.
(4)         """
(4)         locked = locked or {}
(4)         locked_plugins = set()
(4)         for lock, value in locked.items():
(8)             if value is False:
(12)                continue
(8)             if name == lock:
(12)                return LockStatus(entire_extension_locked=True)
(8)             extension_part = lock.partition(":")[0]
(8)             if name == extension_part:
(12)                locked_plugins.add(lock)
(4)         return LockStatus(entire_extension_locked=False, locked_plugins=locked_plugins)
(0)     def _format_compatibility_errors(name, version, errors):
(4)         """Format a message for compatibility errors."""
(4)         msgs = []
(4)         l0 = 10
(4)         l1 = 10
(4)         for error in errors:
(8)             pkg, jlab, ext = error
(8)             jlab = str(Range(jlab, True))
(8)             ext = str(Range(ext, True))
(8)             msgs.append((pkg, jlab, ext))
(8)             l0 = max(l0, len(pkg) + 1)
(8)             l1 = max(l1, len(jlab) + 1)
(4)         msg = '\n"%s@%s" is not compatible with the current JupyterLab'
(4)         msg = msg % (name, version)
(4)         msg += "\nConflicting Dependencies:\n"
(4)         msg += "JupyterLab".ljust(l0)
(4)         msg += "Extension".ljust(l1)
(4)         msg += "Package\n"
(4)         for pkg, jlab, ext in msgs:
(8)             msg += jlab.ljust(l0) + ext.ljust(l1) + pkg + "\n"
(4)         return msg
(0)     def _log_multiple_compat_errors(logger, errors_map, verbose: bool):
(4)         """Log compatibility errors for multiple extensions at once"""
(4)         outdated = []
(4)         for name, (_, errors) in errors_map.items():
(8)             age = _compat_error_age(errors)
(8)             if age > 0:
(12)                outdated.append(name)
(4)         if outdated:
(8)             logger.warning(
(12)                "\n        ".join(
(16)                    [
(20)                        "\n   The following extensions may be outdated or specify dependencies that are incompatible with the current version of jupyterlab:",
(20)                        *outdated,
(20)                        "\n   If you are a user, check if an update is available for these packages.\n"
(20)                        + (
(24)                            "   If you are a developer, re-run with `--verbose` flag for more details.\n"
(24)                            if not verbose
(24)                            else "   See below for the details.\n"
(20)                        ),
(16)                    ]
(12)                )
(8)             )
(4)         for name, (version, errors) in errors_map.items():
(8)             if name in outdated and not verbose:
(12)                continue
```

```
(8)                 msg = _format_compatibility_errors(name, version, errors)
(8)                 logger.warning(f"{msg}\n")
(0)         def _log_single_compat_errors(logger, name, version, errors):
(4)             """Log compatability errors for a single extension"""
(4)             age = _compat_error_age(errors)
(4)             if age > 0:
(8)                 logger.warning('The extension "%s" is outdated.\n', name)
(4)             else:
(8)                 msg = _format_compatibility_errors(name, version, errors)
(8)                 logger.warning(f"{msg}\n")
(0)         def _compat_error_age(errors):
(4)             """Compare all incompatibilities for an extension.
(4)             Returns a number > 0 if all extensions are older than that supported by lab.
(4)             Returns a number < 0 if all extensions are newer than that supported by lab.
(4)             Returns 0 otherwise (i.e. a mix).
(4)             """
(4)             any_older = False
(4)             any_newer = False
(4)             for _, jlab, ext in errors:
(8)                 c = _compare_ranges(ext, jlab, drop_prerelease1=True)
(8)                 any_newer = any_newer or c < 0
(8)                 any_older = any_older or c > 0
(4)             if any_older and not any_newer:
(8)                 return 1
(4)             elif any_newer and not any_older:
(8)                 return -1
(4)             return 0
(0)         def _get_core_extensions(core_data):
(4)             """Get the core extensions."""
(4)             data = core_data["jupyterlab"]
(4)             return list(data["extensions"]) + list(data["mimeExtensions"])
(0)         def _semver_prerelease_key(prerelease):
(4)             """Sort key for prereleases.
(4)             Precedence for two pre-release versions with the same
(4)             major, minor, and patch version MUST be determined by
(4)             comparing each dot separated identifier from left to
(4)             right until a difference is found as follows:
(4)             identifiers consisting of only digits are compare
(4)             numerically and identifiers with letters or hyphens
(4)             are compared lexically in ASCII sort order. Numeric
(4)             identifiers always have lower precedence than non-
(4)             numeric identifiers. A larger set of pre-release
(4)             fields has a higher precedence than a smaller set,
(4)             if all of the preceding identifiers are equal.
(4)             """
(4)             for entry in prerelease:
(8)                 if isinstance(entry, int):
(12)                    yield ("", entry)
(8)                 else:
(12)                    yield (entry,)
(0)         def _semver_key(version, prerelease_first=False):
(4)             """A sort key-function for sorting semver version string.
(4)             The default sorting order is ascending (0.x -> 1.x -> 2.x).
(4)             If `prerelease_first`, pre-releases will come before
(4)             ALL other semver keys (not just those with same version).
(4)             I.e (1.0-pre, 2.0-pre -> 0.x -> 1.x -> 2.x).
(4)             Otherwise it will sort in the standard way that it simply
(4)             comes before any release with shared version string
(4)             (0.x -> 1.0-pre -> 1.x -> 2.0-pre -> 2.x).
(4)             """
(4)             v = make_semver(version, True)
(4)             key = ((0,) if v.prerelease else (1,)) if prerelease_first else ()
(4)             key = (*key, v.major, v.minor, v.patch)
(4)             if not prerelease_first:
(8)                 key = (*key, 0) if v.prerelease else (1,)
(4)             if v.prerelease:
(8)                 key = key + tuple(_semver_prerelease_key(v.prerelease))
```

```
(4)             return key
(0)         def _fetch_package_metadata(registry, name, logger):
(4)             """Fetch the metadata for a package from the npm registry"""
(4)             req = Request(  # noqa S310
(8)                 urljoin(registry, quote(name, safe="@")),
(8)                 headers={
(12)                    "Accept": ("application/vnd.npm.install-v1+json; q=1.0, application/json; q=0.8, */*")
(8)                 },
(4)             )
(4)             try:
(8)                 logger.debug("Fetching URL: %s" % (req.full_url))
(4)             except AttributeError:
(8)                 logger.debug("Fetching URL: %s" % (req.get_full_url()))
(4)             try:
(8)                 with contextlib.closing(urlopen(req)) as response:  # noqa S310
(12)                    return json.loads(response.read().decode("utf-8"))
(4)             except URLError as exc:
(8)                 logger.warning("Failed to fetch package metadata for %r: %r", name, exc)
(8)                 raise
(0)         if __name__ == "__main__":
(4)             watch_dev(HERE)
```

File 3 - coreconfig.py:

```
(0)         import json
(0)         import os.path as osp
(0)         from itertools import filterfalse
(0)         from .jlpmapp import HERE
(0)         def pjoin(*args):
(4)             """Join paths to create a real path."""
(4)             return osp.abspath(osp.join(*args))
(0)         def _get_default_core_data():
(4)             """Get the data for the app template."""
(4)             with open(pjoin(HERE, "staging", "package.json")) as fid:
(8)                 return json.load(fid)
(0)         def _is_lab_package(name):
(4)             """Whether a package name is in the lab namespace"""
(4)             return name.startswith("@jupyterlab/")
(0)         def _only_nonlab(collection):
(4)             """Filter a dict/sequence to remove all lab packages
(4)             This is useful to take the default values of e.g. singletons and filter
(4)             away the '@jupyterlab/' namespace packages, but leave any others (e.g.
(4)             lumino and react).
(4)             """
(4)             if isinstance(collection, dict):
(8)                 return {k: v for (k, v) in collection.items() if not _is_lab_package(k)}
(4)             elif isinstance(collection, (list, tuple)):
(8)                 return list(filterfalse(_is_lab_package, collection))
(4)             msg = "collection arg should be either dict or list/tuple"
(4)             raise TypeError(msg)
(0)         class CoreConfig:
(4)             """An object representing a core config.
(4)             This enables custom lab application to override some parts of the core
(4)             configuration of the build system.
(4)             """
(4)             def __init__(self):
(8)                 self._data = _get_default_core_data()
(4)             def add(self, name, semver, extension=False, mime_extension=False):
(8)                 """Remove an extension/singleton.
(8)                 If neither extension or mimeExtension is True (the default)
(8)                 the package is added as a singleton dependency.
(8)                 name: string
(12)                    The npm package name
(8)                 semver: string
```

```
41: (12)                        The semver range for the package
42: (8)                     extension: bool
43: (12)                        Whether the package is an extension
44: (8)                     mime_extension: bool
45: (12)                        Whether the package is a MIME extension
46: (8)                     """
47: (8)                     data = self._data
48: (8)                     if not name:
49: (12)                        msg = "Missing package name"
50: (12)                        raise ValueError(msg)
51: (8)                     if not semver:
52: (12)                        msg = "Missing package semver"
53: (12)                        raise ValueError(msg)
54: (8)                     if name in data["resolutions"]:
55: (12)                        msg = f"Package already present: {name!r}"
56: (12)                        raise ValueError(msg)
57: (8)                     data["resolutions"][name] = semver
58: (8)                     if mime_extension:
59: (12)                        data["jupyterlab"]["mimeExtensions"][name] = ""
60: (12)                        data["dependencies"][name] = semver
61: (8)                     elif extension:
62: (12)                        data["jupyterlab"]["extensions"][name] = ""
63: (12)                        data["dependencies"][name] = semver
64: (8)                     else:
65: (12)                        data["jupyterlab"]["singletonPackages"].append(name)
66: (4)                 def remove(self, name):
67: (8)                     """Remove a package/extension.
68: (8)                     name: string
69: (12)                        The npm package name
70: (8)                     """
71: (8)                     data = self._data
72: (8)                     maps = (
73: (12)                        data["dependencies"],
74: (12)                        data["resolutions"],
75: (12)                        data["jupyterlab"]["extensions"],
76: (12)                        data["jupyterlab"]["mimeExtensions"],
77: (8)                     )
78: (8)                     for m in maps:
79: (12)                        try:
80: (16)                            del m[name]
81: (12)                        except KeyError:
82: (16)                            pass
83: (8)                     data["jupyterlab"]["singletonPackages"].remove(name)
84: (4)                 def clear_packages(self, lab_only=True):
85: (8)                     """Clear the packages/extensions."""
86: (8)                     data = self._data
87: (8)                     if lab_only:
88: (12)                        data["dependencies"] = _only_nonlab(data["dependencies"])
89: (12)                        data["resolutions"] = _only_nonlab(data["resolutions"])
90: (12)                        data["jupyterlab"]["extensions"] = _only_nonlab(data["jupyterlab"]
["extensions"])
91: (12)                        data["jupyterlab"]["mimeExtensions"] = _only_nonlab(
92: (16)                            data["jupyterlab"]["mimeExtensions"]
93: (12)                        )
94: (12)                        data["jupyterlab"]["singletonPackages"] = _only_nonlab(
95: (16)                            data["jupyterlab"]["singletonPackages"]
96: (12)                        )
97: (8)                     else:
98: (12)                        data["dependencies"] = {}
99: (12)                        data["resolutions"] = {}
100: (12)                       data["jupyterlab"]["extensions"] = {}
101: (12)                       data["jupyterlab"]["mimeExtensions"] = {}
102: (12)                       data["jupyterlab"]["singletonPackages"] = []
103: (4)                @property
104: (4)                def extensions(self):
105: (8)                    """A dict mapping all extension names to their semver"""
106: (8)                    data = self._data
107: (8)                    return {k: data["resolutions"][k] for k in data["jupyterlab"]
["extensions"]}
```

```
108: (4)                @property
109: (4)                def mime_extensions(self):
110: (8)                    """A dict mapping all MIME extension names to their semver"""
111: (8)                    data = self._data
112: (8)                    return {k: data["resolutions"][k] for k in data["jupyterlab"]
["mimeExtensions"]}
113: (4)                @property
114: (4)                def singletons(self):
115: (8)                    """A dict mapping all singleton names to their semver"""
116: (8)                    data = self._data
117: (8)                    return {
118: (12)                       k: data["resolutions"].get(k, None) for k in data["jupyterlab"]
["singletonPackages"]
119: (8)                    }
120: (4)                @property
121: (4)                def static_dir(self):
122: (8)                    return self._data["jupyterlab"]["staticDir"]
123: (4)                @static_dir.setter
124: (4)                def static_dir(self, static_dir):
125: (8)                    self._data["jupyterlab"]["staticDir"] = static_dir

----------------------------------------

File 4 - debuglog.py:

1: (0)              """A mixin for adding a debug log file."""
2: (0)              import contextlib
3: (0)              import logging
4: (0)              import os
5: (0)              import sys
6: (0)              import tempfile
7: (0)              import traceback
8: (0)              import warnings
9: (0)              from traitlets import Unicode
10: (0)             from traitlets.config import Configurable
11: (0)             class DebugLogFileMixin(Configurable):
12: (4)                 debug_log_path = Unicode("", config=True, help="Path to use for the debug
log file")
13: (4)                 @contextlib.contextmanager
14: (4)                 def debug_logging(self):
15: (8)                     log_path = self.debug_log_path
16: (8)                     if os.path.isdir(log_path):
17: (12)                        log_path = os.path.join(log_path, "jupyterlab-debug.log")
18: (8)                     if not log_path:
19: (12)                        handle, log_path = tempfile.mkstemp(prefix="jupyterlab-debug-",
suffix=".log")
20: (12)                        os.close(handle)
21: (8)                     log = self.log
22: (8)                     for h in log.handlers:
23: (12)                        h.setLevel(self.log_level)
24: (8)                     log.setLevel("DEBUG")
25: (8)                     _debug_handler = logging.FileHandler(log_path, "w", "utf8",
delay=True)
26: (8)                     _log_formatter = self._log_formatter_cls(fmt=self.log_format,
datefmt=self.log_datefmt)
27: (8)                     _debug_handler.setFormatter(_log_formatter)
28: (8)                     _debug_handler.setLevel("DEBUG")
29: (8)                     log.addHandler(_debug_handler)
30: (8)                     try:
31: (12)                        yield
32: (8)                     except Exception as ex:
33: (12)                        _, _, exc_traceback = sys.exc_info()
34: (12)                        msg = traceback.format_exception(ex.__class__, ex, exc_traceback)
35: (12)                        for line in msg:
36: (16)                            self.log.debug(line)
37: (12)                        if isinstance(ex, SystemExit):
38: (16)                            warnings.warn(f"An error occurred. See the log file for
details: {log_path!s}")
39: (16)                            raise
```

```
40: (12)                        warnings.warn("An error occurred.")
41: (12)                        warnings.warn(msg[-1].strip())
42: (12)                        warnings.warn(f"See the log file for details: {log_path!s}")
43: (12)                        self.exit(1)
44: (8)                     else:
45: (12)                        log.removeHandler(_debug_handler)
46: (12)                        _debug_handler.flush()
47: (12)                        _debug_handler.close()
48: (12)                        try:
49: (16)                            os.remove(log_path)
50: (12)                        except FileNotFoundError:
51: (16)                            pass
52: (8)                     log.removeHandler(_debug_handler)

----------------------------------------

File 5 - federated_labextensions.py:

1: (0)              """Utilities for installing Javascript extensions for the notebook"""
2: (0)              import importlib
3: (0)              import json
4: (0)              import os
5: (0)              import os.path as osp
6: (0)              import platform
7: (0)              import shutil
8: (0)              import subprocess
9: (0)              import sys
10: (0)             from pathlib import Path
11: (0)             try:
12: (4)                 from importlib.metadata import PackageNotFoundError, version
13: (0)             except ImportError:
14: (4)                 from importlib_metadata import PackageNotFoundError, version
15: (0)             from os.path import basename, normpath
16: (0)             from os.path import join as pjoin
17: (0)             from jupyter_core.paths import ENV_JUPYTER_PATH, SYSTEM_JUPYTER_PATH,
jupyter_data_dir
18: (0)             from jupyter_core.utils import ensure_dir_exists
19: (0)             from jupyter_server.extension.serverextension import ArgumentConflict
20: (0)             from jupyterlab_server.config import get_federated_extensions
21: (0)             try:
22: (4)                 from tomllib import load  # Python 3.11+
23: (0)             except ImportError:
24: (4)                 from tomli import load
25: (0)             from .commands import _test_overlap
26: (0)             DEPRECATED_ARGUMENT = object()
27: (0)             HERE = osp.abspath(osp.dirname(__file__))
28: (0)             def develop_labextension(  # noqa
29: (4)                 path,
30: (4)                 symlink=True,
31: (4)                 overwrite=False,
32: (4)                 user=False,
33: (4)                 labextensions_dir=None,
34: (4)                 destination=None,
35: (4)                 logger=None,
36: (4)                 sys_prefix=False,
37: (0)             ):
38: (4)                 """Install a prebuilt extension for JupyterLab
39: (4)                 Stages files and/or directories into the labextensions directory.
40: (4)                 By default, this compares modification time, and only stages files that
need updating.
41: (4)                 If `overwrite` is specified, matching files are purged before proceeding.
42: (4)                 Parameters
43: (4)                 ----------
44: (4)                 path : path to file, directory, zip or tarball archive, or URL to install
45: (8)                     By default, the file will be installed with its base name, so
'/path/to/foo'
46: (8)                     will install to 'labextensions/foo'. See the destination argument
below to change this.
47: (8)                     Archives (zip or tarballs) will be extracted into the labextensions
```

```
directory.
48: (4)                 user : bool [default: False]
49: (8)                     Whether to install to the user's labextensions directory.
50: (8)                     Otherwise do a system-wide install (e.g.
/usr/local/share/jupyter/labextensions).
51: (4)                 overwrite : bool [default: False]
52: (8)                     If True, always install the files, regardless of what may already be
installed.
53: (4)                 symlink : bool [default: True]
54: (8)                     If True, create a symlink in labextensions, rather than copying files.
55: (8)                     Windows support for symlinks requires a permission bit which only
admin users
56: (8)                     have by default, so don't rely on it.
57: (4)                 labextensions_dir : str [optional]
58: (8)                     Specify absolute path of labextensions directory explicitly.
59: (4)                 destination : str [optional]
60: (8)                     name the labextension is installed to.  For example, if destination is
'foo', then
61: (8)                     the source file will be installed to 'labextensions/foo', regardless
of the source name.
62: (4)                 logger : Jupyter logger [optional]
63: (8)                     Logger instance to use
64: (4)                 """
65: (4)                 full_dest = None
66: (4)                 labext = _get_labextension_dir(
67: (8)                     user=user, sys_prefix=sys_prefix, labextensions_dir=labextensions_dir
68: (4)                 )
69: (4)                 ensure_dir_exists(labext)
70: (4)                 if isinstance(path, (list, tuple)):
71: (8)                     msg = "path must be a string pointing to a single extension to
install; call this function multiple times to install multiple extensions"
72: (8)                     raise TypeError(msg)
73: (4)                 if not destination:
74: (8)                     destination = basename(normpath(path))
75: (4)                 full_dest = normpath(pjoin(labext, destination))
76: (4)                 if overwrite and os.path.lexists(full_dest):
77: (8)                     if logger:
78: (12)                        logger.info("Removing: %s" % full_dest)
79: (8)                     if os.path.isdir(full_dest) and not os.path.islink(full_dest):
80: (12)                        shutil.rmtree(full_dest)
81: (8)                     else:
82: (12)                        os.remove(full_dest)
83: (4)                 os.makedirs(os.path.dirname(full_dest), exist_ok=True)
84: (4)                 if symlink:
85: (8)                     path = os.path.abspath(path)
86: (8)                     if not os.path.exists(full_dest):
87: (12)                        if logger:
88: (16)                            logger.info(f"Symlinking: {full_dest} -> {path}")
89: (12)                        try:
90: (16)                            os.symlink(path, full_dest)
91: (12)                        except OSError as e:
92: (16)                            if platform.platform().startswith("Windows"):
93: (20)                                msg = (
94: (24)                                    "Symlinks can be activated on Windows 10 for Python
version 3.8 or higher"
95: (24)                                    " by activating the 'Developer Mode'. That may not be
allowed by your administrators.\n"
96: (24)                                    "See https://docs.microsoft.com/en-
us/windows/apps/get-started/enable-your-device-for-development"
97: (20)                                )
98: (20)                                raise OSError(msg) from e
99: (16)                            raise
100: (8)                    elif not os.path.islink(full_dest):
101: (12)                       raise ValueError("%s exists and is not a symlink" % full_dest)
102: (4)                elif os.path.isdir(path):
103: (8)                    path = pjoin(os.path.abspath(path), "")  # end in path separator
104: (8)                    for parent, _, files in os.walk(path):
105: (12)                       dest_dir = pjoin(full_dest, parent[len(path) :])
106: (12)                       if not os.path.exists(dest_dir):
```

```
(16)                            if logger:
(20)                                logger.info("Making directory: %s" % dest_dir)
(16)                            os.makedirs(dest_dir)
(12)                        for file_name in files:
(16)                            src = pjoin(parent, file_name)
(16)                            dest_file = pjoin(dest_dir, file_name)
(16)                            _maybe_copy(src, dest_file, logger=logger)
(4)                 else:
(8)                     src = path
(8)                     _maybe_copy(src, full_dest, logger=logger)
(4)                 return full_dest
(0)             def develop_labextension_py(
(4)                 module,
(4)                 user=False,
(4)                 sys_prefix=False,
(4)                 overwrite=True,
(4)                 symlink=True,
(4)                 labextensions_dir=None,
(4)                 logger=None,
(0)             ):
(4)                 """Develop a labextension bundled in a Python package.
(4)                 Returns a list of installed/updated directories.
(4)                 See develop_labextension for parameter information."""
(4)                 m, labexts = _get_labextension_metadata(module)
(4)                 base_path = os.path.split(m.__file__)[0]
(4)                 full_dests = []
(4)                 for labext in labexts:
(8)                     src = os.path.join(base_path, labext["src"])
(8)                     dest = labext["dest"]
(8)                     if logger:
(12)                        logger.info(f"Installing {src} -> {dest}")
(8)                     if not os.path.exists(src):
(12)                        build_labextension(base_path, logger=logger)
(8)                     full_dest = develop_labextension(
(12)                        src,
(12)                        overwrite=overwrite,
(12)                        symlink=symlink,
(12)                        user=user,
(12)                        sys_prefix=sys_prefix,
(12)                        labextensions_dir=labextensions_dir,
(12)                        destination=dest,
(12)                        logger=logger,
(8)                     )
(8)                     full_dests.append(full_dest)
(4)                 return full_dests
(0)             def build_labextension(
(4)                 path, logger=None, development=False, static_url=None, source_map=False,
core_path=None
(0)             ):
(4)                 """Build a labextension in the given path"""
(4)                 core_path = osp.join(HERE, "staging") if core_path is None else
str(Path(core_path).resolve())
(4)                 ext_path = str(Path(path).resolve())
(4)                 if logger:
(8)                     logger.info("Building extension in %s" % path)
(4)                 builder = _ensure_builder(ext_path, core_path)
(4)                 arguments = ["node", builder, "--core-path", core_path, ext_path]
(4)                 if static_url is not None:
(8)                     arguments.extend(["--static-url", static_url])
(4)                 if development:
(8)                     arguments.append("--development")
(4)                 if source_map:
(8)                     arguments.append("--source-map")
(4)                 subprocess.check_call(arguments, cwd=ext_path)  # noqa S603
(0)             def watch_labextension(
(4)                 path, labextensions_path, logger=None, development=False,
source_map=False, core_path=None
(0)             ):
(4)                 """Watch a labextension in a given path"""
```

```
173: (4)                core_path = osp.join(HERE, "staging") if core_path is None else
str(Path(core_path).resolve())
174: (4)                ext_path = str(Path(path).resolve())
175: (4)                if logger:
176: (8)                    logger.info("Building extension in %s" % path)
177: (4)                federated_extensions = get_federated_extensions(labextensions_path)
178: (4)                with open(pjoin(ext_path, "package.json")) as fid:
179: (8)                    ext_data = json.load(fid)
180: (4)                if ext_data["name"] not in federated_extensions:
181: (8)                    develop_labextension_py(ext_path, sys_prefix=True)
182: (4)                else:
183: (8)                    full_dest = pjoin(federated_extensions[ext_data["name"]]["ext_dir"],
ext_data["name"])
184: (8)                    output_dir = pjoin(ext_path, ext_data["jupyterlab"].get("outputDir",
"static"))
185: (8)                    if not osp.islink(full_dest):
186: (12)                       shutil.rmtree(full_dest)
187: (12)                       os.symlink(output_dir, full_dest)
188: (4)                builder = _ensure_builder(ext_path, core_path)
189: (4)                arguments = ["node", builder, "--core-path", core_path, "--watch",
ext_path]
190: (4)                if development:
191: (8)                    arguments.append("--development")
192: (4)                if source_map:
193: (8)                    arguments.append("--source-map")
194: (4)                subprocess.check_call(arguments, cwd=ext_path)  # noqa S603
195: (0)            def _ensure_builder(ext_path, core_path):
196: (4)                """Ensure that we can build the extension and return the builder script
path"""
197: (4)                with open(osp.join(core_path, "package.json")) as fid:
198: (8)                    core_data = json.load(fid)
199: (4)                with open(osp.join(ext_path, "package.json")) as fid:
200: (8)                    ext_data = json.load(fid)
201: (4)                dep_version1 = core_data["devDependencies"]["@jupyterlab/builder"]
202: (4)                dep_version2 = ext_data.get("devDependencies",
{}).get("@jupyterlab/builder")
203: (4)                dep_version2 = dep_version2 or ext_data.get("dependencies",
{}).get("@jupyterlab/builder")
204: (4)                if dep_version2 is None:
205: (8)                    raise ValueError(
206: (12)                       "Extensions require a devDependency on @jupyterlab/builder@%s" %
dep_version1
207: (8)                    )
208: (4)                if "/" in dep_version2:
209: (8)                    with open(osp.join(ext_path, dep_version2, "package.json")) as fid:
210: (12)                       dep_version2 = json.load(fid).get("version")
211: (4)                if not osp.exists(osp.join(ext_path, "node_modules")):
212: (8)                    subprocess.check_call(["jlpm"], cwd=ext_path)  # noqa S603 S607
213: (4)                target = ext_path
214: (4)                while not osp.exists(osp.join(target, "node_modules", "@jupyterlab",
"builder")):
215: (8)                    if osp.dirname(target) == target:
216: (12)                       msg = "Could not find @jupyterlab/builder"
217: (12)                       raise ValueError(msg)
218: (8)                    target = osp.dirname(target)
219: (4)                overlap = _test_overlap(
220: (8)                    dep_version1, dep_version2, drop_prerelease1=True,
drop_prerelease2=True
221: (4)                )
222: (4)                if not overlap:
223: (8)                    with open(
224: (12)                       osp.join(target, "node_modules", "@jupyterlab", "builder",
"package.json")
225: (8)                    ) as fid:
226: (12)                       dep_version2 = json.load(fid).get("version")
227: (8)                    overlap = _test_overlap(
228: (12)                       dep_version1, dep_version2, drop_prerelease1=True,
drop_prerelease2=True
229: (8)                    )
```

```
(4)                if not overlap:
(8)                    msg = f"Extensions require a devDependency on @jupyterlab/builder@{dep_version1}, you have a dependency on {dep_version2}"
(8)                    raise ValueError(msg)
(4)                return osp.join(
(8)                    target, "node_modules", "@jupyterlab", "builder", "lib", "build-labextension.js"
(4)                )
(0)            def _should_copy(src, dest, logger=None):
(4)                """Should a file be copied, if it doesn't exist, or is newer?
(4)                Returns whether the file needs to be updated.
(4)                Parameters
(4)                ----------
(4)                src : string
(8)                    A path that should exist from which to copy a file
(4)                src : string
(8)                    A path that might exist to which to copy a file
(4)                logger : Jupyter logger [optional]
(8)                    Logger instance to use
(4)                """
(4)                if not os.path.exists(dest):
(8)                    return True
(4)                if os.stat(src).st_mtime - os.stat(dest).st_mtime > 1e-6:  # noqa
(8)                    if logger:
(12)                       logger.warning("Out of date: %s" % dest)
(8)                    return True
(4)                if logger:
(8)                    logger.info("Up to date: %s" % dest)
(4)                return False
(0)            def _maybe_copy(src, dest, logger=None):
(4)                """Copy a file if it needs updating.
(4)                Parameters
(4)                ----------
(4)                src : string
(8)                    A path that should exist from which to copy a file
(4)                src : string
(8)                    A path that might exist to which to copy a file
(4)                logger : Jupyter logger [optional]
(8)                    Logger instance to use
(4)                """
(4)                if _should_copy(src, dest, logger=logger):
(8)                    if logger:
(12)                       logger.info(f"Copying: {src} -> {dest}")
(8)                    shutil.copy2(src, dest)
(0)            def _get_labextension_dir(user=False, sys_prefix=False, prefix=None, labextensions_dir=None):
(4)                """Return the labextension directory specified
(4)                Parameters
(4)                ----------
(4)                user : bool [default: False]
(8)                    Get the user's .jupyter/labextensions directory
(4)                sys_prefix : bool [default: False]
(8)                    Get sys.prefix, i.e. ~/.envs/my-env/share/jupyter/labextensions
(4)                prefix : str [optional]
(8)                    Get custom prefix
(4)                labextensions_dir : str [optional]
(8)                    Get what you put in
(4)                """
(4)                conflicting = [
(8)                    ("user", user),
(8)                    ("prefix", prefix),
(8)                    ("labextensions_dir", labextensions_dir),
(8)                    ("sys_prefix", sys_prefix),
(4)                ]
(4)                conflicting_set = [f"{n}={v!r}" for n, v in conflicting if v]
(4)                if len(conflicting_set) > 1:
(8)                    msg = "cannot specify more than one of user, sys_prefix, prefix, or labextensions_dir, but got: {}".format(
(12)                       ", ".join(conflicting_set)
(8)                         )
(8)                         raise ArgumentConflict(msg)
(4)                     if user:
(8)                         labext = pjoin(jupyter_data_dir(), "labextensions")
(4)                     elif sys_prefix:
(8)                         labext = pjoin(ENV_JUPYTER_PATH[0], "labextensions")
(4)                     elif prefix:
(8)                         labext = pjoin(prefix, "share", "jupyter", "labextensions")
(4)                     elif labextensions_dir:
(8)                         labext = labextensions_dir
(4)                     else:
(8)                         labext = pjoin(SYSTEM_JUPYTER_PATH[0], "labextensions")
(4)                     return labext
(0)                 def _get_labextension_metadata(module):  # noqa
(4)                     """Get the list of labextension paths associated with a Python module.
(4)                     Returns a tuple of (the module path,             [{
(8)                         'src': 'mockextension',
(8)                         'dest': '_mockdestination'
(4)                     }])
(4)                     Parameters
(4)                     ----------
(4)                     module : str
(8)                         Importable Python module exposing the
(8)                         magic-named `_jupyter_labextension_paths` function
(4)                     """
(4)                     mod_path = osp.abspath(module)
(4)                     if not osp.exists(mod_path):
(8)                         msg = f"The path `{mod_path}` does not exist."
(8)                         raise FileNotFoundError(msg)
(4)                     errors = []
(4)                     try:
(8)                         m = importlib.import_module(module)
(8)                         if hasattr(m, "_jupyter_labextension_paths"):
(12)                            return m, m._jupyter_labextension_paths()
(4)                     except Exception as exc:
(8)                         errors.append(exc)
(4)                     package = None
(4)                     if os.path.exists(os.path.join(mod_path, "pyproject.toml")):
(8)                         with open(os.path.join(mod_path, "pyproject.toml"), "rb") as fid:
(12)                            data = load(fid)
(8)                         package = data.get("project", {}).get("name")
(4)                     if not package:
(8)                         try:
(12)                            package = (
(16)                                subprocess.check_output(
(20)                                    [sys.executable, "setup.py", "--name"],  # noqa S603
(20)                                    cwd=mod_path,
(16)                                )
(16)                                .decode("utf8")
(16)                                .strip()
(12)                            )
(8)                         except subprocess.CalledProcessError:
(12)                            msg = (
(16)                                f"The Python package `{module}` is not a valid package, "
(16)                                "it is missing the `setup.py` file."
(12)                            )
(12)                            raise FileNotFoundError(msg) from None
(4)                     try:
(8)                         version(package)
(4)                     except PackageNotFoundError:
(8)                         subprocess.check_call([sys.executable, "-m", "pip", "install", "-e",
mod_path])  # noqa S603
(8)                         sys.path.insert(0, mod_path)
(4)                     from setuptools import find_namespace_packages, find_packages
(4)                     package_candidates = [
(8)                         package.replace("-", "_"),  # Module with the same name as package
(4)                     ]
(4)                     package_candidates.extend(find_packages(mod_path))  # Packages in the
module path
```

```
362: (4)                package_candidates.extend(
363: (8)                    find_namespace_packages(mod_path)
364: (4)                )  # Namespace packages in the module path
365: (4)                for package in package_candidates:
366: (8)                    try:
367: (12)                       m = importlib.import_module(package)
368: (12)                       if hasattr(m, "_jupyter_labextension_paths"):
369: (16)                           return m, m._jupyter_labextension_paths()
370: (8)                    except Exception as exc:
371: (12)                       errors.append(exc)
372: (4)                msg = f"There is no labextension at {module}. Errors encountered:
{errors}"
373: (4)                raise ModuleNotFoundError(msg)

----------------------------------------

File 6 - jlpmapp.py:

1: (0)              """A Jupyter-aware wrapper for the yarn package manager"""
2: (0)              import os
3: (0)              import sys
4: (0)              from jupyterlab_server.process import subprocess, which
5: (0)              HERE = os.path.dirname(os.path.abspath(__file__))
6: (0)              YARN_PATH = os.path.join(HERE, "staging", "yarn.js")
7: (0)              def execvp(cmd, argv):
8: (4)                  """Execvp, except on Windows where it uses Popen.
9: (4)                  The first argument, by convention, should point to the filename
10: (4)                 associated with the file being executed.
11: (4)                 Python provides execvp on Windows, but its behavior is problematic
12: (4)                 (Python bug#9148).
13: (4)                 """
14: (4)                 cmd = which(cmd)
15: (4)                 if os.name == "nt":
16: (8)                     import signal
17: (8)                     import sys
18: (8)                     p = subprocess.Popen([cmd] + argv[1:])
19: (8)                     signal.signal(signal.SIGINT, signal.SIG_IGN)
20: (8)                     p.wait()
21: (8)                     sys.exit(p.returncode)
22: (4)                 else:
23: (8)                     os.execvp(cmd, argv)  # noqa S606
24: (0)             def main(argv=None):
25: (4)                 """Run node and return the result."""
26: (4)                 argv = argv or sys.argv[1:]
27: (4)                 execvp("node", ["node", YARN_PATH, *argv])

----------------------------------------

File 7 - labapp.py:

1: (0)              """A tornado based Jupyter lab server."""
2: (0)              import dataclasses
3: (0)              import json
4: (0)              import os
5: (0)              import sys
6: (0)              from jupyter_core.application import JupyterApp, NoStart, base_aliases,
base_flags
7: (0)              from jupyter_server._version import version_info as jpserver_version_info
8: (0)              from jupyter_server.serverapp import flags
9: (0)              from jupyter_server.utils import url_path_join as ujoin
10: (0)             from jupyterlab_server import (
11: (4)                 LabServerApp,
12: (4)                 LicensesApp,
13: (4)                 WorkspaceExportApp,
14: (4)                 WorkspaceImportApp,
15: (4)                 WorkspaceListApp,
16: (0)             )
17: (0)             from notebook_shim.shim import NotebookConfigShimMixin
18: (0)             from traitlets import Bool, Instance, Type, Unicode, default
```

```
19: (0)             from ._version import __version__
20: (0)             from .commands import (
21: (4)                 DEV_DIR,
22: (4)                 HERE,
23: (4)                 AppOptions,
24: (4)                 build,
25: (4)                 clean,
26: (4)                 ensure_app,
27: (4)                 ensure_core,
28: (4)                 ensure_dev,
29: (4)                 get_app_dir,
30: (4)                 get_app_version,
31: (4)                 get_user_settings_dir,
32: (4)                 get_workspaces_dir,
33: (4)                 pjoin,
34: (4)                 watch,
35: (4)                 watch_dev,
36: (0)             )
37: (0)             from .coreconfig import CoreConfig
38: (0)             from .debuglog import DebugLogFileMixin
39: (0)             from .extensions import MANAGERS as EXT_MANAGERS
40: (0)             from .extensions.manager import PluginManager
41: (0)             from .extensions.readonly import ReadOnlyExtensionManager
42: (0)             from .handlers.announcements import (
43: (4)                 CheckForUpdate,
44: (4)                 CheckForUpdateABC,
45: (4)                 CheckForUpdateHandler,
46: (4)                 NewsHandler,
47: (4)                 check_update_handler_path,
48: (4)                 news_handler_path,
49: (0)             )
50: (0)             from .handlers.build_handler import Builder, BuildHandler, build_path
51: (0)             from .handlers.error_handler import ErrorHandler
52: (0)             from .handlers.extension_manager_handler import ExtensionHandler,
extensions_handler_path
53: (0)             from .handlers.plugin_manager_handler import PluginHandler,
plugins_handler_path
54: (0)             DEV_NOTE = """You're running JupyterLab from source.
55: (0)             If you're working on the TypeScript sources of JupyterLab, try running
56: (4)                 jupyter lab --dev-mode --watch
57: (0)             to have the system incrementally watch and build JupyterLab for you, as you
58: (0)             make changes.
59: (0)             """
60: (0)             CORE_NOTE = """
61: (0)             Running the core application with no additional extensions or settings
62: (0)             """
63: (0)             build_aliases = dict(base_aliases)
64: (0)             build_aliases["app-dir"] = "LabBuildApp.app_dir"
65: (0)             build_aliases["name"] = "LabBuildApp.name"
66: (0)             build_aliases["version"] = "LabBuildApp.version"
67: (0)             build_aliases["dev-build"] = "LabBuildApp.dev_build"
68: (0)             build_aliases["minimize"] = "LabBuildApp.minimize"
69: (0)             build_aliases["debug-log-path"] = "DebugLogFileMixin.debug_log_path"
70: (0)             build_flags = dict(base_flags)
71: (0)             build_flags["dev-build"] = (
72: (4)                 {"LabBuildApp": {"dev_build": True}},
73: (4)                 "Build in development mode.",
74: (0)             )
75: (0)             build_flags["no-minimize"] = (
76: (4)                 {"LabBuildApp": {"minimize": False}},
77: (4)                 "Do not minimize a production build.",
78: (0)             )
79: (0)             build_flags["splice-source"] = (
80: (4)                 {"LabBuildApp": {"splice_source": True}},
81: (4)                 "Splice source packages into app directory.",
82: (0)             )
83: (0)             version = __version__
84: (0)             app_version = get_app_version()
85: (0)             if version != app_version:
    version = f"{__version__} (dev), {app_version} (app)"
build_failure_msg = """Build failed.
Troubleshooting: If the build failed due to an out-of-memory error, you
may be able to fix it by disabling the `dev_build` and/or `minimize` options.
If you are building via the `jupyter lab build` command, you can disable
these options like so:
jupyter lab build --dev-build=False --minimize=False
You can also disable these options for all JupyterLab builds by adding these
lines to a Jupyter config file named `jupyter_config.py`:
c.LabBuildApp.minimize = False
c.LabBuildApp.dev_build = False
If you don't already have a `jupyter_config.py` file, you can create one by
adding a blank file of that name to any of the Jupyter config directories.
The config directories can be listed by running:
jupyter --paths
Explanation:
- `dev-build`: This option controls whether a `dev` or a more streamlined
`production` build is used. This option will default to `False` (i.e., the
`production` build) for most users. However, if you have any labextensions
installed from local files, this option will instead default to `True`.
Explicitly setting `dev-build` to `False` will ensure that the `production`
build is used in all circumstances.
- `minimize`: This option controls whether your JS bundle is minified
during the Webpack build, which helps to improve JupyterLab's overall
performance. However, the minifier plugin used by Webpack is very memory
intensive, so turning it off may help the build finish successfully in
low-memory environments.
"""
class LabBuildApp(JupyterApp, DebugLogFileMixin):
    version = version
    description = """
    Build the JupyterLab application
    The application is built in the JupyterLab app directory in `/staging`.
    When the build is complete it is put in the JupyterLab app `/static`
    directory, where it is used to serve the application.
    """
    aliases = build_aliases
    flags = build_flags
    core_config = Instance(CoreConfig, allow_none=True)
    app_dir = Unicode("", config=True, help="The app directory to build in")
    name = Unicode("JupyterLab", config=True, help="The name of the built application")
    version = Unicode("", config=True, help="The version of the built application")
    dev_build = Bool(
        None,
        allow_none=True,
        config=True,
        help="Whether to build in dev mode. Defaults to True (dev mode) if there are any locally linked extensions, else defaults to False (production mode).",
    )
    minimize = Bool(
        True,
        config=True,
        help="Whether to minimize a production build (defaults to True).",
    )
    pre_clean = Bool(
        False, config=True, help="Whether to clean before building (defaults to False)"
    )
    splice_source = Bool(False, config=True, help="Splice source packages into app directory.")
    def start(self):
        app_dir = self.app_dir or get_app_dir()
        app_options = AppOptions(
            app_dir=app_dir,
            logger=self.log,
            core_config=self.core_config,
            splice_source=self.splice_source,
(8)                 )
(8)                 self.log.info("JupyterLab %s", version)
(8)                 with self.debug_logging():
(12)                    if self.pre_clean:
(16)                        self.log.info("Cleaning %s" % app_dir)
(16)                        clean(app_options=app_options)
(12)                    self.log.info("Building in %s", app_dir)
(12)                    try:
(16)                        production = None if self.dev_build is None else not self.dev_build
(16)                        build(
(20)                            name=self.name,
(20)                            version=self.version,
(20)                            app_options=app_options,
(20)                            production=production,
(20)                            minimize=self.minimize,
(16)                        )
(12)                    except Exception as e:
(16)                        self.log.error(build_failure_msg)
(16)                        raise e
(0)         clean_aliases = dict(base_aliases)
(0)         clean_aliases["app-dir"] = "LabCleanApp.app_dir"
(0)         ext_warn_msg = "WARNING: this will delete all of your extensions, which will need to be reinstalled"
(0)         clean_flags = dict(base_flags)
(0)         clean_flags["extensions"] = (
(4)             {"LabCleanApp": {"extensions": True}},
(4)             "Also delete <app-dir>/extensions.\n%s" % ext_warn_msg,
(0)         )
(0)         clean_flags["settings"] = (
(4)             {"LabCleanApp": {"settings": True}},
(4)             "Also delete <app-dir>/settings",
(0)         )
(0)         clean_flags["static"] = (
(4)             {"LabCleanApp": {"static": True}},
(4)             "Also delete <app-dir>/static",
(0)         )
(0)         clean_flags["all"] = (
(4)             {"LabCleanApp": {"all": True}},
(4)             "Delete the entire contents of the app directory.\n%s" % ext_warn_msg,
(0)         )
(0)         class LabCleanAppOptions(AppOptions):
(4)             extensions = Bool(False)
(4)             settings = Bool(False)
(4)             staging = Bool(True)
(4)             static = Bool(False)
(4)             all = Bool(False)
(0)         class LabCleanApp(JupyterApp):
(4)             version = version
(4)             description = """
(4)             Clean the JupyterLab application
(4)             This will clean the app directory by removing the `staging` directories.
(4)             Optionally, the `extensions`, `settings`, and/or `static` directories,
(4)             or the entire contents of the app directory, can also be removed.
(4)             """
(4)             aliases = clean_aliases
(4)             flags = clean_flags
(4)             core_config = Instance(CoreConfig, allow_none=True)
(4)             app_dir = Unicode("", config=True, help="The app directory to clean")
(4)             extensions = Bool(
(8)                 False, config=True, help="Also delete <app-dir>/extensions.\n%s" % ext_warn_msg
(4)             )
(4)             settings = Bool(False, config=True, help="Also delete <app-dir>/settings")
(4)             static = Bool(False, config=True, help="Also delete <app-dir>/static")
(4)             all = Bool(
(8)                 False,
(8)                 config=True,
(8)                 help="Delete the entire contents of the app directory.\n%s" %
```

```
ext_warn_msg,
216: (4)                )
217: (4)                def start(self):
218: (8)                    app_options = LabCleanAppOptions(
219: (12)                       logger=self.log,
220: (12)                       core_config=self.core_config,
221: (12)                       app_dir=self.app_dir,
222: (12)                       extensions=self.extensions,
223: (12)                       settings=self.settings,
224: (12)                       static=self.static,
225: (12)                       all=self.all,
226: (8)                    )
227: (8)                    clean(app_options=app_options)
228: (0)            class LabPathApp(JupyterApp):
229: (4)                version = version
230: (4)                description = """
231: (4)                Print the configured paths for the JupyterLab application
232: (4)                The application path can be configured using the JUPYTERLAB_DIR
233: (8)                    environment variable.
234: (4)                The user settings path can be configured using the JUPYTERLAB_SETTINGS_DIR
235: (8)                    environment variable or it will fall back to
236: (8)                    `/lab/user-settings` in the default Jupyter configuration directory.
237: (4)                The workspaces path can be configured using the JUPYTERLAB_WORKSPACES_DIR
238: (8)                    environment variable or it will fall back to
239: (8)                    '/lab/workspaces' in the default Jupyter configuration directory.
240: (4)                """
241: (4)                def start(self):
242: (8)                    print("Application directory:   %s" % get_app_dir())
243: (8)                    print("User Settings directory: %s" % get_user_settings_dir())
244: (8)                    print("Workspaces directory: %s" % get_workspaces_dir())
245: (0)            class LabWorkspaceExportApp(WorkspaceExportApp):
246: (4)                version = version
247: (4)                @default("workspaces_dir")
248: (4)                def _default_workspaces_dir(self):
249: (8)                    return get_workspaces_dir()
250: (0)            class LabWorkspaceImportApp(WorkspaceImportApp):
251: (4)                version = version
252: (4)                @default("workspaces_dir")
253: (4)                def _default_workspaces_dir(self):
254: (8)                    return get_workspaces_dir()
255: (0)            class LabWorkspaceListApp(WorkspaceListApp):
256: (4)                version = version
257: (4)                @default("workspaces_dir")
258: (4)                def _default_workspaces_dir(self):
259: (8)                    return get_workspaces_dir()
260: (0)            class LabWorkspaceApp(JupyterApp):
261: (4)                version = version
262: (4)                description = """
263: (4)                Import or export a JupyterLab workspace or list all the JupyterLab
workspaces
264: (4)                There are three sub-commands for export, import or listing of workspaces.
This app
265: (8)                    should not otherwise do any work.
266: (4)                """
267: (4)                subcommands = {}
268: (4)                subcommands["export"] = (
269: (8)                    LabWorkspaceExportApp,
270: (8)                    LabWorkspaceExportApp.description.splitlines()[0],
271: (4)                )
272: (4)                subcommands["import"] = (
273: (8)                    LabWorkspaceImportApp,
274: (8)                    LabWorkspaceImportApp.description.splitlines()[0],
275: (4)                )
276: (4)                subcommands["list"] = (
277: (8)                    LabWorkspaceListApp,
278: (8)                    LabWorkspaceListApp.description.splitlines()[0],
279: (4)                )
280: (4)                def start(self):
281: (8)                    try:
```

```
(12)                super().start()
(12)                self.log.error("One of `export`, `import` or `list` must be specified.")
(12)                self.exit(1)
(8)             except NoStart:
(12)                pass
(8)             self.exit(0)
(0)     class LabLicensesApp(LicensesApp):
(4)         version = version
(4)         dev_mode = Bool(
(8)             False,
(8)             config=True,
(8)             help="""Whether to start the app in dev mode. Uses the unpublished local
(8)             JavaScript packages in the `dev_mode` folder.  In this case JupyterLab will
(8)             show a red stripe at the top of the page.  It can only be used if JupyterLab
(8)             is installed as `pip install -e .`.
(8)             """,
(4)         )
(4)         app_dir = Unicode("", config=True, help="The app directory for which to show licenses")
(4)         aliases = {
(8)             **LicensesApp.aliases,
(8)             "app-dir": "LabLicensesApp.app_dir",
(4)         }
(4)         flags = {
(8)             **LicensesApp.flags,
(8)             "dev-mode": (
(12)                {"LabLicensesApp": {"dev_mode": True}},
(12)                "Start the app in dev mode for running from source.",
(8)             ),
(4)         }
(4)         @default("app_dir")
(4)         def _default_app_dir(self):
(8)             return get_app_dir()
(4)         @default("static_dir")
(4)         def _default_static_dir(self):
(8)             return pjoin(self.app_dir, "static")
(0)     aliases = dict(base_aliases)
(0)     aliases.update(
(4)         {
(8)             "ip": "ServerApp.ip",
(8)             "port": "ServerApp.port",
(8)             "port-retries": "ServerApp.port_retries",
(8)             "keyfile": "ServerApp.keyfile",
(8)             "certfile": "ServerApp.certfile",
(8)             "client-ca": "ServerApp.client_ca",
(8)             "notebook-dir": "ServerApp.root_dir",
(8)             "browser": "ServerApp.browser",
(8)             "pylab": "ServerApp.pylab",
(4)         }
(0)     )
(0)     class LabApp(NotebookConfigShimMixin, LabServerApp):
(4)         version = version
(4)         name = "lab"
(4)         app_name = "JupyterLab"
(4)         load_other_extensions = True
(4)         description = """
(4)         JupyterLab - An extensible computational environment for Jupyter.
(4)         This launches a Tornado based HTML Server that serves up an
(4)         HTML5/Javascript JupyterLab client.
(4)         JupyterLab has three different modes of running:
(4)         * Core mode (`--core-mode`): in this mode JupyterLab will run using the JavaScript
(6)           assets contained in the installed `jupyterlab` Python package. In core mode, no
(6)           extensions are enabled. This is the default in a stable JupyterLab
```

```
release if you
344: (6)                  have no extensions installed.
345: (4)                * Dev mode (`--dev-mode`): uses the unpublished local JavaScript packages
in the
346: (6)                  `dev_mode` folder.  In this case JupyterLab will show a red stripe at
the top of
347: (6)                  the page.  It can only be used if JupyterLab is installed as `pip
install -e .`.
348: (4)                * App mode: JupyterLab allows multiple JupyterLab "applications" to be
349: (6)                  created by the user with different combinations of extensions. The `--
app-dir` can
350: (6)                  be used to set a directory for different applications. The default
application
351: (6)                  path can be found using `jupyter lab path`.
352: (4)                """
353: (4)                examples = """
354: (8)                    jupyter lab                       # start JupyterLab
355: (8)                    jupyter lab --dev-mode            # start JupyterLab in development
mode, with no extensions
356: (8)                    jupyter lab --core-mode           # start JupyterLab in core mode,
with no extensions
357: (8)                    jupyter lab --app-dir=~/myjupyterlabapp # start JupyterLab with a
particular set of extensions
358: (8)                    jupyter lab --certfile=mycert.pem # use SSL/TLS certificate
359: (4)                """
360: (4)                aliases = aliases
361: (4)                aliases.update(
362: (8)                    {
363: (12)                       "watch": "LabApp.watch",
364: (8)                    }
365: (4)                )
366: (4)                aliases["app-dir"] = "LabApp.app_dir"
367: (4)                flags = flags
368: (4)                flags["core-mode"] = (
369: (8)                    {"LabApp": {"core_mode": True}},
370: (8)                    "Start the app in core mode.",
371: (4)                )
372: (4)                flags["dev-mode"] = (
373: (8)                    {"LabApp": {"dev_mode": True}},
374: (8)                    "Start the app in dev mode for running from source.",
375: (4)                )
376: (4)                flags["skip-dev-build"] = (
377: (8)                    {"LabApp": {"skip_dev_build": True}},
378: (8)                    "Skip the initial install and JS build of the app in dev mode.",
379: (4)                )
380: (4)                flags["watch"] = ({"LabApp": {"watch": True}}, "Start the app in watch
mode.")
381: (4)                flags["splice-source"] = (
382: (8)                    {"LabApp": {"splice_source": True}},
383: (8)                    "Splice source packages into app directory.",
384: (4)                )
385: (4)                flags["expose-app-in-browser"] = (
386: (8)                    {"LabApp": {"expose_app_in_browser": True}},
387: (8)                    "Expose the global app instance to browser via window.jupyterapp.",
388: (4)                )
389: (4)                flags["extensions-in-dev-mode"] = (
390: (8)                    {"LabApp": {"extensions_in_dev_mode": True}},
391: (8)                    "Load prebuilt extensions in dev-mode.",
392: (4)                )
393: (4)                flags["collaborative"] = (
394: (8)                    {"LabApp": {"collaborative": True}},
395: (8)                    """To enable real-time collaboration, you must install the extension
`jupyter_collaboration`.
396: (8)                    You can install it using pip for example:
397: (12)                       python -m pip install jupyter_collaboration
398: (8)                    This flag is now deprecated and will be removed in JupyterLab v5.""",
399: (4)                )
400: (4)                flags["custom-css"] = (
401: (8)                    {"LabApp": {"custom_css": True}},
(8)                         "Load custom CSS in template html files. Default is False",
(4)                     )
(4)                     subcommands = {
(8)                         "build": (LabBuildApp, LabBuildApp.description.splitlines()[0]),
(8)                         "clean": (LabCleanApp, LabCleanApp.description.splitlines()[0]),
(8)                         "path": (LabPathApp, LabPathApp.description.splitlines()[0]),
(8)                         "paths": (LabPathApp, LabPathApp.description.splitlines()[0]),
(8)                         "workspace": (LabWorkspaceApp, LabWorkspaceApp.description.splitlines()[0]),
(8)                         "workspaces": (LabWorkspaceApp, LabWorkspaceApp.description.splitlines()[0]),
(8)                         "licenses": (LabLicensesApp, LabLicensesApp.description.splitlines()[0]),
(4)                     }
(4)                     default_url = Unicode("/lab", config=True, help="The default URL to redirect to from `/`")
(4)                     override_static_url = Unicode(
(8)                         config=True, help=("The override url for static lab assets, typically a CDN.")
(4)                     )
(4)                     override_theme_url = Unicode(
(8)                         config=True,
(8)                         help=("The override url for static lab theme assets, typically a CDN."),
(4)                     )
(4)                     app_dir = Unicode(None, config=True, help="The app directory to launch JupyterLab from.")
(4)                     user_settings_dir = Unicode(
(8)                         get_user_settings_dir(), config=True, help="The directory for user settings."
(4)                     )
(4)                     workspaces_dir = Unicode(get_workspaces_dir(), config=True, help="The directory for workspaces")
(4)                     core_mode = Bool(
(8)                         False,
(8)                         config=True,
(8)                         help="""Whether to start the app in core mode. In this mode, JupyterLab
(8)                         will run using the JavaScript assets that are within the installed
(8)                         JupyterLab Python package. In core mode, third party extensions are disabled.
(8)                         The `--dev-mode` flag is an alias to this to be used when the Python package
(8)                         itself is installed in development mode (`pip install -e .`).
(8)                         """,
(4)                     )
(4)                     dev_mode = Bool(
(8)                         False,
(8)                         config=True,
(8)                         help="""Whether to start the app in dev mode. Uses the unpublished local
(8)                         JavaScript packages in the `dev_mode` folder.  In this case JupyterLab will
(8)                         show a red stripe at the top of the page.  It can only be used if JupyterLab
(8)                         is installed as `pip install -e .`.
(8)                         """,
(4)                     )
(4)                     extensions_in_dev_mode = Bool(
(8)                         False,
(8)                         config=True,
(8)                         help="""Whether to load prebuilt extensions in dev mode. This may be
(8)                         useful to run and test prebuilt extensions in development installs of
(8)                         JupyterLab. APIs in a JupyterLab development install may be
(8)                         incompatible with published packages, so prebuilt extensions compiled
(8)                         against published packages may not work correctly.""",
(4)                     )
(4)                     extension_manager = Unicode(
(8)                         "pypi",
        config=True,
        help="""The extension manager factory to use. The default options are:
        "readonly" for a manager without installation capability or "pypi" for
        a manager using PyPi.org and pip to install extensions.""",
    )
    watch = Bool(False, config=True, help="Whether to serve the app in watch mode")
    skip_dev_build = Bool(
        False,
        config=True,
        help="Whether to skip the initial install and JS build of the app in dev mode",
    )
    splice_source = Bool(False, config=True, help="Splice source packages into app directory.")
    expose_app_in_browser = Bool(
        False,
        config=True,
        help="Whether to expose the global app instance to browser via window.jupyterapp",
    )
    custom_css = Bool(
        False,
        config=True,
        help="""Whether custom CSS is loaded on the page.
    Defaults to False.
    """,
    )
    collaborative = Bool(
        False,
        config=True,
        help="""To enable real-time collaboration, you must install the extension `jupyter_collaboration`.
        You can install it using pip for example:
            python -m pip install jupyter_collaboration
        This flag is now deprecated and will be removed in JupyterLab v5.""",
    )
    news_url = Unicode(
        "https://jupyterlab.github.io/assets/feed.xml",
        allow_none=True,
        help="""URL that serves news Atom feed; by default the JupyterLab organization announcements will be fetched. Set to None to turn off fetching announcements.""",
        config=True,
    )
    lock_all_plugins = Bool(
        False,
        config=True,
        help="Whether all plugins are locked (cannot be enabled/disabled from the UI)",
    )
    check_for_updates_class = Type(
        default_value=CheckForUpdate,
        klass=CheckForUpdateABC,
        config=True,
        help="""A callable class that receives the current version at instantiation and calling it must return asynchronously a string indicating which version is available and how to install or None if no update is available. The string supports Markdown format.""",
    )
    @default("app_dir")
    def _default_app_dir(self):
        app_dir = get_app_dir()
        if self.core_mode:
            app_dir = HERE
        elif self.dev_mode:
            app_dir = DEV_DIR
        return app_dir
    @default("app_settings_dir")
    def _default_app_settings_dir(self):
```

```
(8)                 return pjoin(self.app_dir, "settings")
(4)             @default("app_version")
(4)             def _default_app_version(self):
(8)                 return app_version
(4)             @default("cache_files")
(4)             def _default_cache_files(self):
(8)                 return False
(4)             @default("schemas_dir")
(4)             def _default_schemas_dir(self):
(8)                 return pjoin(self.app_dir, "schemas")
(4)             @default("templates_dir")
(4)             def _default_templates_dir(self):
(8)                 return pjoin(self.app_dir, "static")
(4)             @default("themes_dir")
(4)             def _default_themes_dir(self):
(8)                 if self.override_theme_url:
(12)                    return ""
(8)                 return pjoin(self.app_dir, "themes")
(4)             @default("static_dir")
(4)             def _default_static_dir(self):
(8)                 return pjoin(self.app_dir, "static")
(4)             @default("static_url_prefix")
(4)             def _default_static_url_prefix(self):
(8)                 if self.override_static_url:
(12)                    return self.override_static_url
(8)                 else:
(12)                    static_url = f"/static/{self.name}/"
(12)                    return ujoin(self.serverapp.base_url, static_url)
(4)             @default("theme_url")
(4)             def _default_theme_url(self):
(8)                 if self.override_theme_url:
(12)                    return self.override_theme_url
(8)                 return ""
(4)             def initialize_templates(self):
(8)                 if self.core_mode or self.app_dir.startswith(HERE + os.sep):
(12)                    self.core_mode = True
(12)                    self.log.info("Running JupyterLab in core mode")
(8)                 if self.dev_mode or self.app_dir.startswith(DEV_DIR + os.sep):
(12)                    self.dev_mode = True
(12)                    self.log.info("Running JupyterLab in dev mode")
(8)                 if self.watch and self.core_mode:
(12)                    self.log.warning("Cannot watch in core mode, did you mean --dev-mode?")
(12)                    self.watch = False
(8)                 if self.core_mode and self.dev_mode:
(12)                    self.log.warning("Conflicting modes, choosing dev_mode over core_mode")
(12)                    self.core_mode = False
(8)                 if self.dev_mode:
(12)                    dev_static_dir = ujoin(DEV_DIR, "static")
(12)                    self.static_paths = [dev_static_dir]
(12)                    self.template_paths = [dev_static_dir]
(12)                    if not self.extensions_in_dev_mode:
(16)                        galata_extension = pjoin(HERE, "galata")
(16)                        self.labextensions_path = (
(20)                            [galata_extension]
(20)                            if galata_extension in map(os.path.abspath, self.labextensions_path)
(20)                            else []
(16)                        )
(16)                        self.extra_labextensions_path = (
(20)                            [galata_extension]
(20)                            if galata_extension in map(os.path.abspath, self.extra_labextensions_path)
(20)                            else []
(16)                        )
(8)                 elif self.core_mode:
(12)                    dev_static_dir = ujoin(HERE, "static")
(12)                    self.static_paths = [dev_static_dir]
```

```
580: (12)                       self.template_paths = [dev_static_dir]
581: (12)                       self.labextensions_path = []
582: (12)                       self.extra_labextensions_path = []
583: (8)                    else:
584: (12)                       self.static_paths = [self.static_dir]
585: (12)                       self.template_paths = [self.templates_dir]
586: (4)                def _prepare_templates(self):
587: (8)                    super()._prepare_templates()
588: (8)                    self.jinja2_env.globals.update(custom_css=self.custom_css)
589: (4)                def initialize_handlers(self):  # noqa
590: (8)                    handlers = []
591: (8)                    page_config =
self.serverapp.web_app.settings.setdefault("page_config_data", {})
592: (8)                    page_config.setdefault("buildAvailable", not self.core_mode and not
self.dev_mode)
593: (8)                    page_config.setdefault("buildCheck", not self.core_mode and not
self.dev_mode)
594: (8)                    page_config["devMode"] = self.dev_mode
595: (8)                    page_config["token"] = self.serverapp.identity_provider.token
596: (8)                    page_config["exposeAppInBrowser"] = self.expose_app_in_browser
597: (8)                    page_config["quitButton"] = self.serverapp.quit_button
598: (8)                    page_config["allow_hidden_files"] =
self.serverapp.contents_manager.allow_hidden
599: (8)                    page_config["notebookVersion"] = json.dumps(jpserver_version_info)
600: (8)                    self.log.info("JupyterLab extension loaded from %s" % HERE)
601: (8)                    self.log.info("JupyterLab application directory is %s" % self.app_dir)
602: (8)                    if self.custom_css:
603: (12)                       handlers.append(
604: (16)                           (
605: (20)                               r"/custom/(.*)(?<!\.js)$",
606: (20)                               self.serverapp.web_app.settings["static_handler_class"],
607: (20)                               {
608: (24)                                   "path":
self.serverapp.web_app.settings["static_custom_path"],
609: (24)                                   "no_cache_paths": ["/"],  # don't cache anything in
custom
610: (20)                               },
611: (16)                           )
612: (12)                       )
613: (8)                    app_options = AppOptions(
614: (12)                       logger=self.log,
615: (12)                       app_dir=self.app_dir,
616: (12)                       labextensions_path=self.extra_labextensions_path +
self.labextensions_path,
617: (12)                       splice_source=self.splice_source,
618: (8)                    )
619: (8)                    builder = Builder(self.core_mode, app_options=app_options)
620: (8)                    build_handler = (build_path, BuildHandler, {"builder": builder})
621: (8)                    handlers.append(build_handler)
622: (8)                    errored = False
623: (8)                    if self.core_mode:
624: (12)                       self.log.info(CORE_NOTE.strip())
625: (12)                       ensure_core(self.log)
626: (8)                    elif self.dev_mode:
627: (12)                       if not (self.watch or self.skip_dev_build):
628: (16)                           ensure_dev(self.log)
629: (16)                           self.log.info(DEV_NOTE)
630: (8)                    else:
631: (12)                       if self.splice_source:
632: (16)                           ensure_dev(self.log)
633: (12)                       msgs = ensure_app(self.app_dir)
634: (12)                       if msgs:
635: (16)                           [self.log.error(msg) for msg in msgs]
636: (16)                           handler = (self.app_url, ErrorHandler, {"messages": msgs})
637: (16)                           handlers.append(handler)
638: (16)                           errored = True
639: (8)                    if self.watch:
640: (12)                       self.log.info("Starting JupyterLab watch mode...")
641: (12)                       if self.dev_mode:
```

```
642: (16)                           watch_dev(self.log)
643: (12)                       else:
644: (16)                           watch(app_options=app_options)
645: (16)                           page_config["buildAvailable"] = False
646: (12)                       self.cache_files = False
647: (8)                    if not self.core_mode and not errored:
648: (12)                       provider = self.extension_manager
649: (12)                       entry_point = EXT_MANAGERS.get(provider)
650: (12)                       if entry_point is None:
651: (16)                           self.log.error(f"Extension Manager: No manager defined for
provider '{provider}'.")
652: (16)                           raise NotImplementedError()
653: (12)                       else:
654: (16)                           self.log.info(f"Extension Manager is '{provider}'.")
655: (12)                       manager_factory = entry_point.load()
656: (12)                       config = self.settings.get("config", {}).get("LabServerApp", {})
657: (12)                       blocked_extensions_uris = config.get("blocked_extensions_uris",
"")
658: (12)                       allowed_extensions_uris = config.get("allowed_extensions_uris",
"")
659: (12)                       if (blocked_extensions_uris) and (allowed_extensions_uris):
660: (16)                           self.log.error(
661: (20)                               "Simultaneous LabServerApp.blocked_extensions_uris and
LabServerApp.allowed_extensions_uris is not supported. Please define only one of those."
662: (16)                           )
663: (16)                           import sys
664: (16)                           sys.exit(-1)
665: (12)                       listings_config = {
666: (16)                           "blocked_extensions_uris": set(
667: (20)                               filter(lambda uri: len(uri) > 0,
blocked_extensions_uris.split(","))
668: (16)                           ),
669: (16)                           "allowed_extensions_uris": set(
670: (20)                               filter(lambda uri: len(uri) > 0,
allowed_extensions_uris.split(","))
671: (16)                           ),
672: (16)                           "listings_refresh_seconds":
config.get("listings_refresh_seconds", 60 * 60),
673: (16)                           "listings_tornado_options":
config.get("listings_tornado_options", {}),
674: (12)                       }
675: (12)                       if len(listings_config["blocked_extensions_uris"]) or len(
676: (16)                           listings_config["allowed_extensions_uris"]
677: (12)                       ):
678: (16)                           self.log.debug(f"Extension manager will be constrained by
{listings_config}")
679: (12)                       try:
680: (16)                           ext_manager = manager_factory(app_options, listings_config,
self)
681: (16)                           metadata = dataclasses.asdict(ext_manager.metadata)
682: (12)                       except Exception as err:
683: (16)                           self.log.warning(
684: (20)                               f"Failed to instantiate the extension manager {provider}.
Falling back to read-only manager.",
685: (20)                               exc_info=err,
686: (16)                           )
687: (16)                           ext_manager = ReadOnlyExtensionManager(app_options,
listings_config, self)
688: (16)                           metadata = dataclasses.asdict(ext_manager.metadata)
689: (12)                       page_config["extensionManager"] = metadata
690: (12)                       ext_handler = (
691: (16)                           extensions_handler_path,
692: (16)                           ExtensionHandler,
693: (16)                           {"manager": ext_manager},
694: (12)                       )
695: (12)                       handlers.append(ext_handler)
696: (12)                       lock_rules = frozenset(
697: (16)                           {rule for rule, value in page_config.get("lockedExtensions",
{}).items() if value}
698: (12)                           )
699: (12)                           handlers.append(
700: (16)                               (
701: (20)                                   plugins_handler_path,
702: (20)                                   PluginHandler,
703: (20)                                   {
704: (24)                                       "manager": PluginManager(
705: (28)                                           app_options=app_options,
706: (28)                                           ext_options={
707: (32)                                               "lock_rules": lock_rules,
708: (32)                                               "all_locked": self.lock_all_plugins,
709: (28)                                           },
710: (28)                                           parent=self,
711: (24)                                       )
712: (20)                                   },
713: (16)                               )
714: (12)                           )
715: (12)                           page_config["news"] = {"disabled": self.news_url is None}
716: (12)                           handlers.extend(
717: (16)                               [
718: (20)                                   (
719: (24)                                       check_update_handler_path,
720: (24)                                       CheckForUpdateHandler,
721: (24)                                       {
722: (28)                                           "update_checker":
self.check_for_updates_class(__version__),
723: (24)                                       },
724: (20)                                   ),
725: (20)                                   (
726: (24)                                       news_handler_path,
727: (24)                                       NewsHandler,
728: (24)                                       {
729: (28)                                           "news_url": self.news_url,
730: (24)                                       },
731: (20)                                   ),
732: (16)                               ]
733: (12)                           )
734: (8)                        if "hub_prefix" in self.serverapp.tornado_settings:
735: (12)                           tornado_settings = self.serverapp.tornado_settings
736: (12)                           hub_prefix = tornado_settings["hub_prefix"]
737: (12)                           page_config["hubPrefix"] = hub_prefix
738: (12)                           page_config["hubHost"] = tornado_settings["hub_host"]
739: (12)                           page_config["hubUser"] = tornado_settings["user"]
740: (12)                           page_config["shareUrl"] = ujoin(hub_prefix, "user-redirect")
741: (12)                           if hasattr(self.serverapp, "server_name"):
742: (16)                               page_config["hubServerName"] = self.serverapp.server_name
743: (12)                           page_config["token"] = ""
744: (8)                        self.serverapp.web_app.settings["page_config_data"] = page_config
745: (8)                        self.handlers.extend(handlers)
746: (8)                        super().initialize_handlers()
747: (4)                    def initialize(self, argv=None):
748: (8)                        """Subclass because the ExtensionApp.initialize() method does not take
arguments"""
749: (8)                        super().initialize()
750: (8)                        if self.collaborative:
751: (12)                           try:
752: (16)                               import jupyter_collaboration  # noqa
753: (12)                           except ImportError:
754: (16)                               self.log.critical(
755: (20)                                   """\
756: (0)                Jupyter Lab cannot start, because `jupyter_collaboration` was configured but
cannot be `import`ed.
757: (0)                To fix this, either:
758: (0)                1) install the extension `jupyter-collaboration`; for example: `python -m pip
install jupyter-collaboration`
759: (0)                2) disable collaboration; for example, remove the `--collaborative` flag from
the commandline.  To see more ways to adjust the collaborative behavior, see https://jupyterlab-
realtime-collaboration.readthedocs.io/en/latest/configuration.html .
760: (0)                """
```

```
(16)                            )
(16)                            sys.exit(1)
(0)             main = launch_new_instance = LabApp.launch_instance
(0)             if __name__ == "__main__":
(4)                 main()
```

File 8 - labextensions.py:

```
(0)             """Jupyter LabExtension Entry Points."""
(0)             import os
(0)             import sys
(0)             from copy import copy
(0)             from jupyter_core.application import JupyterApp, base_aliases, base_flags
(0)             from traitlets import Bool, Instance, List, Unicode, default
(0)             from jupyterlab.coreconfig import CoreConfig
(0)             from jupyterlab.debuglog import DebugLogFileMixin
(0)             from .commands import (
(4)                 HERE,
(4)                 AppOptions,
(4)                 build,
(4)                 check_extension,
(4)                 disable_extension,
(4)                 enable_extension,
(4)                 get_app_version,
(4)                 install_extension,
(4)                 link_package,
(4)                 list_extensions,
(4)                 lock_extension,
(4)                 uninstall_extension,
(4)                 unlink_package,
(4)                 unlock_extension,
(4)                 update_extension,
(0)             )
(0)             from .federated_labextensions import build_labextension, develop_labextension_py, watch_labextension
(0)             from .labapp import LabApp
(0)             flags = dict(base_flags)
(0)             flags["no-build"] = (
(4)                 {"BaseExtensionApp": {"should_build": False}},
(4)                 "Defer building the app after the action.",
(0)             )
(0)             flags["dev-build"] = (
(4)                 {"BaseExtensionApp": {"dev_build": True}},
(4)                 "Build in development mode.",
(0)             )
(0)             flags["no-minimize"] = (
(4)                 {"BaseExtensionApp": {"minimize": False}},
(4)                 "Do not minimize a production build.",
(0)             )
(0)             flags["clean"] = (
(4)                 {"BaseExtensionApp": {"should_clean": True}},
(4)                 "Cleanup intermediate files after the action.",
(0)             )
(0)             flags["splice-source"] = (
(4)                 {"BaseExtensionApp": {"splice_source": True}},
(4)                 "Splice source packages into app directory.",
(0)             )
(0)             check_flags = copy(flags)
(0)             check_flags["installed"] = (
(4)                 {"CheckLabExtensionsApp": {"should_check_installed_only": True}},
(4)                 "Check only if the extension is installed.",
(0)             )
(0)             develop_flags = copy(flags)
(0)             develop_flags["overwrite"] = (
(4)                 {"DevelopLabExtensionApp": {"overwrite": True}},
(4)                 "Overwrite files",
(0)             )
```

```
59: (0)             update_flags = copy(flags)
60: (0)             update_flags["all"] = (
61: (4)                 {"UpdateLabExtensionApp": {"all": True}},
62: (4)                 "Update all extensions",
63: (0)             )
64: (0)             uninstall_flags = copy(flags)
65: (0)             uninstall_flags["all"] = (
66: (4)                 {"UninstallLabExtensionApp": {"all": True}},
67: (4)                 "Uninstall all extensions",
68: (0)             )
69: (0)             list_flags = copy(flags)
70: (0)             list_flags["verbose"] = (
71: (4)                 {"ListLabExtensionsApp": {"verbose": True}},
72: (4)                 "Increase verbosity level",
73: (0)             )
74: (0)             aliases = dict(base_aliases)
75: (0)             aliases["app-dir"] = "BaseExtensionApp.app_dir"
76: (0)             aliases["dev-build"] = "BaseExtensionApp.dev_build"
77: (0)             aliases["minimize"] = "BaseExtensionApp.minimize"
78: (0)             aliases["debug-log-path"] = "DebugLogFileMixin.debug_log_path"
79: (0)             install_aliases = copy(aliases)
80: (0)             install_aliases["pin-version-as"] = "InstallLabExtensionApp.pin"
81: (0)             enable_aliases = copy(aliases)
82: (0)             enable_aliases["level"] = "EnableLabExtensionsApp.level"
83: (0)             disable_aliases = copy(aliases)
84: (0)             disable_aliases["level"] = "DisableLabExtensionsApp.level"
85: (0)             lock_aliases = copy(aliases)
86: (0)             lock_aliases["level"] = "LockLabExtensionsApp.level"
87: (0)             unlock_aliases = copy(aliases)
88: (0)             unlock_aliases["level"] = "UnlockLabExtensionsApp.level"
89: (0)             VERSION = get_app_version()
90: (0)             LABEXTENSION_COMMAND_WARNING = "Users should manage prebuilt extensions with
package managers like pip and conda, and extension authors are encouraged to distribute their
extensions as prebuilt packages"
91: (0)             class BaseExtensionApp(JupyterApp, DebugLogFileMixin):
92: (4)                 version = VERSION
93: (4)                 flags = flags
94: (4)                 aliases = aliases
95: (4)                 name = "lab"
96: (4)                 core_config = Instance(CoreConfig, allow_none=True)
97: (4)                 app_dir = Unicode("", config=True, help="The app directory to target")
98: (4)                 should_build = Bool(True, config=True, help="Whether to build the app
after the action")
99: (4)                 dev_build = Bool(
100: (8)                    None,
101: (8)                    allow_none=True,
102: (8)                    config=True,
103: (8)                    help="Whether to build in dev mode. Defaults to True (dev mode) if
there are any locally linked extensions, else defaults to False (production mode).",
104: (4)                )
105: (4)                minimize = Bool(
106: (8)                    True,
107: (8)                    config=True,
108: (8)                    help="Whether to minimize a production build (defaults to True).",
109: (4)                )
110: (4)                should_clean = Bool(
111: (8)                    False,
112: (8)                    config=True,
113: (8)                    help="Whether temporary files should be cleaned up after building
jupyterlab",
114: (4)                )
115: (4)                splice_source = Bool(False, config=True, help="Splice source packages into
app directory.")
116: (4)                labextensions_path = List(
117: (8)                    Unicode(),
118: (8)                    help="The standard paths to look in for prebuilt JupyterLab
extensions",
119: (4)                )
120: (4)                @default("labextensions_path")
```

```
121: (4)                def _default_labextensions_path(self):
122: (8)                    lab = LabApp()
123: (8)                    lab.load_config_file()
124: (8)                    return lab.extra_labextensions_path + lab.labextensions_path
125: (4)                @default("splice_source")
126: (4)                def _default_splice_source(self):
127: (8)                    version = get_app_version(AppOptions(app_dir=self.app_dir))
128: (8)                    return version.endswith("-spliced")
129: (4)                def start(self):
130: (8)                    if self.app_dir and self.app_dir.startswith(HERE):
131: (12)                       msg = "Cannot run lab extension commands in core app"
132: (12)                       raise ValueError(msg)
133: (8)                    with self.debug_logging():
134: (12)                       ans = self.run_task()
135: (12)                       if ans and self.should_build:
136: (16)                           production = None if self.dev_build is None else not self.dev_build
137: (16)                           app_options = AppOptions(
138: (20)                               app_dir=self.app_dir,
139: (20)                               logger=self.log,
140: (20)                               core_config=self.core_config,
141: (20)                               splice_source=self.splice_source,
142: (16)                           )
143: (16)                           build(
144: (20)                               clean_staging=self.should_clean,
145: (20)                               production=production,
146: (20)                               minimize=self.minimize,
147: (20)                               app_options=app_options,
148: (16)                           )
149: (4)                def run_task(self):
150: (8)                    pass
151: (4)                def deprecation_warning(self, msg):
152: (8)                    return self.log.warning(
153: (12)                       "\033[33m(Deprecated) %s\n\n%s \033[0m", msg, LABEXTENSION_COMMAND_WARNING
154: (8)                    )
155: (4)                def _log_format_default(self):
156: (8)                    """A default format for messages"""
157: (8)                    return "%(message)s"
158: (0)            class InstallLabExtensionApp(BaseExtensionApp):
159: (4)                description = """Install labextension(s)
160: (5)                 Usage
161: (8)                    jupyter labextension install [--pin-version-as <alias,...>] <package...>
162: (4)                This installs JupyterLab extensions similar to yarn add or npm install.
163: (4)                Pass a list of comma separate names to the --pin-version-as flag
164: (4)                to use as aliases for the packages providers. This is useful to
165: (4)                install multiple versions of the same extension.
166: (4)                These can be uninstalled with the alias you provided
167: (4)                to the flag, similar to the "alias" feature of yarn add.
168: (4)                """
169: (4)                aliases = install_aliases
170: (4)                pin = Unicode("", config=True, help="Pin this version with a certain alias")
171: (4)                def run_task(self):
172: (8)                    self.deprecation_warning(
173: (12)                       "Installing extensions with the jupyter labextension install command is now deprecated and will be removed in a future major version of JupyterLab."
174: (8)                    )
175: (8)                    pinned_versions = self.pin.split(",")
176: (8)                    self.extra_args = self.extra_args or [os.getcwd()]
177: (8)                    return any(
178: (12)                       install_extension(
179: (16)                           arg,
180: (16)                           pin=pinned_versions[i] if i < len(pinned_versions) else None,
181: (16)                           app_options=AppOptions(
182: (20)                               app_dir=self.app_dir,
183: (20)                               logger=self.log,
184: (20)                               core_config=self.core_config,
(20)                    labextensions_path=self.labextensions_path,
(16)                ),
(12)            )
(12)            for i, arg in enumerate(self.extra_args)
(8)         )
(0) class DevelopLabExtensionApp(BaseExtensionApp):
(4)     description = "(developer) Develop labextension"
(4)     flags = develop_flags
(4)     user = Bool(False, config=True, help="Whether to do a user install")
(4)     sys_prefix = Bool(True, config=True, help="Use the sys.prefix as the prefix")
(4)     overwrite = Bool(False, config=True, help="Whether to overwrite files")
(4)     symlink = Bool(True, config=False, help="Whether to use a symlink")
(4)     labextensions_dir = Unicode(
(8)         "",
(8)         config=True,
(8)         help="Full path to labextensions dir (probably use prefix or user)",
(4)     )
(4)     def run_task(self):
(8)         """Add config for this labextension"""
(8)         self.extra_args = self.extra_args or [os.getcwd()]
(8)         for arg in self.extra_args:
(12)            develop_labextension_py(
(16)                arg,
(16)                user=self.user,
(16)                sys_prefix=self.sys_prefix,
(16)                labextensions_dir=self.labextensions_dir,
(16)                logger=self.log,
(16)                overwrite=self.overwrite,
(16)                symlink=self.symlink,
(12)            )
(0) class BuildLabExtensionApp(BaseExtensionApp):
(4)     description = "(developer) Build labextension"
(4)     static_url = Unicode("", config=True, help="Sets the url for static assets when building")
(4)     development = Bool(False, config=True, help="Build in development mode")
(4)     source_map = Bool(False, config=True, help="Generate source maps")
(4)     core_path = Unicode(
(8)         os.path.join(HERE, "staging"),
(8)         config=True,
(8)         help="Directory containing core application package.json file",
(4)     )
(4)     aliases = {
(8)         "static-url": "BuildLabExtensionApp.static_url",
(8)         "development": "BuildLabExtensionApp.development",
(8)         "source-map": "BuildLabExtensionApp.source_map",
(8)         "core-path": "BuildLabExtensionApp.core_path",
(4)     }
(4)     def run_task(self):
(8)         self.extra_args = self.extra_args or [os.getcwd()]
(8)         build_labextension(
(12)            self.extra_args[0],
(12)            logger=self.log,
(12)            development=self.development,
(12)            static_url=self.static_url or None,
(12)            source_map=self.source_map,
(12)            core_path=self.core_path or None,
(8)         )
(0) class WatchLabExtensionApp(BaseExtensionApp):
(4)     description = "(developer) Watch labextension"
(4)     development = Bool(True, config=True, help="Build in development mode")
(4)     source_map = Bool(False, config=True, help="Generate source maps")
(4)     core_path = Unicode(
(8)         os.path.join(HERE, "staging"),
(8)         config=True,
(8)         help="Directory containing core application package.json file",
(4)     )
(4)     aliases = {
(8)         "core-path": "WatchLabExtensionApp.core_path",
252: (8)                    "development": "WatchLabExtensionApp.development",
253: (8)                    "source-map": "WatchLabExtensionApp.source_map",
254: (4)                }
255: (4)                def run_task(self):
256: (8)                    self.extra_args = self.extra_args or [os.getcwd()]
257: (8)                    labextensions_path = self.labextensions_path
258: (8)                    watch_labextension(
259: (12)                       self.extra_args[0],
260: (12)                       labextensions_path,
261: (12)                       logger=self.log,
262: (12)                       development=self.development,
263: (12)                       source_map=self.source_map,
264: (12)                       core_path=self.core_path or None,
265: (8)                    )
266: (0)            class UpdateLabExtensionApp(BaseExtensionApp):
267: (4)                description = "Update labextension(s)"
268: (4)                flags = update_flags
269: (4)                all = Bool(False, config=True, help="Whether to update all extensions")
270: (4)                def run_task(self):
271: (8)                    self.deprecation_warning(
272: (12)                       "Updating extensions with the jupyter labextension update command
is now deprecated and will be removed in a future major version of JupyterLab."
273: (8)                    )
274: (8)                    if not self.all and not self.extra_args:
275: (12)                       self.log.warning(
276: (16)                           "Specify an extension to update, or use --all to update all
extensions"
277: (12)                       )
278: (12)                       return False
279: (8)                    app_options = AppOptions(
280: (12)                       app_dir=self.app_dir,
281: (12)                       logger=self.log,
282: (12)                       core_config=self.core_config,
283: (12)                       labextensions_path=self.labextensions_path,
284: (8)                    )
285: (8)                    if self.all:
286: (12)                       return update_extension(all_=True, app_options=app_options)
287: (8)                    return any(update_extension(name=arg, app_options=app_options) for arg
in self.extra_args)
288: (0)            class LinkLabExtensionApp(BaseExtensionApp):
289: (4)                description = """
290: (4)                Link local npm packages that are not lab extensions.
291: (4)                Links a package to the JupyterLab build process. A linked
292: (4)                package is manually re-installed from its source location when
293: (4)                `jupyter lab build` is run.
294: (4)                """
295: (4)                should_build = Bool(True, config=True, help="Whether to build the app
after the action")
296: (4)                def run_task(self):
297: (8)                    self.extra_args = self.extra_args or [os.getcwd()]
298: (8)                    options = AppOptions(
299: (12)                       app_dir=self.app_dir,
300: (12)                       logger=self.log,
301: (12)                       labextensions_path=self.labextensions_path,
302: (12)                       core_config=self.core_config,
303: (8)                    )
304: (8)                    return any(link_package(arg, app_options=options) for arg in
self.extra_args)
305: (0)            class UnlinkLabExtensionApp(BaseExtensionApp):
306: (4)                description = "Unlink packages by name or path"
307: (4)                def run_task(self):
308: (8)                    self.extra_args = self.extra_args or [os.getcwd()]
309: (8)                    options = AppOptions(
310: (12)                       app_dir=self.app_dir,
311: (12)                       logger=self.log,
312: (12)                       labextensions_path=self.labextensions_path,
313: (12)                       core_config=self.core_config,
314: (8)                    )
315: (8)                    return any(unlink_package(arg, app_options=options) for arg in
self.extra_args)
316: (0)            class UninstallLabExtensionApp(BaseExtensionApp):
317: (4)                description = "Uninstall labextension(s) by name"
318: (4)                flags = uninstall_flags
319: (4)                all = Bool(False, config=True, help="Whether to uninstall all extensions")
320: (4)                def run_task(self):
321: (8)                    self.deprecation_warning(
322: (12)                       "Uninstalling extensions with the jupyter labextension uninstall
command is now deprecated and will be removed in a future major version of JupyterLab."
323: (8)                    )
324: (8)                    self.extra_args = self.extra_args or [os.getcwd()]
325: (8)                    options = AppOptions(
326: (12)                       app_dir=self.app_dir,
327: (12)                       logger=self.log,
328: (12)                       labextensions_path=self.labextensions_path,
329: (12)                       core_config=self.core_config,
330: (8)                    )
331: (8)                    return any(
332: (12)                       uninstall_extension(arg, all_=self.all, app_options=options) for
arg in self.extra_args
333: (8)                    )
334: (0)            class ListLabExtensionsApp(BaseExtensionApp):
335: (4)                description = "List the installed labextensions"
336: (4)                verbose = Bool(False, help="Increase verbosity level.").tag(config=True)
337: (4)                flags = list_flags
338: (4)                def run_task(self):
339: (8)                    list_extensions(
340: (12)                       app_options=AppOptions(
341: (16)                           app_dir=self.app_dir,
342: (16)                           logger=self.log,
343: (16)                           core_config=self.core_config,
344: (16)                           labextensions_path=self.labextensions_path,
345: (16)                           verbose=self.verbose,
346: (12)                       )
347: (8)                    )
348: (0)            class EnableLabExtensionsApp(BaseExtensionApp):
349: (4)                description = "Enable labextension(s) by name"
350: (4)                aliases = enable_aliases
351: (4)                level = Unicode("sys_prefix", help="Level at which to enable: sys_prefix,
user, system").tag(
352: (8)                    config=True
353: (4)                )
354: (4)                def run_task(self):
355: (8)                    app_options = AppOptions(
356: (12)                       app_dir=self.app_dir,
357: (12)                       logger=self.log,
358: (12)                       core_config=self.core_config,
359: (12)                       labextensions_path=self.labextensions_path,
360: (8)                    )
361: (8)                    [
362: (12)                       enable_extension(arg, app_options=app_options, level=self.level)
363: (12)                       for arg in self.extra_args
364: (8)                    ]
365: (0)            class DisableLabExtensionsApp(BaseExtensionApp):
366: (4)                description = "Disable labextension(s) by name"
367: (4)                aliases = disable_aliases
368: (4)                level = Unicode("sys_prefix", help="Level at which to disable: sys_prefix,
user, system").tag(
369: (8)                    config=True
370: (4)                )
371: (4)                def run_task(self):
372: (8)                    app_options = AppOptions(
373: (12)                       app_dir=self.app_dir,
374: (12)                       logger=self.log,
375: (12)                       core_config=self.core_config,
376: (12)                       labextensions_path=self.labextensions_path,
377: (8)                    )
378: (8)                    [
379: (12)                       disable_extension(arg, app_options=app_options, level=self.level)
```

```
380: (12)                       for arg in self.extra_args
381: (8)                    ]
382: (8)                    self.log.info(
383: (12)                       "Starting with JupyterLab 4.1 individual plugins can be re-enabled"
384: (12)                       " in the user interface. While all plugins which were previously"
385: (12)                       " disabled have been locked, you need to explicitly lock any newly"
386: (12)                       " disabled plugins by using `jupyter labextension lock` command."
387: (8)                    )
388: (0)            class LockLabExtensionsApp(BaseExtensionApp):
389: (4)                description = "Lock labextension(s) by name"
390: (4)                aliases = lock_aliases
391: (4)                level = Unicode("sys_prefix", help="Level at which to lock: sys_prefix, user, system").tag(
392: (8)                    config=True
393: (4)                )
394: (4)                def run_task(self):
395: (8)                    app_options = AppOptions(
396: (12)                       app_dir=self.app_dir,
397: (12)                       logger=self.log,
398: (12)                       core_config=self.core_config,
399: (12)                       labextensions_path=self.labextensions_path,
400: (8)                    )
401: (8)                    [lock_extension(arg, app_options=app_options, level=self.level) for arg in self.extra_args]
402: (0)            class UnlockLabExtensionsApp(BaseExtensionApp):
403: (4)                description = "Unlock labextension(s) by name"
404: (4)                aliases = unlock_aliases
405: (4)                level = Unicode("sys_prefix", help="Level at which to unlock: sys_prefix, user, system").tag(
406: (8)                    config=True
407: (4)                )
408: (4)                def run_task(self):
409: (8)                    app_options = AppOptions(
410: (12)                       app_dir=self.app_dir,
411: (12)                       logger=self.log,
412: (12)                       core_config=self.core_config,
413: (12)                       labextensions_path=self.labextensions_path,
414: (8)                    )
415: (8)                    [
416: (12)                       unlock_extension(arg, app_options=app_options, level=self.level)
417: (12)                       for arg in self.extra_args
418: (8)                    ]
419: (0)            class CheckLabExtensionsApp(BaseExtensionApp):
420: (4)                description = "Check labextension(s) by name"
421: (4)                flags = check_flags
422: (4)                should_check_installed_only = Bool(
423: (8)                    False,
424: (8)                    config=True,
425: (8)                    help="Whether it should check only if the extensions is installed",
426: (4)                )
427: (4)                def run_task(self):
428: (8)                    app_options = AppOptions(
429: (12)                       app_dir=self.app_dir,
430: (12)                       logger=self.log,
431: (12)                       core_config=self.core_config,
432: (12)                       labextensions_path=self.labextensions_path,
433: (8)                    )
434: (8)                    all_enabled = all(
435: (12)                       check_extension(
436: (16)                           arg, installed=self.should_check_installed_only, app_options=app_options
437: (12)                       )
438: (12)                       for arg in self.extra_args
439: (8)                    )
440: (8)                    if not all_enabled:
441: (12)                       self.exit(1)
442: (0)            _EXAMPLES = """
```

```
(0)           jupyter labextension list                        # list all configured labextensions
(0)           jupyter labextension install <extension name>    # install a labextension
(0)           jupyter labextension uninstall <extension name>  # uninstall a labextension
(0)           jupyter labextension develop                     # (developer) develop a prebuilt labextension
(0)           jupyter labextension build                       # (developer) build a prebuilt labextension
(0)           jupyter labextension watch                       # (developer) watch a prebuilt labextension
(0)           """
(0)           class LabExtensionApp(JupyterApp):
(4)               """Base jupyter labextension command entry point"""
(4)               name = "jupyter labextension"
(4)               version = VERSION
(4)               description = "Work with JupyterLab extensions"
(4)               examples = _EXAMPLES
(4)               subcommands = {
(8)                   "install": (InstallLabExtensionApp, "Install labextension(s)"),
(8)                   "update": (UpdateLabExtensionApp, "Update labextension(s)"),
(8)                   "uninstall": (UninstallLabExtensionApp, "Uninstall labextension(s)"),
(8)                   "list": (ListLabExtensionsApp, "List labextensions"),
(8)                   "link": (LinkLabExtensionApp, "Link labextension(s)"),
(8)                   "unlink": (UnlinkLabExtensionApp, "Unlink labextension(s)"),
(8)                   "enable": (EnableLabExtensionsApp, "Enable labextension(s)"),
(8)                   "disable": (DisableLabExtensionsApp, "Disable labextension(s)"),
(8)                   "lock": (LockLabExtensionsApp, "Lock labextension(s)"),
(8)                   "unlock": (UnlockLabExtensionsApp, "Unlock labextension(s)"),
(8)                   "check": (CheckLabExtensionsApp, "Check labextension(s)"),
(8)                   "develop": (DevelopLabExtensionApp, "(developer) Develop labextension(s)"),
(8)                   "build": (BuildLabExtensionApp, "(developer) Build labextension"),
(8)                   "watch": (WatchLabExtensionApp, "(developer) Watch labextension"),
(4)               }
(4)               def start(self):
(8)                   """Perform the App's functions as configured"""
(8)                   super().start()
(8)                   subcmds = ", ".join(sorted(self.subcommands))
(8)                   self.exit("Please supply at least one subcommand: %s" % subcmds)
(0)           main = LabExtensionApp.launch_instance
(0)           if __name__ == "__main__":
(4)               sys.exit(main())

----------------------------------------

File 9 - labhubapp.py:

(0)           """A JupyterHub EntryPoint that defaults to use JupyterLab"""
(0)           import os
(0)           from jupyter_server.serverapp import ServerApp
(0)           from traitlets import default
(0)           from .labapp import LabApp
(0)           if not os.environ.get("JUPYTERHUB_SINGLEUSER_APP"):
(4)               os.environ["JUPYTERHUB_SINGLEUSER_APP"] = "jupyter_server.serverapp.ServerApp"
(0)           try:
(4)               from jupyterhub.singleuser.mixins import make_singleuser_app
(0)           except ImportError:
(4)               from jupyterhub.singleuser import SingleUserNotebookApp as SingleUserServerApp
(0)           else:
(4)               SingleUserServerApp = make_singleuser_app(ServerApp)
(0)           class SingleUserLabApp(SingleUserServerApp):
(4)               @default("default_url")
(4)               def _default_url(self):
(8)                   return "/lab"
(4)               def find_server_extensions(self):
(8)                   """unconditionally enable jupyterlab server extension
(8)                   never called if using legacy SingleUserNotebookApp
```

```
21: (8)                     """
22: (8)                     super().find_server_extensions()
23: (8)                     self.jpserver_extensions[LabApp.get_extension_package()] = True
24: (0)             def main(argv=None):
25: (4)                 return SingleUserLabApp.launch_instance(argv)
26: (0)             if __name__ == "__main__":
27: (4)                 main()
```

File 10 - pytest_plugin.py:

```
1: (0)              import urllib.parse
2: (0)              import pytest
3: (0)              from jupyter_server.utils import url_path_join
4: (0)              from jupyterlab_server import LabConfig
5: (0)              from tornado.escape import url_escape
6: (0)              from traitlets import Unicode
7: (0)              from jupyterlab.labapp import LabApp
8: (0)              def mkdir(tmp_path, *parts):
9: (4)                  path = tmp_path.joinpath(*parts)
10: (4)                 if not path.exists():
11: (8)                     path.mkdir(parents=True)
12: (4)                 return path
13: (0)             app_settings_dir = pytest.fixture(lambda tmp_path: mkdir(tmp_path,
"app_settings"))
14: (0)             user_settings_dir = pytest.fixture(lambda tmp_path: mkdir(tmp_path,
"user_settings"))
15: (0)             schemas_dir = pytest.fixture(lambda tmp_path: mkdir(tmp_path, "schemas"))
16: (0)             workspaces_dir = pytest.fixture(lambda tmp_path: mkdir(tmp_path,
"workspaces"))
17: (0)             @pytest.fixture
18: (0)             def make_lab_app(
19: (4)                 jp_root_dir, jp_template_dir, app_settings_dir, user_settings_dir,
schemas_dir, workspaces_dir
20: (0)             ):
21: (4)                 def _make_lab_app(**kwargs):
22: (8)                     class TestLabApp(LabApp):
23: (12)                        base_url = "/lab"
24: (12)                        extension_url = "/lab"
25: (12)                        default_url = Unicode("/", help="The default URL to redirect to
from `/`")
26: (12)                        lab_config = LabConfig(
27: (16)                            app_name="JupyterLab Test App",
28: (16)                            static_dir=str(jp_root_dir),
29: (16)                            templates_dir=str(jp_template_dir),
30: (16)                            app_url="/lab",
31: (16)                            app_settings_dir=str(app_settings_dir),
32: (16)                            user_settings_dir=str(user_settings_dir),
33: (16)                            schemas_dir=str(schemas_dir),
34: (16)                            workspaces_dir=str(workspaces_dir),
35: (12)                        )
36: (8)                     app = TestLabApp()
37: (8)                     return app
38: (4)                 index = jp_template_dir.joinpath("index.html")
39: (4)                 index.write_text(
40: (8)                     """
41: (0)             <!DOCTYPE html>
42: (0)             <html>
43: (0)             <head>
44: (2)               <title>{{page_config['appName'] | e}}</title>
45: (0)             </head>
46: (0)             <body>
47: (4)                 {# Copy so we do not modify the page_config with updates. #}
48: (4)                 {% set page_config_full = page_config.copy() %}
49: (4)                 {# Set a dummy variable - we just want the side effect of the update. #}
50: (4)                 {% set _ = page_config_full.update(baseUrl=base_url, wsUrl=ws_url) %}
51: (6)                   <script id="jupyter-config-data" type="application/json">
52: (8)                     {{ page_config_full | tojson }}
53: (6)                     </script>
54: (2)                 <script src="{{page_config['fullStaticUrl'] | e}}/bundle.js" main="index">
</script>
55: (2)                 <script type="text/javascript">
56: (4)                   /* Remove token from URL. */
57: (4)                   (function () {
58: (6)                     var parsedUrl = new URL(window.location.href);
59: (6)                     if (parsedUrl.searchParams.get('token')) {
60: (8)                       parsedUrl.searchParams.delete('token');
61: (8)                       window.history.replaceState({ }, '', parsedUrl.href);
62: (6)                     }
63: (4)                   })();
64: (2)                 </script>
65: (0)               </body>
66: (0)               </html>
67: (0)               """
68: (4)                   )
69: (4)                   return _make_lab_app
70: (0)               @pytest.fixture
71: (0)               def labapp(jp_serverapp, make_lab_app):
72: (4)                   app = make_lab_app()
73: (4)                   app._link_jupyter_server_extension(jp_serverapp)
74: (4)                   app.initialize()
75: (4)                   return app
76: (0)               @pytest.fixture
77: (0)               def fetch_long(http_server_client, jp_auth_header, jp_base_url):
78: (4)                   """fetch fixture that handles auth, base_url, and path"""
79: (4)                   def client_fetch(*parts, headers=None, params=None, **kwargs):
80: (8)                       path_url = url_escape(url_path_join(*parts), plus=False)
81: (8)                       path_url = url_path_join(jp_base_url, path_url)
82: (8)                       params_url = urllib.parse.urlencode(params or {})
83: (8)                       url = path_url + "?" + params_url
84: (8)                       headers = headers or {}
85: (8)                       headers.update(jp_auth_header)
86: (8)                       return http_server_client.fetch(url, headers=headers,
request_timeout=250, **kwargs)
87: (4)                   return client_fetch

----------------------------------------

File 11 - semver.py:

1: (0)                import logging
2: (0)                import re
3: (0)                logger = logging.getLogger(__name__)
4: (0)                SEMVER_SPEC_VERSION = "2.0.0"
5: (0)                string_type = str
6: (0)                class _R:
7: (4)                    def __init__(self, i):
8: (8)                        self.i = i
9: (4)                    def __call__(self):
10: (8)                       v = self.i
11: (8)                       self.i += 1
12: (8)                       return v
13: (4)                   def value(self):
14: (8)                       return self.i
15: (0)               class Extendlist(list):
16: (4)                   def __setitem__(self, i, v):
17: (8)                       try:
18: (12)                          list.__setitem__(self, i, v)
19: (8)                       except IndexError:
20: (12)                          if len(self) == i:
21: (16)                              self.append(v)
22: (12)                          else:
23: (16)                              raise
24: (0)               def list_get(xs, i):
25: (4)                   try:
26: (8)                       return xs[i]
27: (4)                   except IndexError:
        return None
R = _R(0)
src = Extendlist()
regexp = {}
NUMERICIDENTIFIER = R()
src[NUMERICIDENTIFIER] = "0|[1-9]\\d*"
NUMERICIDENTIFIERLOOSE = R()
src[NUMERICIDENTIFIERLOOSE] = "[0-9]+"
NONNUMERICIDENTIFIER = R()
src[NONNUMERICIDENTIFIER] = "\\d*[a-zA-Z-][a-zA-Z0-9-]*"
MAINVERSION = R()
src[MAINVERSION] = (
    "("
    + src[NUMERICIDENTIFIER]
    + ")\\."
    + "("
    + src[NUMERICIDENTIFIER]
    + ")\\."
    + "("
    + src[NUMERICIDENTIFIER]
    + ")"
)
MAINVERSIONLOOSE = R()
src[MAINVERSIONLOOSE] = (
    "("
    + src[NUMERICIDENTIFIERLOOSE]
    + ")\\."
    + "("
    + src[NUMERICIDENTIFIERLOOSE]
    + ")\\."
    + "("
    + src[NUMERICIDENTIFIERLOOSE]
    + ")"
)
PRERELEASEIDENTIFIER = R()
src[PRERELEASEIDENTIFIER] = "(?:" + src[NUMERICIDENTIFIER] + "|" + src[NONNUMERICIDENTIFIER] + ")"
PRERELEASEIDENTIFIERLOOSE = R()
src[PRERELEASEIDENTIFIERLOOSE] = (
    "(?:" + src[NUMERICIDENTIFIERLOOSE] + "|" + src[NONNUMERICIDENTIFIER] + ")"
)
PRERELEASE = R()
src[PRERELEASE] = (
    "(?:-(" + src[PRERELEASEIDENTIFIER] + "(?:\\." + src[PRERELEASEIDENTIFIER] + ")*))"
)
PRERELEASELOOSE = R()
src[PRERELEASELOOSE] = (
    "(?:-?(" + src[PRERELEASEIDENTIFIERLOOSE] + "(?:\\." + src[PRERELEASEIDENTIFIERLOOSE] + ")*))"
)
BUILDIDENTIFIER = R()
src[BUILDIDENTIFIER] = "[0-9A-Za-z-]+"
BUILD = R()
src[BUILD] = "(?:\\+(" + src[BUILDIDENTIFIER] + "(?:\\." + src[BUILDIDENTIFIER] + ")*))"
FULL = R()
FULLPLAIN = "v?" + src[MAINVERSION] + src[PRERELEASE] + "?" + src[BUILD] + "?"
src[FULL] = "^" + FULLPLAIN + "$"
LOOSEPLAIN = "[v=\\s]*" + src[MAINVERSIONLOOSE] + src[PRERELEASELOOSE] + "?" + src[BUILD] + "?"
LOOSE = R()
src[LOOSE] = "^" + LOOSEPLAIN + "$"
GTLT = R()
src[GTLT] = "((?:<|>)?=?)"
XRANGEIDENTIFIERLOOSE = R()
src[XRANGEIDENTIFIERLOOSE] = src[NUMERICIDENTIFIERLOOSE] + "|x|X|\\*"
XRANGEIDENTIFIER = R()
```

```
(0)             src[XRANGEIDENTIFIER] = src[NUMERICIDENTIFIER] + "|x|X|\\*"
(0)             XRANGEPLAIN = R()
(0)             src[XRANGEPLAIN] = (
(4)                 "[v=\\s]*("
(4)                 + src[XRANGEIDENTIFIER]
(4)                 + ")"
(4)                 + "(?:\\.("
(4)                 + src[XRANGEIDENTIFIER]
(4)                 + ")"
(4)                 + "(?:\\.("
(4)                 + src[XRANGEIDENTIFIER]
(4)                 + ")"
(4)                 + "(?:"
(4)                 + src[PRERELEASE]
(4)                 + ")?"
(4)                 + src[BUILD]
(4)                 + "?"
(4)                 + ")?)?"
(0)             )
(0)             XRANGEPLAINLOOSE = R()
(0)             src[XRANGEPLAINLOOSE] = (
(4)                 "[v=\\s]*("
(4)                 + src[XRANGEIDENTIFIERLOOSE]
(4)                 + ")"
(4)                 + "(?:\\.("
(4)                 + src[XRANGEIDENTIFIERLOOSE]
(4)                 + ")"
(4)                 + "(?:\\.("
(4)                 + src[XRANGEIDENTIFIERLOOSE]
(4)                 + ")"
(4)                 + "(?:"
(4)                 + src[PRERELEASELOOSE]
(4)                 + ")?"
(4)                 + src[BUILD]
(4)                 + "?"
(4)                 + ")?)?"
(0)             )
(0)             XRANGE = R()
(0)             src[XRANGE] = "^" + src[GTLT] + "\\s*" + src[XRANGEPLAIN] + "$"
(0)             XRANGELOOSE = R()
(0)             src[XRANGELOOSE] = "^" + src[GTLT] + "\\s*" + src[XRANGEPLAINLOOSE] + "$"
(0)             LONETILDE = R()
(0)             src[LONETILDE] = "(?:~>?)"
(0)             TILDETRIM = R()
(0)             src[TILDETRIM] = "(\\s*)" + src[LONETILDE] + "\\s+"
(0)             regexp[TILDETRIM] = re.compile(src[TILDETRIM], re.M)
(0)             tildeTrimReplace = r"\1~"
(0)             TILDE = R()
(0)             src[TILDE] = "^" + src[LONETILDE] + src[XRANGEPLAIN] + "$"
(0)             TILDELOOSE = R()
(0)             src[TILDELOOSE] = "^" + src[LONETILDE] + src[XRANGEPLAINLOOSE] + "$"
(0)             LONECARET = R()
(0)             src[LONECARET] = "(?:\\^)"
(0)             CARETTRIM = R()
(0)             src[CARETTRIM] = "(\\s*)" + src[LONECARET] + "\\s+"
(0)             regexp[CARETTRIM] = re.compile(src[CARETTRIM], re.M)
(0)             caretTrimReplace = r"\1^"
(0)             CARET = R()
(0)             src[CARET] = "^" + src[LONECARET] + src[XRANGEPLAIN] + "$"
(0)             CARETLOOSE = R()
(0)             src[CARETLOOSE] = "^" + src[LONECARET] + src[XRANGEPLAINLOOSE] + "$"
(0)             COMPARATORLOOSE = R()
(0)             src[COMPARATORLOOSE] = "^" + src[GTLT] + "\\s*(" + LOOSEPLAIN + ")$|^$"
(0)             COMPARATOR = R()
(0)             src[COMPARATOR] = "^" + src[GTLT] + "\\s*(" + FULLPLAIN + ")$|^$"
(0)             COMPARATORTRIM = R()
(0)             src[COMPARATORTRIM] = "(\\s*)" + src[GTLT] + "\\s*(" + LOOSEPLAIN + "|" + src[XRANGEPLAIN] + ")"
(0)             regexp[COMPARATORTRIM] = re.compile(src[COMPARATORTRIM], re.M)
```

```
(0)         comparatorTrimReplace = r"\1\2\3"
(0)         HYPHENRANGE = R()
(0)         src[HYPHENRANGE] = (
(4)             "^\\s*(" + src[XRANGEPLAIN] + ")" + "\\s+-\\s+" + "(" + src[XRANGEPLAIN] + ")" + "\\s*$"
(0)         )
(0)         HYPHENRANGELOOSE = R()
(0)         src[HYPHENRANGELOOSE] = (
(4)             "^\\s*("
(4)             + src[XRANGEPLAINLOOSE]
(4)             + ")"
(4)             + "\\s+-\\s+"
(4)             + "("
(4)             + src[XRANGEPLAINLOOSE]
(4)             + ")"
(4)             + "\\s*$"
(0)         )
(0)         STAR = R()
(0)         src[STAR] = "(<|>)?=?\\s*\\*"
(0)         RECOVERYVERSIONNAME = R()
(0)         _n = src[NUMERICIDENTIFIER]
(0)         _pre = src[PRERELEASELOOSE]
(0)         src[RECOVERYVERSIONNAME] = f"v?({_n})(?:\\.({_n}))?{_pre}?"
(0)         for i in range(R.value()):
(4)             logger.debug("genregxp %s %s", i, src[i])
(4)             if i not in regexp:
(8)                 regexp[i] = re.compile(src[i])
(0)         def parse(version, loose):
(4)             r = regexp[LOOSE] if loose else regexp[FULL]
(4)             m = r.search(version)
(4)             if m:
(8)                 return semver(version, loose)
(4)             else:
(8)                 return None
(0)         def valid(version, loose):
(4)             v = parse(version, loose)
(4)             if v.version:
(8)                 return v
(4)             else:
(8)                 return None
(0)         def clean(version, loose):
(4)             s = parse(version, loose)
(4)             if s:
(8)                 return s.version
(4)             else:
(8)                 return None
(0)         NUMERIC = re.compile(r"^\d+$")
(0)         def semver(version, loose):
(4)             if isinstance(version, SemVer):
(8)                 if version.loose == loose:
(12)                    return version
(8)                 else:
(12)                    version = version.version
(4)             elif not isinstance(version, string_type):  # xxx:
(8)                 raise ValueError(f"Invalid Version: {version}")
(4)             """
(4)             if (!(this instanceof SemVer))
(7)                return new SemVer(version, loose);
(4)             """
(4)             return SemVer(version, loose)
(0)         make_semver = semver
(0)         class SemVer:
(4)             def __init__(self, version, loose):
(8)                 logger.debug("SemVer %s, %s", version, loose)
(8)                 self.loose = loose
(8)                 self.raw = version
(8)                 m = regexp[LOOSE if loose else FULL].search(version.strip())
(8)                 if not m:
(12)                    if not loose:
```

```
227: (16)                           raise ValueError(f"Invalid Version: {version}")
228: (12)                       m = regexp[RECOVERYVERSIONNAME].search(version.strip())
229: (12)                       self.major = int(m.group(1)) if m.group(1) else 0
230: (12)                       self.minor = int(m.group(2)) if m.group(2) else 0
231: (12)                       self.patch = 0
232: (12)                       if not m.group(3):
233: (16)                           self.prerelease = []
234: (12)                       else:
235: (16)                           self.prerelease = [
236: (20)                               (int(id_) if NUMERIC.search(id_) else id_) for id_ in
m.group(3).split(".")
237: (16)                           ]
238: (8)                    else:
239: (12)                       self.major = int(m.group(1))
240: (12)                       self.minor = int(m.group(2))
241: (12)                       self.patch = int(m.group(3))
242: (12)                       if not m.group(4):
243: (16)                           self.prerelease = []
244: (12)                       else:
245: (16)                           self.prerelease = [
246: (20)                               (int(id_) if NUMERIC.search(id_) else id_) for id_ in
m.group(4).split(".")
247: (16)                           ]
248: (12)                       if m.group(5):
249: (16)                           self.build = m.group(5).split(".")
250: (12)                       else:
251: (16)                           self.build = []
252: (8)                    self.format()  # xxx:
253: (4)                def format(self):
254: (8)                    self.version = f"{self.major}.{self.minor}.{self.patch}"
255: (8)                    if len(self.prerelease) > 0:
256: (12)                       self.version += "-{}".format(".".join(str(v) for v in
self.prerelease))
257: (8)                    return self.version
258: (4)                def __repr__(self):
259: (8)                    return f"<SemVer {self} >"
260: (4)                def __str__(self):
261: (8)                    return self.version
262: (4)                def compare(self, other):
263: (8)                    logger.debug("SemVer.compare %s %s %s", self.version, self.loose,
other)
264: (8)                    if not isinstance(other, SemVer):
265: (12)                       other = make_semver(other, self.loose)
266: (8)                    result = self.compare_main(other) or self.compare_pre(other)
267: (8)                    logger.debug("compare result %s", result)
268: (8)                    return result
269: (4)                def compare_main(self, other):
270: (8)                    if not isinstance(other, SemVer):
271: (12)                       other = make_semver(other, self.loose)
272: (8)                    return (
273: (12)                       compare_identifiers(str(self.major), str(other.major))
274: (12)                       or compare_identifiers(str(self.minor), str(other.minor))
275: (12)                       or compare_identifiers(str(self.patch), str(other.patch))
276: (8)                    )
277: (4)                def compare_pre(self, other):  # noqa PLR0911
278: (8)                    if not isinstance(other, SemVer):
279: (12)                       other = make_semver(other, self.loose)
280: (8)                    is_self_more_than_zero = len(self.prerelease) > 0
281: (8)                    is_other_more_than_zero = len(other.prerelease) > 0
282: (8)                    if not is_self_more_than_zero and is_other_more_than_zero:
283: (12)                       return 1
284: (8)                    elif is_self_more_than_zero and not is_other_more_than_zero:
285: (12)                       return -1
286: (8)                    elif not is_self_more_than_zero and not is_other_more_than_zero:
287: (12)                       return 0
288: (8)                    i = 0
289: (8)                    while True:
290: (12)                       a = list_get(self.prerelease, i)
291: (12)                       b = list_get(other.prerelease, i)
```

```
(12)             logger.debug("prerelease compare %s: %s %s", i, a, b)
(12)             i += 1
(12)             if a is None and b is None:
(16)                 return 0
(12)             elif b is None:
(16)                 return 1
(12)             elif a is None:
(16)                 return -1
(12)             elif a == b:
(16)                 continue
(12)             else:
(16)                 return compare_identifiers(str(a), str(b))
(4)     def inc(self, release, identifier=None):  # noqa PLR0915
(8)         logger.debug("inc release %s %s", self.prerelease, release)
(8)         if release == "premajor":
(12)             self.prerelease = []
(12)             self.patch = 0
(12)             self.minor = 0
(12)             self.major += 1
(12)             self.inc("pre", identifier=identifier)
(8)         elif release == "preminor":
(12)             self.prerelease = []
(12)             self.patch = 0
(12)             self.minor += 1
(12)             self.inc("pre", identifier=identifier)
(8)         elif release == "prepatch":
(12)             self.prerelease = []
(12)             self.inc("patch", identifier=identifier)
(12)             self.inc("pre", identifier=identifier)
(8)         elif release == "prerelease":
(12)             if len(self.prerelease) == 0:
(16)                 self.inc("patch", identifier=identifier)
(12)             self.inc("pre", identifier=identifier)
(8)         elif release == "major":
(12)             if self.minor != 0 or self.patch != 0 or len(self.prerelease) == 0:
(16)                 self.major += 1
(12)             self.minor = 0
(12)             self.patch = 0
(12)             self.prerelease = []
(8)         elif release == "minor":
(12)             if self.patch != 0 or len(self.prerelease) == 0:
(16)                 self.minor += 1
(12)             self.patch = 0
(12)             self.prerelease = []
(8)         elif release == "patch":
(12)             if len(self.prerelease) == 0:
(16)                 self.patch += 1
(12)             self.prerelease = []
(8)         elif release == "pre":
(12)             logger.debug("inc prerelease %s", self.prerelease)
(12)             if len(self.prerelease) == 0:
(16)                 self.prerelease = [0]
(12)             else:
(16)                 i = len(self.prerelease) - 1
(16)                 while i >= 0:
(20)                     if isinstance(self.prerelease[i], int):
(24)                         self.prerelease[i] += 1
(24)                         i -= 2
(20)                     i -= 1
(12)             if identifier is not None:
(16)                 if self.prerelease[0] == identifier:
(20)                     if not isinstance(self.prerelease[1], int):
(24)                         self.prerelease = [identifier, 0]
(16)                 else:
(20)                     self.prerelease = [identifier, 0]
(8)         else:
(12)             raise ValueError(f"invalid increment argument: {release}")
(8)         self.format()
```

```
(8)                     self.raw = self.version
(8)                     return self
(0)             def inc(version, release, loose, identifier=None):  # wow!
(4)                 try:
(8)                     return make_semver(version, loose).inc(release, identifier=identifier).version
(4)                 except Exception as e:
(8)                     logger.debug(e, exc_info=5)
(8)                     return None
(0)             def compare_identifiers(a, b):
(4)                 anum = NUMERIC.search(a)
(4)                 bnum = NUMERIC.search(b)
(4)                 if anum and bnum:
(8)                     a = int(a)
(8)                     b = int(b)
(4)                 if anum and not bnum:
(8)                     return -1
(4)                 elif bnum and not anum:
(8)                     return 1
(4)                 elif a < b:
(8)                     return -1
(4)                 elif a > b:
(8)                     return 1
(4)                 else:
(8)                     return 0
(0)             def rcompare_identifiers(a, b):
(4)                 return compare_identifiers(b, a)
(0)             def compare(a, b, loose):
(4)                 return make_semver(a, loose).compare(b)
(0)             def compare_loose(a, b):
(4)                 return compare(a, b, True)
(0)             def rcompare(a, b, loose):
(4)                 return compare(b, a, loose)
(0)             def make_key_function(loose):
(4)                 def key_function(version):
(8)                     v = make_semver(version, loose)
(8)                     key = (v.major, v.minor, v.patch)
(8)                     if v.prerelease:  # noqa SIM108
(12)                        key = key + tuple(v.prerelease)
(8)                     else:
(12)                        key = (*key, float("inf"))
(8)                     return key
(4)                 return key_function
(0)             loose_key_function = make_key_function(True)
(0)             full_key_function = make_key_function(True)
(0)             def sort(list_, loose):
(4)                 keyf = loose_key_function if loose else full_key_function
(4)                 list_.sort(key=keyf)
(4)                 return list_
(0)             def rsort(list_, loose):
(4)                 keyf = loose_key_function if loose else full_key_function
(4)                 list_.sort(key=keyf, reverse=True)
(4)                 return list_
(0)             def gt(a, b, loose):
(4)                 return compare(a, b, loose) > 0
(0)             def lt(a, b, loose):
(4)                 return compare(a, b, loose) < 0
(0)             def eq(a, b, loose):
(4)                 return compare(a, b, loose) == 0
(0)             def neq(a, b, loose):
(4)                 return compare(a, b, loose) != 0
(0)             def gte(a, b, loose):
(4)                 return compare(a, b, loose) >= 0
(0)             def lte(a, b, loose):
(4)                 return compare(a, b, loose) <= 0
(0)             def cmp(a, op, b, loose):  # noqa PLR0911
(4)                 logger.debug("cmp: %s", op)
(4)                 if op == "===":
(8)                     return a == b
```

```
(4)                 elif op == "!==":
(8)                     return a != b
(4)                 elif op == "" or op == "=" or op == "==":
(8)                     return eq(a, b, loose)
(4)                 elif op == "!=":
(8)                     return neq(a, b, loose)
(4)                 elif op == ">":
(8)                     return gt(a, b, loose)
(4)                 elif op == ">=":
(8)                     return gte(a, b, loose)
(4)                 elif op == "<":
(8)                     return lt(a, b, loose)
(4)                 elif op == "<=":
(8)                     return lte(a, b, loose)
(4)                 else:
(8)                     raise ValueError(f"Invalid operator: {op}")
(0)             def comparator(comp, loose):
(4)                 if isinstance(comp, Comparator):
(8)                     if comp.loose == loose:
(12)                        return comp
(8)                     else:
(12)                        comp = comp.value
(4)                 return Comparator(comp, loose)
(0)             make_comparator = comparator
(0)             ANY = object()
(0)             class Comparator:
(4)                 semver = None
(4)                 def __init__(self, comp, loose):
(8)                     logger.debug("comparator: %s %s", comp, loose)
(8)                     self.loose = loose
(8)                     self.parse(comp)
(8)                     if self.semver == ANY:
(12)                        self.value = ""
(8)                     else:
(12)                        self.value = self.operator + self.semver.version
(4)                 def parse(self, comp):
(8)                     r = regexp[COMPARATORLOOSE] if self.loose else regexp[COMPARATOR]
(8)                     logger.debug("parse comp=%s", comp)
(8)                     m = r.search(comp)
(8)                     if m is None:
(12)                        raise ValueError(f"Invalid comparator: {comp}")
(8)                     self.operator = m.group(1)
(8)                     if m.group(2) is None:
(12)                        self.semver = ANY
(8)                     else:
(12)                        self.semver = semver(m.group(2), self.loose)
(4)                 def __repr__(self):
(8)                     return f'<SemVer Comparator "{self}">'
(4)                 def __str__(self):
(8)                     return self.value
(4)                 def test(self, version):
(8)                     logger.debug("Comparator, test %s, %s", version, self.loose)
(8)                     if self.semver == ANY:
(12)                        return True
(8)                     else:
(12)                        return cmp(version, self.operator, self.semver, self.loose)
(0)             def make_range(range_, loose):
(4)                 if isinstance(range_, Range) and range_.loose == loose:
(8)                     return range_
(4)                 return Range(range_, loose)
(0)             class Range:
(4)                 def __init__(self, range_, loose):
(8)                     self.loose = loose
(8)                     self.raw = range_
(8)                     xs = [self.parse_range(r.strip()) for r in re.split(r"\s*\|\|\s*", range_)]
(8)                     self.set = [r for r in xs if r]
(8)                     if not len(self.set):
(12)                        raise ValueError(f"Invalid SemVer Range: {range_}")
```

```
(8)             self.format()
(4)         def __repr__(self):
(8)             return f'<SemVer Range "{self.range}">'
(4)         def format(self):
(8)             self.range = "||".join(
(12)                [" ".join(c.value for c in comps).strip() for comps in self.set]
(8)             ).strip()
(8)             logger.debug("Range format %s", self.range)
(8)             return self.range
(4)         def __str__(self):
(8)             return self.range
(4)         def parse_range(self, range_):
(8)             loose = self.loose
(8)             logger.debug("range %s %s", range_, loose)
(8)             hr = regexp[HYPHENRANGELOOSE] if loose else regexp[HYPHENRANGE]
(8)             range_ = hr.sub(
(12)                hyphen_replace,
(12)                range_,
(8)             )
(8)             logger.debug("hyphen replace %s", range_)
(8)             range_ = regexp[COMPARATORTRIM].sub(comparatorTrimReplace, range_)
(8)             logger.debug("comparator trim %s, %s", range_, regexp[COMPARATORTRIM])
(8)             range_ = regexp[TILDETRIM].sub(tildeTrimReplace, range_)
(8)             range_ = regexp[CARETTRIM].sub(caretTrimReplace, range_)
(8)             range_ = " ".join(re.split(r"\s+", range_))
(8)             comp_re = regexp[COMPARATORLOOSE] if loose else regexp[COMPARATOR]
(8)             set_ = re.split(
(12)                r"\s+", " ".join([parse_comparator(comp, loose) for comp in range_.split(" ")])
(8)             )
(8)             if self.loose:
(12)                set_ = [comp for comp in set_ if comp_re.search(comp)]
(8)             set_ = [make_comparator(comp, loose) for comp in set_]
(8)             return set_
(4)         def test(self, version):
(8)             if not version:  # xxx
(12)                return False
(8)             if isinstance(version, string_type):
(12)                version = make_semver(version, loose=self.loose)
(8)             return any(test_set(e, version) for e in self.set)
(0)     def to_comparators(range_, loose):
(4)         return [
(8)             " ".join([c.value for c in comp]).strip().split(" ")
(8)             for comp in make_range(range_, loose).set
(4)         ]
(0)     def parse_comparator(comp, loose):
(4)         logger.debug("comp %s", comp)
(4)         comp = replace_carets(comp, loose)
(4)         logger.debug("caret %s", comp)
(4)         comp = replace_tildes(comp, loose)
(4)         logger.debug("tildes %s", comp)
(4)         comp = replace_xranges(comp, loose)
(4)         logger.debug("xrange %s", comp)
(4)         comp = replace_stars(comp, loose)
(4)         logger.debug("stars %s", comp)
(4)         return comp
(0)     def is_x(id_):
(4)         return id_ is None or id_ == "" or id_.lower() == "x" or id_ == "*"
(0)     def replace_tildes(comp, loose):
(4)         return " ".join([replace_tilde(c, loose) for c in re.split(r"\s+", comp.strip())])
(0)     def replace_tilde(comp, loose):
(4)         r = regexp[TILDELOOSE] if loose else regexp[TILDE]
(4)         def repl(mob):
(8)             _ = mob.group(0)
(8)             M, m, p, pr, _ = mob.groups()
(8)             logger.debug("tilde %s %s %s %s %s %s", comp, _, M, m, p, pr)
(8)             if is_x(M):
(12)                ret = ""
```

```
(8)             elif is_x(m):
(12)                ret = ">=" + M + ".0.0 <" + str(int(M) + 1) + ".0.0"
(8)             elif is_x(p):
(12)                ret = ">=" + M + "." + m + ".0 <" + M + "." + str(int(m) + 1) + ".0"
(8)             elif pr:
(12)                logger.debug("replaceTilde pr %s", pr)
(12)                if pr[0] != "-":
(16)                    pr = "-" + pr
(12)                ret = ">=" + M + "." + m + "." + p + pr + " <" + M + "." + str(int(m) + 1) + ".0"
(8)             else:
(12)                ret = ">=" + M + "." + m + "." + p + " <" + M + "." + str(int(m) + 1) + ".0"
(8)             logger.debug("tilde return, %s", ret)
(8)             return ret
(4)         return r.sub(repl, comp)
(0)     def replace_carets(comp, loose):
(4)         return " ".join([replace_caret(c, loose) for c in re.split(r"\s+", comp.strip())])
(0)     def replace_caret(comp, loose):
(4)         r = regexp[CARETLOOSE] if loose else regexp[CARET]
(4)         def repl(mob):  # noqa PLR0911
(8)             m0 = mob.group(0)
(8)             M, m, p, pr, _ = mob.groups()
(8)             logger.debug("caret %s %s %s %s %s %s", comp, m0, M, m, p, pr)
(8)             if is_x(M):
(12)                ret = ""
(8)             elif is_x(m):
(12)                ret = ">=" + M + ".0.0 <" + str(int(M) + 1) + ".0.0"
(8)             elif is_x(p):
(12)                if M == "0":
(16)                    ret = ">=" + M + "." + m + ".0 <" + M + "." + str(int(m) + 1) + ".0"
(12)                else:
(16)                    ret = ">=" + M + "." + m + ".0 <" + str(int(M) + 1) + ".0.0"
(8)             elif pr:
(12)                logger.debug("replaceCaret pr %s", pr)
(12)                if pr[0] != "-":
(16)                    pr = "-" + pr
(12)                if M == "0":
(16)                    if m == "0":
(20)                        ret = (
(24)                            ">="
(24)                            + M
(24)                            + "."
(24)                            + m
(24)                            + "."
(24)                            + (p or "")
(24)                            + pr
(24)                            + " <"
(24)                            + M
(24)                            + "."
(24)                            + m
(24)                            + "."
(24)                            + str(int(p or 0) + 1)
(20)                        )
(16)                    else:
(20)                        ret = (
(24)                            ">="
(24)                            + M
(24)                            + "."
(24)                            + m
(24)                            + "."
(24)                            + (p or "")
(24)                            + pr
(24)                            + " <"
(24)                            + M
(24)                            + "."
(24)                        + str(int(m) + 1)
(24)                        + ".0"
(20)                    )
(12)            else:
(16)                ret = ">=" + M + "." + m + "." + (p or "") + pr + " <" + str(int(M) + 1) + ".0.0"
(8)         else:
(12)            if M == "0":
(16)                if m == "0":
(20)                    ret = (
(24)                        ">="
(24)                        + M
(24)                        + "."
(24)                        + m
(24)                        + "."
(24)                        + (p or "")
(24)                        + " <"
(24)                        + M
(24)                        + "."
(24)                        + m
(24)                        + "."
(24)                        + str(int(p or 0) + 1)
(20)                    )
(16)                else:
(20)                    ret = (
(24)                        ">="
(24)                        + M
(24)                        + "."
(24)                        + m
(24)                        + "."
(24)                        + (p or "")
(24)                        + " <"
(24)                        + M
(24)                        + "."
(24)                        + str(int(m) + 1)
(24)                        + ".0"
(20)                    )
(12)            else:
(16)                ret = ">=" + M + "." + m + "." + (p or "") + " <" + str(int(M) + 1) + ".0.0"
(8)         logger.debug("caret return %s", ret)
(8)         return ret
(4)     return r.sub(repl, comp)
(0) def replace_xranges(comp, loose):
(4)     logger.debug("replaceXRanges %s %s", comp, loose)
(4)     return " ".join([replace_xrange(c, loose) for c in re.split(r"\s+", comp.strip())])
(0) def replace_xrange(comp, loose):
(4)     comp = comp.strip()
(4)     r = regexp[XRANGELOOSE] if loose else regexp[XRANGE]
(4)     def repl(mob):  # noqa PLR0911
(8)         ret = mob.group(0)
(8)         gtlt, M, m, p, pr, _ = mob.groups()
(8)         logger.debug("xrange %s %s %s %s %s %s %s", comp, ret, gtlt, M, m, p, pr)
(8)         xM = is_x(M)
(8)         xm = xM or is_x(m)
(8)         xp = xm or is_x(p)
(8)         any_x = xp
(8)         if gtlt == "=" and any_x:
(12)            gtlt = ""
(8)         logger.debug("xrange gtlt=%s any_x=%s", gtlt, any_x)
(8)         if xM:
(12)            if gtlt == ">" or gtlt == "<":  # noqa SIM108
(16)                ret = "<0.0.0"
(12)            else:
(16)                ret = "*"
(8)         elif gtlt and any_x:
(12)            if xm:
```

```
(16)                    m = 0
(12)                if xp:
(16)                    p = 0
(12)                if gtlt == ">":
(16)                    gtlt = ">="
(16)                    if xm:
(20)                        M = int(M) + 1
(20)                        m = 0
(20)                        p = 0
(16)                    elif xp:
(20)                        m = int(m) + 1
(20)                        p = 0
(12)                elif gtlt == "<=":
(16)                    gtlt = "<"
(16)                    if xm:
(20)                        M = int(M) + 1
(16)                    else:
(20)                        m = int(m) + 1
(12)                ret = gtlt + str(M) + "." + str(m) + "." + str(p)
(8)             elif xm:
(12)                ret = ">=" + M + ".0.0 <" + str(int(M) + 1) + ".0.0"
(8)             elif xp:
(12)                ret = ">=" + M + "." + m + ".0 <" + M + "." + str(int(m) + 1) +
".0"
(8)             logger.debug("xRange return %s", ret)
(8)             return ret
(4)         return r.sub(repl, comp)
(0)     def replace_stars(comp, loose):
(4)         logger.debug("replaceStars %s %s", comp, loose)
(4)         return regexp[STAR].sub("", comp.strip())
(0)     def hyphen_replace(mob):
(4)         from_, fM, fm, fp, fpr, fb, to, tM, tm, tp, tpr, tb = mob.groups()
(4)         if is_x(fM):
(8)             from_ = ""
(4)         elif is_x(fm):
(8)             from_ = ">=" + fM + ".0.0"
(4)         elif is_x(fp):
(8)             from_ = ">=" + fM + "." + fm + ".0"
(4)         else:
(8)             from_ = ">=" + from_
(4)         if is_x(tM):
(8)             to = ""
(4)         elif is_x(tm):
(8)             to = "<" + str(int(tM) + 1) + ".0.0"
(4)         elif is_x(tp):
(8)             to = "<" + tM + "." + str(int(tm) + 1) + ".0"
(4)         elif tpr:
(8)             to = "<=" + tM + "." + tm + "." + tp + "-" + tpr
(4)         else:
(8)             to = "<=" + to
(4)         return (from_ + " " + to).strip()
(0)     def test_set(set_, version):
(4)         for e in set_:
(8)             if not e.test(version):
(12)                return False
(4)         if len(version.prerelease) > 0:
(8)             for e in set_:
(12)                if e.semver == ANY:
(16)                    continue
(12)                if len(e.semver.prerelease) > 0:
(16)                    allowed = e.semver
(16)                    if (
(20)                        allowed.major == version.major
(20)                        and allowed.minor == version.minor
(20)                        and allowed.patch == version.patch
(16)                    ):
(20)                        return True
(8)             return False
(4)         return True
```

```
(0)         def satisfies(version, range_, loose=False):
(4)             try:
(8)                 range_ = make_range(range_, loose)
(4)             except Exception:
(8)                 return False
(4)             return range_.test(version)
(0)         def max_satisfying(versions, range_, loose=False):
(4)             try:
(8)                 range_ob = make_range(range_, loose=loose)
(4)             except Exception:
(8)                 return None
(4)             max_ = None
(4)             max_sv = None
(4)             for v in versions:
(8)                 if range_ob.test(v):  # noqa  # satisfies(v, range_, loose=loose)
(12)                    if max_ is None or max_sv.compare(v) == -1:  # compare(max, v, true)
(16)                        max_ = v
(16)                        max_sv = make_semver(max_, loose=loose)
(4)             return max_
(0)         def valid_range(range_, loose):
(4)             try:
(8)                 return make_range(range_, loose).range or "*"
(4)             except Exception:
(8)                 return None
(0)         def ltr(version, range_, loose):
(4)             return outside(version, range_, "<", loose)
(0)         def rtr(version, range_, loose):
(4)             return outside(version, range_, ">", loose)
(0)         def outside(version, range_, hilo, loose):
(4)             version = make_semver(version, loose)
(4)             range_ = make_range(range_, loose)
(4)             if hilo == ">":
(8)                 gtfn = gt
(8)                 ltefn = lte
(8)                 ltfn = lt
(8)                 comp = ">"
(8)                 ecomp = ">="
(4)             elif hilo == "<":
(8)                 gtfn = lt
(8)                 ltefn = gte
(8)                 ltfn = gt
(8)                 comp = "<"
(8)                 ecomp = "<="
(4)             else:
(8)                 raise ValueError("Must provide a hilo val of '<' or '>'")
(4)             if satisfies(version, range_, loose):
(8)                 return False
(4)             for comparators in range_.set:
(8)                 high = None
(8)                 low = None
(8)                 for comparator in comparators:
(12)                    high = high or comparator
(12)                    low = low or comparator
(12)                    if gtfn(comparator.semver, high.semver, loose):
(16)                        high = comparator
(12)                    elif ltfn(comparator.semver, low.semver, loose):
(16)                        low = comparator
(4)             if high.operator == comp or high.operator == ecomp:
(8)                 return False
(4)             if (not low.operator or low.operator == comp) and ltefn(version, low.semver):  # noqa SIM114
(8)                 return False
(4)             elif low.operator == ecomp and ltfn(version, low.semver):
(8)                 return False
(4)             return True
```

File 12 - serverextension.py:

```
from jupyter_server.utils import url_path_join
from tornado.web import RedirectHandler
def load_jupyter_server_extension(serverapp):
    from .labapp import LabApp
    """Temporary server extension shim when using
    old notebook server.
    """
    extension = LabApp()
    extension.serverapp = serverapp
    extension.load_config_file()
    extension.update_config(serverapp.config)
    extension.parse_command_line(serverapp.extra_args)
    extension.handlers.extend(
        [
            (
                r"/static/favicons/favicon.ico",
                RedirectHandler,
                {"url": url_path_join(serverapp.base_url, "static/base/images/favicon.ico")},
            ),
            (
                r"/static/favicons/favicon-busy-1.ico",
                RedirectHandler,
                {"url": url_path_join(serverapp.base_url, "static/base/images/favicon-busy-1.ico")},
            ),
            (
                r"/static/favicons/favicon-busy-2.ico",
                RedirectHandler,
                {"url": url_path_join(serverapp.base_url, "static/base/images/favicon-busy-2.ico")},
            ),
            (
                r"/static/favicons/favicon-busy-3.ico",
                RedirectHandler,
                {"url": url_path_join(serverapp.base_url, "static/base/images/favicon-busy-3.ico")},
            ),
            (
                r"/static/favicons/favicon-file.ico",
                RedirectHandler,
                {"url": url_path_join(serverapp.base_url, "static/base/images/favicon-file.ico")},
            ),
            (
                r"/static/favicons/favicon-notebook.ico",
                RedirectHandler,
                {
                    "url": url_path_join(
                        serverapp.base_url, "static/base/images/favicon-notebook.ico"
                    )
                },
            ),
            (
                r"/static/favicons/favicon-terminal.ico",
                RedirectHandler,
                {
                    "url": url_path_join(
                        serverapp.base_url, "static/base/images/favicon-terminal.ico"
                    )
                },
            ),
            (
                r"/static/logo/logo.png",
                RedirectHandler,
61: (16)                            {"url": url_path_join(serverapp.base_url,
"static/base/images/logo.png")},
62: (12)                        ),
63: (8)                     ]
64: (4)                 )
65: (4)                 extension.initialize()

----------------------------------------

File 13 - upgrade_extension.py:

1: (0)              import configparser
2: (0)              import json
3: (0)              import re
4: (0)              import shutil
5: (0)              import subprocess
6: (0)              import sys
7: (0)              from typing import Optional
8: (0)              try:
9: (4)                  import tomllib
10: (0)             except ImportError:
11: (4)                 import tomli as tomllib
12: (0)             try:
13: (4)                 from importlib.resources import files
14: (0)             except ImportError:
15: (4)                 from importlib_resources import files
16: (0)             from pathlib import Path
17: (0)             try:
18: (4)                 import copier
19: (0)             except ModuleNotFoundError:
20: (4)                 msg = "Please install copier; you can use `pip install jupyterlab[upgrade-
extension]`"
21: (4)                 raise RuntimeError(msg) from None
22: (0)             RECOMMENDED_TO_OVERRIDE = [
23: (4)                 ".github/workflows/binder-on-pr.yml",
24: (4)                 ".github/workflows/build.yml",
25: (4)                 ".github/workflows/check-release.yml",
26: (4)                 ".github/workflows/enforce-label.yml",
27: (4)                 ".github/workflows/prep-release.yml",
28: (4)                 ".github/workflows/publish-release.yml",
29: (4)                 ".github/workflows/update-integration-tests.yml",
30: (4)                 "binder/postBuild",
31: (4)                 ".eslintignore",
32: (4)                 ".eslintrc.js",
33: (4)                 ".gitignore",
34: (4)                 ".prettierignore",
35: (4)                 ".prettierrc",
36: (4)                 ".stylelintrc",
37: (4)                 "RELEASE.md",
38: (4)                 "babel.config.js",
39: (4)                 "conftest.py",
40: (4)                 "jest.config.js",
41: (4)                 "pyproject.toml",
42: (4)                 "setup.py",
43: (4)                 "tsconfig.json",
44: (4)                 "tsconfig.test.json",
45: (4)                 "ui-tests/README.md",
46: (4)                 "ui-tests/jupyter_server_test_config.py",
47: (4)                 "ui-tests/package.json",
48: (4)                 "ui-tests/playwright.config.js",
49: (0)             ]
50: (0)             JUPYTER_SERVER_REQUIREMENT = re.compile("^jupyter_server([^\\w]|$)")
51: (0)             def update_extension(  # noqa
52: (4)                 target: str, vcs_ref: Optional[str] = None, interactive: bool = True
53: (0)             ) -> None:
54: (4)                 """Update an extension to the current JupyterLab
55: (4)                 target: str
56: (8)                     Path to the extension directory containing the extension
57: (4)                 vcs_ref: str [default: None]
```

```
        Template vcs_ref to checkout
    interactive: bool [default: true]
        Whether to ask before overwriting content
    """
    target = Path(target).resolve()
    package_file = target / "package.json"
    pyproject_file = target / "pyproject.toml"
    setup_file = target / "setup.py"
    if not package_file.exists():
        msg = f"No package.json exists in {target!s}"
        raise RuntimeError(msg)
    with open(package_file) as fid:
        data = json.load(fid)
    python_name = None
    if pyproject_file.exists():
        pyproject = tomllib.loads(pyproject_file.read_text())
        python_name = pyproject.get("project", {}).get("name")
    if python_name is None:
        if setup_file.exists():
            python_name = (
                subprocess.check_output(
                    [sys.executable, "setup.py", "--name"],  # noqa: S603
                    cwd=target,
                )
                .decode("utf8")
                .strip()
            )
        else:
            python_name = data["name"]
            if "@" in python_name:
                python_name = python_name[1:]
        python_name = python_name.replace("/", "_").replace("-", "_")
    output_dir = target / "_temp_extension"
    if output_dir.exists():
        shutil.rmtree(output_dir)
    author = data.get("author", "<author_name>")
    author_email = ""
    if isinstance(author, dict):
        author_name = author.get("name", "<author_name>")
        author_email = author.get("email", author_email)
    else:
        author_name = author
    kind = "frontend"
    if (target / "jupyter-config").exists():
        kind = "server"
    elif data.get("jupyterlab", {}).get("themePath", ""):
        kind = "theme"
    has_test = (
        (target / "conftest.py").exists()
        or (target / "jest.config.js").exists()
        or (target / "ui-tests").exists()
    )
    extra_context = {
        "kind": kind,
        "author_name": author_name,
        "author_email": author_email,
        "labextension_name": data["name"],
        "python_name": python_name,
        "project_short_description": data.get("description", "<description>"),
        "has_settings": bool(data.get("jupyterlab", {}).get("schemaDir", "")),
        "has_binder": bool((target / "binder").exists()),
        "test": bool(has_test),
        "repository": data.get("repository", {}).get("url", "<repository"),
    }
    template = "https://github.com/jupyterlab/extension-template"
    if tuple(copier.__version__.split(".")) < ("8", "0", "0"):
        copier.run_auto(template, output_dir, vcs_ref=vcs_ref, data=extra_context, defaults=True)
    else:
```

```
126: (8)                    copier.run_copy(
127: (12)                       template, output_dir, vcs_ref=vcs_ref, data=extra_context,
defaults=True, unsafe=True
128: (8)                    )
129: (4)                with (output_dir / "package.json").open() as fid:
130: (8)                    temp_data = json.load(fid)
131: (4)                if data.get("devDependencies"):
132: (8)                    for key, value in temp_data["devDependencies"].items():
133: (12)                       data["devDependencies"][key] = value
134: (4)                else:
135: (8)                    data["devDependencies"] = temp_data["devDependencies"].copy()
136: (4)                warnings = []
137: (4)                choice = input("Overwrite scripts in package.json? [n]: ") if interactive
else "y"
138: (4)                if choice.upper().startswith("Y"):
139: (8)                    warnings.append("Updated scripts in package.json")
140: (8)                    data.setdefault("scripts", {})
141: (8)                    for key, value in temp_data["scripts"].items():
142: (12)                       data["scripts"][key] = value
143: (8)                    if "install-ext" in data["scripts"]:
144: (12)                       del data["scripts"]["install-ext"]
145: (8)                    if "prepare" in data["scripts"]:
146: (12)                       del data["scripts"]["prepare"]
147: (4)                else:
148: (8)                    warnings.append("package.json scripts must be updated manually")
149: (4)                data["jupyterlab"]["outputDir"] = temp_data["jupyterlab"]["outputDir"]
150: (4)                linters = {
151: (8)                    "eslintConfig": ".eslintrc.js",
152: (8)                    "eslintIgnore": ".eslintignore",
153: (8)                    "prettier": ".prettierrc",
154: (8)                    "stylelint": ".stylelintrc",
155: (4)                }
156: (4)                for key, file in linters.items():
157: (8)                    if key in temp_data:
158: (12)                       data[key] = temp_data[key]
159: (12)                       linter_file = target / file
160: (12)                       if linter_file.exists():
161: (16)                           linter_file.unlink()
162: (16)                           warnings.append(f"DELETED {file}")
163: (4)                root_jlab_package = files("jupyterlab").joinpath("staging/package.json")
164: (4)                with root_jlab_package.open() as fid:
165: (8)                    root_jlab_data = json.load(fid)
166: (4)                data.setdefault("dependencies", {})
167: (4)                data.setdefault("devDependencies", {})
168: (4)                for key, value in root_jlab_data["resolutions"].items():
169: (8)                    if key in data["dependencies"]:
170: (12)                       data["dependencies"][key] = value.replace("~", "^")
171: (8)                    if key in data["devDependencies"]:
172: (12)                       data["devDependencies"][key] = value.replace("~", "^")
173: (4)                for key in ["scripts", "dependencies", "devDependencies"]:
174: (8)                    if data[key]:
175: (12)                       data[key] = dict(sorted(data[key].items()))
176: (8)                    else:
177: (12)                       del data[key]
178: (4)                data.setdefault("styleModule", "style/index.js")
179: (4)                if isinstance(data.get("sideEffects"), list) and "style/index.js" not in
data["sideEffects"]:
180: (8)                    data["sideEffects"].append("style/index.js")
181: (4)                if "files" in data and "style/index.js" not in data["files"]:
182: (8)                    data["files"].append("style/index.js")
183: (4)                package_file.write_text(json.dumps(data, indent=2))
184: (4)                override_pyproject = False
185: (4)                for p in output_dir.rglob("*"):
186: (8)                    relpath = p.relative_to(output_dir)
187: (8)                    if str(relpath) == "package.json":
188: (12)                       continue
189: (8)                    if p.is_dir():
190: (12)                       continue
191: (8)                    file_target = target / relpath
```

```
192: (8)                    if not file_target.exists():
193: (12)                       file_target.parent.mkdir(parents=True, exist_ok=True)
194: (12)                       shutil.copy(p, file_target)
195: (12)                       if file_target.name == "pyproject.toml":
196: (16)                           override_pyproject = True
197: (8)                    else:
198: (12)                       old_data = p.read_bytes()
199: (12)                       new_data = file_target.read_bytes()
200: (12)                       if old_data == new_data:
201: (16)                           continue
202: (12)                       default = "y" if relpath.as_posix() in RECOMMENDED_TO_OVERRIDE
else "n"
203: (12)                       choice = (
204: (16)                           (input(f'overwrite "{relpath!s}"? [{default}]: ') or default)
205: (16)                           if interactive
206: (16)                           else "n"
207: (12)                       )
208: (12)                       if choice.upper().startswith("Y"):
209: (16)                           shutil.copy(p, file_target)
210: (16)                           if file_target.name == "pyproject.toml":
211: (20)                               override_pyproject = True
212: (12)                       else:
213: (16)                           warnings.append(f"skipped _temp_extension/{relpath!s}")
214: (4)                if override_pyproject:
215: (8)                    if (target / "setup.cfg").exists():
216: (12)                       try:
217: (16)                           import tomli_w
218: (12)                       except ImportError:
219: (16)                           msg = "To update pyproject.toml, you need to install tomli-w"
220: (16)                           print(msg)
221: (12)                       else:
222: (16)                           config = configparser.ConfigParser()
223: (16)                           with (target / "setup.cfg").open() as setup_cfg_file:
224: (20)                               config.read_file(setup_cfg_file)
225: (16)                           pyproject_file = target / "pyproject.toml"
226: (16)                           pyproject = tomllib.loads(pyproject_file.read_text())
227: (16)                           requirements_raw = config.get("options", "install_requires",
fallback=None)
228: (16)                           if requirements_raw is not None:
229: (20)                               requirements = list(
230: (24)                                   filter(
231: (28)                                       lambda r: r and
JUPYTER_SERVER_REQUIREMENT.match(r) is None,
232: (28)                                       requirements_raw.splitlines(),
233: (24)                                   )
234: (20)                               )
235: (16)                           else:
236: (20)                               requirements = []
237: (16)                           pyproject["project"]["dependencies"] = (
238: (20)                               pyproject["project"].get("dependencies", []) +
requirements
239: (16)                           )
240: (16)                           if config.has_section("options.extras_require"):
241: (20)                               for extra, deps_raw in
config.items("options.extras_require"):
242: (24)                                   deps = list(filter(lambda r: r,
deps_raw.splitlines()))
243: (24)                                   if extra in pyproject["project"].get("optional-
dependencies", {}):
244: (28)                                       if pyproject["project"].get("optional-
dependencies") is None:
245: (32)                                           pyproject["project"]["optional-dependencies"]
= {}
246: (28)                                       deps = pyproject["project"]["optional-
dependencies"][extra] + deps
247: (24)                                   pyproject["project"]["optional-dependencies"][extra] =
deps
248: (16)                           pyproject_file.write_text(tomli_w.dumps(pyproject))
249: (16)                           (target / "setup.cfg").unlink()
```

```
250: (16)                       warnings.append("DELETED setup.cfg")
251: (8)                    manifest_in = target / "MANIFEST.in"
252: (8)                    if manifest_in.exists():
253: (12)                       manifest_in.unlink()
254: (12)                       warnings.append("DELETED MANIFEST.in")
255: (4)                for warning in warnings:
256: (8)                    print("**", warning)
257: (4)                print("** Remove _temp_extensions directory when finished")
258: (0)            if __name__ == "__main__":
259: (4)                import argparse
260: (4)                parser = argparse.ArgumentParser(description="Upgrade a JupyterLab extension")
261: (4)                parser.add_argument("--no-input", action="store_true", help="whether to prompt for information")
262: (4)                parser.add_argument("path", action="store", type=str, help="the target path")
263: (4)                parser.add_argument("--vcs-ref", help="the template hash to checkout", default=None)
264: (4)                args = parser.parse_args()
265: (4)                answer_file = Path(args.path) / ".copier-answers.yml"
266: (4)                if answer_file.exists():
267: (8)                    msg = "This script won't do anything for copier template, instead execute in your extension directory:\n\n    copier update"
268: (8)                    if tuple(copier.__version__.split(".")) >= ("8", "0", "0"):
269: (12)                       msg += " --trust"
270: (8)                    print(msg)
271: (4)                else:
272: (8)                    update_extension(args.path, args.vcs_ref, args.no_input is False)

----------------------------------------

File 14 - utils.py:

1: (0)              import functools
2: (0)              import warnings
3: (0)              class jupyterlab_deprecation(Warning):  # noqa
4: (4)                  """Create our own deprecation class, since Python >= 2.7
5: (4)                  silences deprecations by default.
6: (4)                  """
7: (4)                  pass
8: (0)              class deprecated:  # noqa
9: (4)                  """Decorator to mark deprecated functions with warning.
10: (4)                 Adapted from `scikit-image/skimage/_shared/utils.py`.
11: (4)                 Parameters
12: (4)                 ----------
13: (4)                 alt_func : str
14: (8)                     If given, tell user what function to use instead.
15: (4)                 behavior : {'warn', 'raise'}
16: (8)                     Behavior during call to deprecated function: 'warn' = warn user that
17: (8)                     function is deprecated; 'raise' = raise error.
18: (4)                 removed_version : str
19: (8)                     The package version in which the deprecated function will be removed.
20: (4)                 """
21: (4)                 def __init__(self, alt_func=None, behavior="warn", removed_version=None):
22: (8)                     self.alt_func = alt_func
23: (8)                     self.behavior = behavior
24: (8)                     self.removed_version = removed_version
25: (4)                 def __call__(self, func):
26: (8)                     alt_msg = ""
27: (8)                     if self.alt_func is not None:
28: (12)                        alt_msg = " Use ``%s`` instead." % self.alt_func
29: (8)                     rmv_msg = ""
30: (8)                     if self.removed_version is not None:
31: (12)                        rmv_msg = " and will be removed in version %s" % self.removed_version
32: (8)                     msg = "Function ``%s`` is deprecated" % func.__name__ + rmv_msg + "." + alt_msg
33: (8)                     @functools.wraps(func)
34: (8)                     def wrapped(*args, **kwargs):
```

```
35: (12)                        if self.behavior == "warn":
36: (16)                            func_code = func.__code__
37: (16)                            warnings.simplefilter("always", jupyterlab_deprecation)
38: (16)                            warnings.warn_explicit(
39: (20)                                msg,
40: (20)                                category=jupyterlab_deprecation,
41: (20)                                filename=func_code.co_filename,
42: (20)                                lineno=func_code.co_firstlineno + 1,
43: (16)                            )
44: (12)                        elif self.behavior == "raise":
45: (16)                            raise jupyterlab_deprecation(msg)
46: (12)                        return func(*args, **kwargs)
47: (8)                     doc = "**Deprecated function**." + alt_msg
48: (8)                     if wrapped.__doc__ is None:
49: (12)                        wrapped.__doc__ = doc
50: (8)                     else:
51: (12)                        wrapped.__doc__ = doc + "\n\n    " + wrapped.__doc__
52: (8)                     return wrapped

----------------------------------------

File 15 - _version.py:

1: (0)                 from collections import namedtuple
2: (0)                 VersionInfo = namedtuple("VersionInfo", ["major", "minor", "micro",
"releaselevel", "serial"])
3: (0)                 version_info = VersionInfo(4, 2, 5, "final", 0)
4: (0)                 _specifier_ = {"alpha": "a", "beta": "b", "candidate": "rc", "final": ""}
5: (0)                 __version__ = "{}.{}.{}{}".format(
6: (4)                     version_info.major,
7: (4)                     version_info.minor,
8: (4)                     version_info.micro,
9: (4)                     (
10: (8)                        ""
11: (8)                        if version_info.releaselevel == "final"
12: (8)                        else _specifier_[version_info.releaselevel] + str(version_info.serial)
13: (4)                    ),
14: (0)                )

----------------------------------------

File 16 - __init__.py:

1: (0)                 """Server extension for JupyterLab."""
2: (0)                 from ._version import __version__  # noqa
3: (0)                 from .serverextension import load_jupyter_server_extension  # noqa
4: (0)                 from .handlers.announcements import (
5: (4)                     CheckForUpdate,  # noqa
6: (4)                     CheckForUpdateABC,  # noqa
7: (4)                     NeverCheckForUpdate,  # noqa
8: (0)                 )
9: (0)                 def _jupyter_server_extension_paths():
10: (4)                    return [{"module": "jupyterlab"}]
11: (0)                def _jupyter_server_extension_points():
12: (4)                    from .labapp import LabApp
13: (4)                    return [{"module": "jupyterlab", "app": LabApp}]

----------------------------------------

File 17 - __main__.py:

1: (0)                 import sys
2: (0)                 from jupyterlab.labapp import main
3: (0)                 sys.exit(main())

----------------------------------------

File 18 - manager.py:
```

```
1: (0)              """Base classes for the extension manager."""
2: (0)              import json
3: (0)              import re
4: (0)              from dataclasses import dataclass, field, fields, replace
5: (0)              from pathlib import Path
6: (0)              from typing import Dict, FrozenSet, List, Optional, Set, Tuple, Union
7: (0)              import tornado
8: (0)              from jupyterlab_server.translation_utils import translator
9: (0)              from traitlets import Enum
10: (0)             from traitlets.config import Configurable, LoggingConfigurable
11: (0)             from jupyterlab.commands import (
12: (4)                 _AppHandler,
13: (4)                 _ensure_options,
14: (4)                 disable_extension,
15: (4)                 enable_extension,
16: (4)                 get_app_info,
17: (0)             )
18: (0)             PYTHON_TO_SEMVER = {"a": "-alpha.", "b": "-beta.", "rc": "-rc."}
19: (0)             def _ensure_compat_errors(info, app_options):
20: (4)                 """Ensure that the app info has compat_errors field"""
21: (4)                 handler = _AppHandler(app_options)
22: (4)                 info["compat_errors"] = handler._get_extension_compat()
23: (0)             _message_map = {
24: (4)                 "install": re.compile(r"(?P<name>.*) needs to be included in build"),
25: (4)                 "uninstall": re.compile(r"(?P<name>.*) needs to be removed from build"),
26: (4)                 "update": re.compile(r"(?P<name>.*) changed from (?P<oldver>.*) to (?
P<newver>.*)"),
27: (0)             }
28: (0)             def _build_check_info(app_options):
29: (4)                 """Get info about packages scheduled for (un)install/update"""
30: (4)                 handler = _AppHandler(app_options)
31: (4)                 messages = handler.build_check(fast=True)
32: (4)                 status = {"install": [], "uninstall": [], "update": []}
33: (4)                 for msg in messages:
34: (8)                     for key, pattern in _message_map.items():
35: (12)                        match = pattern.match(msg)
36: (12)                        if match:
37: (16)                            status[key].append(match.group("name"))
38: (4)                 return status
39: (0)             @dataclass(frozen=True)
40: (0)             class ExtensionPackage:
41: (4)                 """Extension package entry.
42: (4)                 Attributes:
43: (8)                     name: Package name
44: (8)                     description: Package description
45: (8)                     homepage_url: Package home page
46: (8)                     pkg_type: Type of package - ["prebuilt", "source"]
47: (8)                     allowed: [optional] Whether this extension is allowed or not - default
True
48: (8)                     approved: [optional] Whether the package is approved by your
administrators - default False
49: (8)                     companion: [optional] Type of companion for the frontend extension -
[None, "kernel", "server"]; default None
50: (8)                     core: [optional] Whether the package is a core package or not -
default False
51: (8)                     enabled: [optional] Whether the package is enabled or not - default
False
52: (8)                     install: [optional] Extension package installation instructions -
default None
53: (8)                     installed: [optional] Whether the extension is currently installed -
default None
54: (8)                     installed_version: [optional] Installed version - default ""
55: (8)                     latest_version: [optional] Latest available version - default ""
56: (8)                     status: [optional] Package status - ["ok", "warning", "error"];
default "ok"
57: (8)                     author: [optional] Package author - default None
58: (8)                     license: [optional] Package license - default None
59: (8)                     bug_tracker_url: [optional] Package bug tracker URL - default None
60: (8)                     documentation_url: [optional] Package documentation URL - default None
```

```
61: (8)                     package_manager_url: Package home page in the package manager -
default None
62: (8)                     repository_url: [optional] Package code repository URL - default None
63: (4)                 """
64: (4)                 name: str
65: (4)                 description: str
66: (4)                 homepage_url: str
67: (4)                 pkg_type: str
68: (4)                 allowed: bool = True
69: (4)                 approved: bool = False
70: (4)                 companion: Optional[str] = None
71: (4)                 core: bool = False
72: (4)                 enabled: bool = False
73: (4)                 install: Optional[dict] = None
74: (4)                 installed: Optional[bool] = None
75: (4)                 installed_version: str = ""
76: (4)                 latest_version: str = ""
77: (4)                 status: str = "ok"
78: (4)                 author: Optional[str] = None
79: (4)                 license: Optional[str] = None
80: (4)                 bug_tracker_url: Optional[str] = None
81: (4)                 documentation_url: Optional[str] = None
82: (4)                 package_manager_url: Optional[str] = None
83: (4)                 repository_url: Optional[str] = None
84: (0)             @dataclass(frozen=True)
85: (0)             class ActionResult:
86: (4)                 """Action result
87: (4)                 Attributes:
88: (8)                     status: Action status - ["ok", "warning", "error"]
89: (8)                     message: Action status explanation
90: (8)                     needs_restart: Required action follow-up - Valid follow-up are
"frontend", "kernel" and "server"
91: (4)                 """
92: (4)                 status: str
93: (4)                 message: Optional[str] = None
94: (4)                 needs_restart: List[str] = field(default_factory=list)
95: (0)             @dataclass(frozen=True)
96: (0)             class PluginManagerOptions:
97: (4)                 """Plugin manager options.
98: (4)                 Attributes:
99: (8)                     lock_all: Whether to lock (prevent enabling/disabling) all plugins.
100: (8)                    lock_rules: A list of plugins or extensions that cannot be toggled.
101: (12)                       If extension name is provided, all its plugins will be disabled.
102: (12)                       The plugin names need to follow colon-separated format of
`extension:plugin`.
103: (4)                """
104: (4)                lock_rules: FrozenSet[str] = field(default_factory=frozenset)
105: (4)                lock_all: bool = False
106: (0)            @dataclass(frozen=True)
107: (0)            class ExtensionManagerOptions(PluginManagerOptions):
108: (4)                """Extension manager options.
109: (4)                Attributes:
110: (8)                    allowed_extensions_uris: A list of comma-separated URIs to get the
allowed extensions list
111: (8)                    blocked_extensions_uris: A list of comma-separated URIs to get the
blocked extensions list
112: (8)                    listings_refresh_seconds: The interval delay in seconds to refresh the
lists
113: (8)                    listings_tornado_options: The optional kwargs to use for the listings
HTTP requests as described on
https://www.tornadoweb.org/en/stable/httpclient.html#tornado.httpclient.HTTPRequest
114: (4)                """
115: (4)                allowed_extensions_uris: Set[str] = field(default_factory=set)
116: (4)                blocked_extensions_uris: Set[str] = field(default_factory=set)
117: (4)                listings_refresh_seconds: int = 60 * 60
118: (4)                listings_tornado_options: dict = field(default_factory=dict)
119: (0)            @dataclass(frozen=True)
120: (0)            class ExtensionManagerMetadata:
121: (4)                """Extension manager metadata.
```

```
(4)                Attributes:
(8)                    name: Extension manager name to be displayed
(8)                    can_install: Whether the extension manager can un-/install packages (default False)
(8)                    install_path: Installation path for the extensions (default None); e.g. environment path
(4)                """
(4)                name: str
(4)                can_install: bool = False
(4)                install_path: Optional[str] = None
(0)            @dataclass
(0)            class ExtensionsCache:
(4)                """Extensions cache
(4)                Attributes:
(8)                    cache: Extension list per page
(8)                    last_page: Last available page result
(4)                """
(4)                cache: Dict[int, Optional[Dict[str, ExtensionPackage]]] = field(default_factory=dict)
(4)                last_page: int = 1
(0)            class PluginManager(LoggingConfigurable):
(4)                """Plugin manager enables or disables plugins unless locked.
(4)                It can also disable/enable all plugins in an extension.
(4)                Args:
(8)                    app_options: Application options
(8)                    ext_options: Plugin manager (subset of extension manager) options
(8)                    parent: Configurable parent
(4)                Attributes:
(8)                    app_options: Application options
(8)                    options: Plugin manager options
(4)                """
(4)                level = Enum(
(8)                    values=["sys_prefix", "user", "system"],
(8)                    default_value="sys_prefix",
(8)                    help="Level at which to manage plugins: sys_prefix, user, system",
(4)                ).tag(config=True)
(4)                def __init__(
(8)                    self,
(8)                    app_options: Optional[dict] = None,
(8)                    ext_options: Optional[dict] = None,
(8)                    parent: Optional[Configurable] = None,
(4)                ) -> None:
(8)                    super().__init__(parent=parent)
(8)                    self.log.debug(
(12)                       "Plugins in %s will managed on the %s level", self.__class__.__name__, self.level
(8)                    )
(8)                    self.app_options = _ensure_options(app_options)
(8)                    plugin_options_field = {f.name for f in fields(PluginManagerOptions)}
(8)                    plugin_options = {
(12)                       option: value
(12)                       for option, value in (ext_options or {}).items()
(12)                       if option in plugin_options_field
(8)                    }
(8)                    self.options = PluginManagerOptions(**plugin_options)
(4)                async def plugin_locks(self) -> dict:
(8)                    """Get information about locks on plugin enabling/disabling"""
(8)                    return {
(12)                       "lockRules": list(self.options.lock_rules),
(12)                       "allLocked": self.options.lock_all,
(8)                    }
(4)                def _find_locked(self, plugins_or_extensions: List[str]) -> FrozenSet[str]:
(8)                    """Find a subset of plugins (or extensions) which are locked"""
(8)                    if self.options.lock_all:
(12)                       return set(plugins_or_extensions)
(8)                    locked_subset = set()
(8)                    extensions_with_locked_plugins = {
(12)                       plugin.split(":")[0] for plugin in self.options.lock_rules
186: (8)                        }
187: (8)                        for plugin in plugins_or_extensions:
188: (12)                           if ":" in plugin:
189: (16)                               if plugin in self.options.lock_rules:
190: (20)                                   locked_subset.add(plugin)
191: (12)                           elif plugin in extensions_with_locked_plugins:
192: (16)                               locked_subset.add(plugin)
193: (8)                        return locked_subset
194: (4)                    async def disable(self, plugins: Union[str, List[str]]) -> ActionResult:
195: (8)                        """Disable a set of plugins (or an extension).
196: (8)                        Args:
197: (12)                           plugins: The list of plugins to disable
198: (8)                        Returns:
199: (12)                           The action result
200: (8)                        """
201: (8)                        plugins = plugins if isinstance(plugins, list) else [plugins]
202: (8)                        locked = self._find_locked(plugins)
203: (8)                        trans = translator.load("jupyterlab")
204: (8)                        if locked:
205: (12)                           return ActionResult(
206: (16)                               status="error",
207: (16)                               message=trans.gettext(
208: (20)                                   "The following plugins cannot be disabled as they are
locked: "
209: (16)                               )
210: (16)                               + ", ".join(locked),
211: (12)                           )
212: (8)                        try:
213: (12)                           for plugin in plugins:
214: (16)                               disable_extension(plugin, app_options=self.app_options,
level=self.level)
215: (12)                           return ActionResult(status="ok", needs_restart=["frontend"])
216: (8)                        except Exception as err:
217: (12)                           return ActionResult(status="error", message=repr(err))
218: (4)                    async def enable(self, plugins: Union[str, List[str]]) -> ActionResult:
219: (8)                        """Enable a set of plugins (or an extension).
220: (8)                        Args:
221: (12)                           plugins: The list of plugins to enable
222: (8)                        Returns:
223: (12)                           The action result
224: (8)                        """
225: (8)                        plugins = plugins if isinstance(plugins, list) else [plugins]
226: (8)                        locked = self._find_locked(plugins)
227: (8)                        trans = translator.load("jupyterlab")
228: (8)                        if locked:
229: (12)                           return ActionResult(
230: (16)                               status="error",
231: (16)                               message=trans.gettext(
232: (20)                                   "The following plugins cannot be enabled as they are
locked: "
233: (16)                               )
234: (16)                               + ", ".join(locked),
235: (12)                           )
236: (8)                        try:
237: (12)                           for plugin in plugins:
238: (16)                               enable_extension(plugin, app_options=self.app_options,
level=self.level)
239: (12)                           return ActionResult(status="ok", needs_restart=["frontend"])
240: (8)                        except Exception as err:
241: (12)                           return ActionResult(status="error", message=repr(err))
242: (0)                class ExtensionManager(PluginManager):
243: (4)                    """Base abstract extension manager.
244: (4)                    Note:
245: (8)                        Any concrete implementation will need to implement the five
246: (8)                        following abstract methods:
247: (8)                        - :ref:`metadata`
248: (8)                        - :ref:`get_latest_version`
249: (8)                        - :ref:`list_packages`
250: (8)                        - :ref:`install`
```

```
(8)                     - :ref:`uninstall`
(8)                     It could be interesting to override the :ref:`get_normalized_name`
(8)                     method too.
(4)                 Args:
(8)                     app_options: Application options
(8)                     ext_options: Extension manager options
(8)                     parent: Configurable parent
(4)                 Attributes:
(8)                     log: Logger
(8)                     app_dir: Application directory
(8)                     core_config: Core configuration
(8)                     app_options: Application options
(8)                     options: Extension manager options
(4)                 """
(4)                 def __init__(
(8)                     self,
(8)                     app_options: Optional[dict] = None,
(8)                     ext_options: Optional[dict] = None,
(8)                     parent: Optional[Configurable] = None,
(4)                 ) -> None:
(8)                     super().__init__(app_options=app_options, ext_options=ext_options, parent=parent)
(8)                     self.log = self.app_options.logger
(8)                     self.app_dir = Path(self.app_options.app_dir)
(8)                     self.core_config = self.app_options.core_config
(8)                     self.options = ExtensionManagerOptions(**(ext_options or {}))
(8)                     self._extensions_cache: Dict[Optional[str], ExtensionsCache] = {}
(8)                     self._listings_cache: Optional[dict] = None
(8)                     self._listings_block_mode = True
(8)                     self._listing_fetch: Optional[tornado.ioloop.PeriodicCallback] = None
(8)                     if len(self.options.allowed_extensions_uris) or len(self.options.blocked_extensions_uris):
(12)                        self._listings_block_mode = len(self.options.allowed_extensions_uris) == 0
(12)                        if not self._listings_block_mode and len(self.options.blocked_extensions_uris) > 0:
(16)                            self.log.warning(
(20)                                "You have define simultaneously blocked and allowed extensions listings. The allowed listing will take precedence."
(16)                            )
(12)                        self._listing_fetch = tornado.ioloop.PeriodicCallback(
(16)                            self._fetch_listings,
(16)                            callback_time=self.options.listings_refresh_seconds * 1000,
(16)                            jitter=0.1,
(12)                        )
(12)                        self._listing_fetch.start()
(4)                 def __del__(self):
(8)                     if self._listing_fetch is not None:
(12)                        self._listing_fetch.stop()
(4)                 @property
(4)                 def metadata(self) -> ExtensionManagerMetadata:
(8)                     """Extension manager metadata."""
(8)                     raise NotImplementedError()
(4)                 async def get_latest_version(self, extension: str) -> Optional[str]:
(8)                     """Return the latest available version for a given extension.
(8)                     Args:
(12)                        pkg: The extension name
(8)                     Returns:
(12)                        The latest available version
(8)                     """
(8)                     raise NotImplementedError()
(4)                 async def list_packages(
(8)                     self, query: str, page: int, per_page: int
(4)                 ) -> Tuple[Dict[str, ExtensionPackage], Optional[int]]:
(8)                     """List the available extensions.
(8)                     Args:
(12)                        query: The search extension query
(12)                        page: The result page
(12)                        per_page: The number of results per page
```

```
315: (8)                    Returns:
316: (12)                       The available extensions in a mapping {name: metadata}
317: (12)                       The results last page; None if the manager does not support pagination
318: (8)                    """
319: (8)                    raise NotImplementedError()
320: (4)                async def install(self, extension: str, version: Optional[str] = None) -> ActionResult:
321: (8)                    """Install the required extension.
322: (8)                    Note:
323: (12)                       If the user must be notified with a message (like asking to restart the
324: (12)                       server), the result should be
325: (12)                       {"status": "warning", "message": "<explanation for the user>"}
326: (8)                    Args:
327: (12)                       extension: The extension name
328: (12)                       version: The version to install; default None (i.e. the latest possible)
329: (8)                    Returns:
330: (12)                       The action result
331: (8)                    """
332: (8)                    raise NotImplementedError()
333: (4)                async def uninstall(self, extension: str) -> ActionResult:
334: (8)                    """Uninstall the required extension.
335: (8)                    Note:
336: (12)                       If the user must be notified with a message (like asking to restart the
337: (12)                       server), the result should be
338: (12)                       {"status": "warning", "message": "<explanation for the user>"}
339: (8)                    Args:
340: (12)                       extension: The extension name
341: (8)                    Returns:
342: (12)                       The action result
343: (8)                    """
344: (8)                    raise NotImplementedError()
345: (4)                @staticmethod
346: (4)                def get_semver_version(version: str) -> str:
347: (8)                    """Convert a Python version to Semver version.
348: (8)                    It:
349: (8)                    - drops ``.devN`` and ``.postN``
350: (8)                    - converts ``aN``, ``bN`` and ``rcN`` to ``-alpha.N``, ``-beta.N``, ``-rc.N`` respectively
351: (8)                    Args:
352: (12)                       version: Version to convert
353: (8)                    Returns
354: (12)                       Semver compatible version
355: (8)                    """
356: (8)                    return re.sub(
357: (12)                       r"(a|b|rc)(\d+)$",
358: (12)                       lambda m: f"{PYTHON_TO_SEMVER[m.group(1)]}{m.group(2)}",
359: (12)                       re.subn(r"\.(dev|post)\d+", "", version)[0],
360: (8)                    )
361: (4)                def get_normalized_name(self, extension: ExtensionPackage) -> str:
362: (8)                    """Normalize extension name.
363: (8)                    Extension have multiple parts, npm package, Python package,...
364: (8)                    Sub-classes may override this method to ensure the name of
365: (8)                    an extension from the service provider and the local installed
366: (8)                    listing is matching.
367: (8)                    Args:
368: (12)                       extension: The extension metadata
369: (8)                    Returns:
370: (12)                       The normalized name
371: (8)                    """
372: (8)                    return extension.name
373: (4)                async def list_extensions(
374: (8)                    self, query: Optional[str] = None, page: int = 1, per_page: int = 30
375: (4)                ) -> Tuple[List[ExtensionPackage], Optional[int]]:
376: (8)                    """List extensions for a given ``query`` search term.
377: (8)                    This will return the extensions installed (if ``query`` is None) or
```

```
(8)             available if allowed by the listing settings.
(8)             Args:
(12)                query: [optional] Query search term.
(8)             Returns:
(12)                The extensions
(12)                Last page of results
(8)             """
(8)             if query not in self._extensions_cache or page not in self._extensions_cache[query].cache:
(12)                await self.refresh(query, page, per_page)
(8)             if self._listings_cache is None and self._listing_fetch is not None:
(12)                await self._listing_fetch.callback()
(8)             cache = self._extensions_cache[query].cache[page]
(8)             if cache is None:
(12)                cache = {}
(8)             extensions = list(cache.values())
(8)             if query is not None and self._listings_cache is not None:
(12)                listing = list(self._listings_cache)
(12)                extensions = []
(12)                if self._listings_block_mode:
(16)                    for name, ext in cache.items():
(20)                        if name not in listing:
(24)                            extensions.append(replace(ext, allowed=True))
(20)                        elif ext.installed_version:
(24)                            self.log.warning(f"Blocked extension '{name}' is installed.")
(24)                            extensions.append(replace(ext, allowed=False))
(12)                else:
(16)                    for name, ext in cache.items():
(20)                        if name in listing:
(24)                            extensions.append(replace(ext, allowed=True))
(20)                        elif ext.installed_version:
(24)                            self.log.warning(f"Not allowed extension '{name}' is installed.")
(24)                            extensions.append(replace(ext, allowed=False))
(8)             return extensions, self._extensions_cache[query].last_page
(4)         async def refresh(self, query: Optional[str], page: int, per_page: int) -> None:
(8)             """Refresh the list of extensions."""
(8)             if query in self._extensions_cache:
(12)                self._extensions_cache[query].cache[page] = None
(8)             await self._update_extensions_list(query, page, per_page)
(4)         async def _fetch_listings(self) -> None:
(8)             """Fetch the listings for the extension manager."""
(8)             rules = []
(8)             client = tornado.httpclient.AsyncHTTPClient()
(8)             if self._listings_block_mode:
(12)                if len(self.options.blocked_extensions_uris):
(16)                    self.log.info(
(20)                        f"Fetching blocked extensions from {self.options.blocked_extensions_uris}"
(16)                    )
(16)                    for blocked_extensions_uri in self.options.blocked_extensions_uris:
(20)                        r = await client.fetch(
(24)                            blocked_extensions_uri,
(24)                            **self.options.listings_tornado_options,
(20)                        )
(20)                        j = json.loads(r.body)
(20)                        rules.extend(j.get("blocked_extensions", []))
(8)             elif len(self.options.allowed_extensions_uris):
(12)                self.log.info(
(16)                    f"Fetching allowed extensions from { self.options.allowed_extensions_uris}"
(12)                )
(12)                for allowed_extensions_uri in self.options.allowed_extensions_uris:
(16)                    r = await client.fetch(
(20)                        allowed_extensions_uri,
(20)                        **self.options.listings_tornado_options,
(16)                    )
(16)                    j = json.loads(r.body)
(16)                    rules.extend(j.get("allowed_extensions", []))
(8)             self._listings_cache = {r["name"]: r for r in rules}
(4)         async def _get_installed_extensions(
(8)             self, get_latest_version=True
(4)         ) -> Dict[str, ExtensionPackage]:
(8)             """Get the installed extensions.
(8)             Args:
(12)                get_latest_version: Whether to fetch the latest extension version or not.
(8)             Returns:
(12)                The installed extensions as a mapping {name: metadata}
(8)             """
(8)             app_options = self.app_options
(8)             info = get_app_info(app_options=app_options)
(8)             build_check_info = _build_check_info(app_options)
(8)             _ensure_compat_errors(info, app_options)
(8)             extensions = {}
(8)             for name, data in info["federated_extensions"].items():
(12)                status = "ok"
(12)                pkg_info = data
(12)                if info["compat_errors"].get(name, None):
(16)                    status = "error"
(12)                normalized_name = self._normalize_name(name)
(12)                pkg = ExtensionPackage(
(16)                    name=normalized_name,
(16)                    description=pkg_info.get("description", ""),
(16)                    homepage_url=data.get("url", ""),
(16)                    enabled=(name not in info["disabled"]),
(16)                    core=False,
(16) latest_version=ExtensionManager.get_semver_version(data["version"]),
(16)                    installed=True,
(16) installed_version=ExtensionManager.get_semver_version(data["version"]),
(16)                    status=status,
(16)                    install=data.get("install", {}),
(16)                    pkg_type="prebuilt",
(16)                    companion=self._get_companion(data),
(16)                    author=data.get("author", {}).get("name", data.get("author")),
(16)                    license=data.get("license"),
(16)                    bug_tracker_url=data.get("bugs", {}).get("url"),
(16)                    repository_url=data.get("repository", {}).get("url", data.get("repository")),
(12)                )
(12)                if get_latest_version:
(16)                    pkg = replace(pkg, latest_version=await self.get_latest_version(pkg.name))
(12)                extensions[normalized_name] = pkg
(8)             for name, data in info["extensions"].items():
(12)                if name in info["shadowed_exts"]:
(16)                    continue
(12)                status = "ok"
(12)                if info["compat_errors"].get(name, None):
(16)                    status = "error"
(12)                else:
(16)                    for packages in build_check_info.values():
(20)                        if name in packages:
(24)                            status = "warning"
(12)                normalized_name = self._normalize_name(name)
(12)                pkg = ExtensionPackage(
(16)                    name=normalized_name,
(16)                    description=data.get("description", ""),
(16)                    homepage_url=data["url"],
(16)                    enabled=(name not in info["disabled"]),
(16)                    core=False,
(16)
```

```
latest_version=ExtensionManager.get_semver_version(data["version"]),
503: (16)                           installed=True,
504: (16)
installed_version=ExtensionManager.get_semver_version(data["version"]),
505: (16)                           status=status,
506: (16)                           pkg_type="source",
507: (16)                           companion=self._get_companion(data),
508: (16)                           author=data.get("author", {}).get("name", data.get("author")),
509: (16)                           license=data.get("license"),
510: (16)                           bug_tracker_url=data.get("bugs", {}).get("url"),
511: (16)                           repository_url=data.get("repository", {}).get("url",
data.get("repository")),
512: (12)                       )
513: (12)                       if get_latest_version:
514: (16)                           pkg = replace(pkg, latest_version=await
self.get_latest_version(pkg.name))
515: (12)                       extensions[normalized_name] = pkg
516: (8)                    for name in build_check_info["uninstall"]:
517: (12)                       data = self._get_scheduled_uninstall_info(name)
518: (12)                       if data is not None:
519: (16)                           normalized_name = self._normalize_name(name)
520: (16)                           pkg = ExtensionPackage(
521: (20)                               name=normalized_name,
522: (20)                               description=data.get("description", ""),
523: (20)                               homepage_url=data.get("homepage", ""),
524: (20)                               installed=False,
525: (20)                               enabled=False,
526: (20)                               core=False,
527: (20)
latest_version=ExtensionManager.get_semver_version(data["version"]),
528: (20)
installed_version=ExtensionManager.get_semver_version(data["version"]),
529: (20)                               status="warning",
530: (20)                               pkg_type="prebuilt",
531: (20)                               author=data.get("author", {}).get("name",
data.get("author")),
532: (20)                               license=data.get("license"),
533: (20)                               bug_tracker_url=data.get("bugs", {}).get("url"),
534: (20)                               repository_url=data.get("repository", {}).get("url",
data.get("repository")),
535: (16)                           )
536: (16)                           extensions[normalized_name] = pkg
537: (8)                    return extensions
538: (4)                def _get_companion(self, data: dict) -> Optional[str]:
539: (8)                    companion = None
540: (8)                    if "discovery" in data["jupyterlab"]:
541: (12)                       if "server" in data["jupyterlab"]["discovery"]:
542: (16)                           companion = "server"
543: (12)                       elif "kernel" in data["jupyterlab"]["discovery"]:
544: (16)                           companion = "kernel"
545: (8)                    return companion
546: (4)                def _get_scheduled_uninstall_info(self, name) -> Optional[dict]:
547: (8)                    """Get information about a package that is scheduled for
uninstallation"""
548: (8)                    target = self.app_dir / "staging" / "node_modules" / name /
"package.json"
549: (8)                    if target.exists():
550: (12)                       with target.open() as fid:
551: (16)                           return json.load(fid)
552: (8)                    else:
553: (12)                       return None
554: (4)                def _normalize_name(self, name: str) -> str:
555: (8)                    """Normalize extension name; by default does nothing.
556: (8)                    Args:
557: (12)                       name: Extension name
558: (8)                    Returns:
559: (12)                       Normalized name
560: (8)                    """
561: (8)                    return name
```

```
562: (4)                async def _update_extensions_list(
563: (8)                    self, query: Optional[str] = None, page: int = 1, per_page: int = 30
564: (4)                ) -> None:
565: (8)                    """Update the list of extensions"""
566: (8)                    last_page = None
567: (8)                    if query is not None:
568: (12)                       extensions, last_page = await self.list_packages(query, page,
per_page)
569: (8)                    else:
570: (12)                       extensions = await self._get_installed_extensions()
571: (8)                    if query in self._extensions_cache:
572: (12)                       self._extensions_cache[query].cache[page] = extensions
573: (12)                       self._extensions_cache[query].last_page = last_page or 1
574: (8)                    else:
575: (12)                       self._extensions_cache[query] = ExtensionsCache({page:
extensions}, last_page or 1)

----------------------------------------

File 19 - pypi.py:

1: (0)              """Extension manager using pip as package manager and PyPi.org as packages
source."""
2: (0)              import asyncio
3: (0)              import http.client
4: (0)              import io
5: (0)              import json
6: (0)              import math
7: (0)              import re
8: (0)              import sys
9: (0)              import tempfile
10: (0)             import xmlrpc.client
11: (0)             from datetime import datetime, timedelta, timezone
12: (0)             from functools import partial
13: (0)             from itertools import groupby
14: (0)             from os import environ
15: (0)             from pathlib import Path
16: (0)             from subprocess import CalledProcessError, run
17: (0)             from tarfile import TarFile
18: (0)             from typing import Any, Callable, Dict, List, Optional, Tuple
19: (0)             from urllib.parse import urlparse
20: (0)             from zipfile import ZipFile
21: (0)             import httpx
22: (0)             import tornado
23: (0)             from async_lru import alru_cache
24: (0)             from traitlets import CFloat, CInt, Unicode, config, observe
25: (0)             from jupyterlab._version import __version__
26: (0)             from jupyterlab.extensions.manager import (
27: (4)                 ActionResult,
28: (4)                 ExtensionManager,
29: (4)                 ExtensionManagerMetadata,
30: (4)                 ExtensionPackage,
31: (0)             )
32: (0)             class ProxiedTransport(xmlrpc.client.Transport):
33: (4)                 def set_proxy(self, host, port=None, headers=None):
34: (8)                     self.proxy = host, port
35: (8)                     self.proxy_headers = headers
36: (4)                 def make_connection(self, host):
37: (8)                     connection = http.client.HTTPConnection(*self.proxy)
38: (8)                     connection.set_tunnel(host, headers=self.proxy_headers)
39: (8)                     self._connection = host, connection
40: (8)                     return connection
41: (0)             xmlrpc_transport_override = None
42: (0)             all_proxy_url = environ.get("ALL_PROXY")
43: (0)             http_proxy_url = environ.get("http_proxy") or environ.get("HTTP_PROXY") or
all_proxy_url
44: (0)             https_proxy_url = (
45: (4)                 environ.get("https_proxy") or environ.get("HTTPS_PROXY") or http_proxy_url
or all_proxy_url
(0)             )
(0)             proxies = None
(0)             if http_proxy_url:
(4)                 http_proxy = urlparse(http_proxy_url)
(4)                 proxy_host, _, proxy_port = http_proxy.netloc.partition(":")
(4)                 proxies = {
(8)                     "http://": http_proxy_url,
(8)                     "https://": https_proxy_url,
(4)                 }
(4)                 xmlrpc_transport_override = ProxiedTransport()
(4)                 xmlrpc_transport_override.set_proxy(proxy_host, proxy_port)
(0)             async def _fetch_package_metadata(
(4)                 client: httpx.AsyncClient,
(4)                 name: str,
(4)                 latest_version: str,
(4)                 base_url: str,
(0)             ) -> dict:
(4)                 response = await client.get(
(8)                     base_url + f"/{name}/{latest_version}/json",
(8)                     headers={"Content-Type": "application/json"},
(4)                 )
(4)                 if response.status_code < 400:  # noqa PLR2004
(8)                     data = json.loads(response.text).get("info")
(8)                     return {
(12)                        k: data.get(k)
(12)                        for k in [
(16)                            "author",
(16)                            "bugtrack_url",
(16)                            "docs_url",
(16)                            "home_page",
(16)                            "license",
(16)                            "package_url",
(16)                            "project_url",
(16)                            "project_urls",
(16)                            "summary",
(12)                        ]
(8)                     }
(4)                 else:
(8)                     return {}
(0)             class PyPIExtensionManager(ExtensionManager):
(4)                 """Extension manager using pip as package manager and PyPi.org as packages source."""
(4)                 base_url = Unicode("https://pypi.org/pypi", config=True, help="The base URL of PyPI index.")
(4)                 cache_timeout = CFloat(
(8)                     5 * 60.0, config=True, help="PyPI extensions list cache timeout in seconds."
(4)                 )
(4)                 package_metadata_cache_size = CInt(
(8)                     1500, config=True, help="The cache size for package metadata."
(4)                 )
(4)                 rpc_request_throttling = CFloat(
(8)                     1.0,
(8)                     config=True,
(8)                     help="Throttling time in seconds between PyPI requests using the XML-RPC API.",
(4)                 )
(4)                 def __init__(
(8)                     self,
(8)                     app_options: Optional[dict] = None,
(8)                     ext_options: Optional[dict] = None,
(8)                     parent: Optional[config.Configurable] = None,
(4)                 ) -> None:
(8)                     super().__init__(app_options, ext_options, parent)
(8)                     self._httpx_client = httpx.AsyncClient(proxies=proxies)
(8)                     self._fetch_package_metadata = partial(_fetch_package_metadata, self._httpx_client)
(8)                     self._observe_package_metadata_cache_size({"new": self.package_metadata_cache_size})
```

```
        self._rpc_client = xmlrpc.client.ServerProxy(
            self.base_url, transport=xmlrpc_transport_override
        )
        self.__last_all_packages_request_time = datetime.now(tz=timezone.utc) - timedelta(
            seconds=self.cache_timeout * 1.01
        )
        self.__all_packages_cache = None
        self.log.debug(f"Extensions list will be fetched from {self.base_url}.")
        if xmlrpc_transport_override:
            self.log.info(
                f"Extensions will be fetched using proxy, proxy host and port: {xmlrpc_transport_override.proxy}"
            )
    @property
    def metadata(self) -> ExtensionManagerMetadata:
        """Extension manager metadata."""
        return ExtensionManagerMetadata("PyPI", True, sys.prefix)
    async def get_latest_version(self, pkg: str) -> Optional[str]:
        """Return the latest available version for a given extension.
        Args:
            pkg: The extension to search for
        Returns:
            The latest available version
        """
        try:
            response = await self._httpx_client.get(
                self.base_url + f"/{pkg}/json", headers={"Content-Type": "application/json"}
            )
            if response.status_code < 400:  # noqa PLR2004
                data = json.loads(response.content).get("info", {})
            else:
                self.log.debug(f"Failed to get package information on PyPI; {response!s}")
                return None
        except Exception:
            return None
        else:
            return ExtensionManager.get_semver_version(data.get("version", "")) or None
    def get_normalized_name(self, extension: ExtensionPackage) -> str:
        """Normalize extension name.
        Extension have multiple parts, npm package, Python package,...
        Sub-classes may override this method to ensure the name of
        an extension from the service provider and the local installed
        listing is matching.
        Args:
            extension: The extension metadata
        Returns:
            The normalized name
        """
        if extension.install is not None:
            install_metadata = extension.install
            if install_metadata["packageManager"] == "python":
                return self._normalize_name(install_metadata["packageName"])
        return self._normalize_name(extension.name)
    async def __throttleRequest(self, recursive: bool, fn: Callable, *args) -> Any:  # noqa
        """Throttle XMLRPC API request
        Args:
            recursive: Whether to call the throttling recursively once or not.
            fn: API method to call
            *args: API method arguments
        Returns:
            Result of the method
        Raises:
            xmlrpc.client.Fault
```

```
171: (8)                """
172: (8)                current_loop = tornado.ioloop.IOLoop.current()
173: (8)                try:
174: (12)                   data = await current_loop.run_in_executor(None, fn, *args)
175: (8)                except xmlrpc.client.Fault as err:
176: (12)                   if err.faultCode == -32500 and err.faultString.startswith(  # noqa
PLR2004
177: (16)                       "HTTPTooManyRequests:"
178: (12)                   ):
179: (16)                       delay = 1.01
180: (16)                       match = re.search(r"Limit may reset in (\d+) seconds.",
err.faultString)
181: (16)                       if match is not None:
182: (20)                           delay = int(match.group(1) or "1")
183: (16)                       self.log.info(
184: (20)                           f"HTTPTooManyRequests - Perform next call to PyPI XMLRPC
API in {delay}s."
185: (16)                       )
186: (16)                       await asyncio.sleep(delay * self.rpc_request_throttling +
0.01)
187: (16)                       if recursive:
188: (20)                           data = await self.__throttleRequest(False, fn, *args)
189: (16)                       else:
190: (20)                           data = await current_loop.run_in_executor(None, fn, *args)
191: (8)                return data
192: (4)            @observe("package_metadata_cache_size")
193: (4)            def _observe_package_metadata_cache_size(self, change):
194: (8)                self._fetch_package_metadata = alru_cache(maxsize=change["new"])(
195: (12)                   partial(_fetch_package_metadata, self._httpx_client)
196: (8)                )
197: (4)            async def list_packages(
198: (8)                self, query: str, page: int, per_page: int
199: (4)            ) -> Tuple[Dict[str, ExtensionPackage], Optional[int]]:
200: (8)                """List the available extensions.
201: (8)                Note:
202: (12)                   This will list the packages based on the classifier
203: (16)                       Framework :: Jupyter :: JupyterLab :: Extensions :: Prebuilt
204: (12)                   Then it filters it with the query
205: (12)                   We do not try to check if they are compatible (version wise)
206: (8)                Args:
207: (12)                   query: The search extension query
208: (12)                   page: The result page
209: (12)                   per_page: The number of results per page
210: (8)                Returns:
211: (12)                   The available extensions in a mapping {name: metadata}
212: (12)                   The results last page; None if the manager does not support
pagination
213: (8)                """
214: (8)                matches = await self.__get_all_extensions()
215: (8)                extensions = {}
216: (8)                counter = -1
217: (8)                min_index = (page - 1) * per_page
218: (8)                max_index = page * per_page
219: (8)                for name, group in groupby(filter(lambda m: query in m[0], matches),
lambda e: e[0]):
220: (12)                   counter += 1
221: (12)                   if counter < min_index or counter >= max_index:
222: (16)                       continue
223: (12)                   _, latest_version = list(group)[-1]
224: (12)                   data = await self._fetch_package_metadata(name, latest_version,
self.base_url)
225: (12)                   normalized_name = self._normalize_name(name)
226: (12)                   package_urls = data.get("project_urls") or {}
227: (12)                   source_url = package_urls.get("Source Code")
228: (12)                   homepage_url = data.get("home_page") or
package_urls.get("Homepage")
229: (12)                   documentation_url = data.get("docs_url") or
package_urls.get("Documentation")
230: (12)                   bug_tracker_url = data.get("bugtrack_url") or
```

```
package_urls.get("Bug Tracker")
231: (12)                       best_guess_home_url = (
232: (16)                           homepage_url
233: (16)                           or data.get("project_url")
234: (16)                           or data.get("package_url")
235: (16)                           or documentation_url
236: (16)                           or source_url
237: (16)                           or bug_tracker_url
238: (12)                       )
239: (12)                       extensions[normalized_name] = ExtensionPackage(
240: (16)                           name=normalized_name,
241: (16)                           description=data.get("summary"),
242: (16)                           homepage_url=best_guess_home_url,
243: (16)                           author=data.get("author"),
244: (16)                           license=data.get("license"),
245: (16)
latest_version=ExtensionManager.get_semver_version(latest_version),
246: (16)                           pkg_type="prebuilt",
247: (16)                           bug_tracker_url=bug_tracker_url,
248: (16)                           documentation_url=documentation_url,
249: (16)                           package_manager_url=data.get("package_url"),
250: (16)                           repository_url=source_url,
251: (12)                       )
252: (8)                    return extensions, math.ceil((counter + 1) / per_page)
253: (4)                async def __get_all_extensions(self) -> List[Tuple[str, str]]:
254: (8)                    if self.__all_packages_cache is None or datetime.now(
255: (12)                       tz=timezone.utc
256: (8)                    ) > self.__last_all_packages_request_time +
timedelta(seconds=self.cache_timeout):
257: (12)                       self.log.debug("Requesting PyPI.org RPC API for prebuilt
JupyterLab extensions.")
258: (12)                       self.__all_packages_cache = await self.__throttleRequest(
259: (16)                           True,
260: (16)                           self._rpc_client.browse,
261: (16)                           ["Framework :: Jupyter :: JupyterLab :: Extensions ::
Prebuilt"],
262: (12)                       )
263: (12)                       self.__last_all_packages_request_time =
datetime.now(tz=timezone.utc)
264: (8)                    return self.__all_packages_cache
265: (4)                async def install(self, name: str, version: Optional[str] = None) ->
ActionResult:  # noqa
266: (8)                    """Install the required extension.
267: (8)                    Note:
268: (12)                       If the user must be notified with a message (like asking to
restart the
269: (12)                       server), the result should be
270: (12)                       {"status": "warning", "message": "<explanation for the user>"}
271: (8)                    Args:
272: (12)                       name: The extension name
273: (12)                       version: The version to install; default None (i.e. the latest
possible)
274: (8)                    Returns:
275: (12)                       The action result
276: (8)                    """
277: (8)                    current_loop = tornado.ioloop.IOLoop.current()
278: (8)                    with tempfile.TemporaryDirectory() as ve_dir,
tempfile.NamedTemporaryFile(
279: (12)                       mode="w+", dir=ve_dir, delete=False
280: (8)                    ) as fconstraint:
281: (12)                       fconstraint.write(f"jupyterlab=={__version__}")
282: (12)                       fconstraint.flush()
283: (12)                       cmdline = [
284: (16)                           sys.executable,
285: (16)                           "-m",
286: (16)                           "pip",
287: (16)                           "install",
288: (16)                           "--no-input",
289: (16)                           "--quiet",
290: (16)                               "--progress-bar",
291: (16)                               "off",
292: (16)                               "--constraint",
293: (16)                               fconstraint.name,
294: (12)                           ]
295: (12)                           if version is not None:
296: (16)                               cmdline.append(f"{name}=={version}")
297: (12)                           else:
298: (16)                               cmdline.append(name)
299: (12)                           pkg_action = {}
300: (12)                           try:
301: (16)                               tmp_cmd = cmdline.copy()
302: (16)                               tmp_cmd.insert(-1, "--dry-run")
303: (16)                               tmp_cmd.insert(-1, "--report")
304: (16)                               tmp_cmd.insert(-1, "-")
305: (16)                               result = await current_loop.run_in_executor(
306: (20)                                   None, partial(run, tmp_cmd, capture_output=True,
check=True)
307: (16)                               )
308: (16)                               action_info = json.loads(result.stdout.decode("utf-8"))
309: (16)                               pkg_action = next(
310: (20)                                   filter(
311: (24)                                       lambda p: p.get("metadata", {}).get("name") ==
name.replace("_", "-"),
312: (24)                                       action_info.get("install", []),
313: (20)                                   )
314: (16)                               )
315: (12)                           except CalledProcessError as e:
316: (16)                               self.log.debug(f"Fail to get installation report: {e.stderr}",
exc_info=e)
317: (12)                           except Exception as err:
318: (16)                               self.log.debug("Fail to get installation report.",
exc_info=err)
319: (12)                           else:
320: (16)                               self.log.debug(f"Actions to be executed by pip
{json.dumps(action_info)}.")
321: (12)                           self.log.debug(f"Executing '{' '.join(cmdline)}'")
322: (12)                           result = await current_loop.run_in_executor(
323: (16)                               None, partial(run, cmdline, capture_output=True)
324: (12)                           )
325: (12)                           self.log.debug(f"return code: {result.returncode}")
326: (12)                           self.log.debug(f"stdout: {result.stdout.decode('utf-8')}")
327: (12)                           error = result.stderr.decode("utf-8")
328: (12)                           if result.returncode == 0:
329: (16)                               self.log.debug(f"stderr: {error}")
330: (16)                               jlab_metadata = None
331: (16)                               try:
332: (20)                                   download_url: str = pkg_action.get("download_info",
{}).get("url")
333: (20)                                   if download_url is not None:
334: (24)                                       response = await self._httpx_client.get(download_url)
335: (24)                                       if response.status_code < 400:  # noqa PLR2004
336: (28)                                           if download_url.endswith(".whl"):
337: (32)                                               with ZipFile(io.BytesIO(response.content)) as
wheel:
338: (36)                                                   for name in filter(
339: (40)                                                       lambda f: Path(f).name ==
"package.json",
340: (40)                                                       wheel.namelist(),
341: (36)                                                   ):
342: (40)                                                       data = json.loads(wheel.read(name))
343: (40)                                                       jlab_metadata = data.get("jupyterlab")
344: (40)                                                       if jlab_metadata is not None:
345: (44)                                                           break
346: (28)                                           elif download_url.endswith("tar.gz"):
347: (32)                                               with TarFile(io.BytesIO(response.content)) as
sdist:
348: (36)                                                   for name in filter(
349: (40)                                                       lambda f: Path(f).name ==
"package.json",
(40)                                        sdist.getnames(),
(36)                                    ):
(40)                                        data = json.load(sdist.extractfile(sdist.getmember(name)))
(40)                                        jlab_metadata = data.get("jupyterlab")
(40)                                        if jlab_metadata is not None:
(44)                                            break
(24)                        else:
(28)                            self.log.debug(f"Failed to get '{download_url}'; {response!s}")
(16)                except Exception as e:
(20)                    self.log.debug("Fail to get package.json.", exc_info=e)
(16)                follow_ups = [
(20)                    "frontend",
(16)                ]
(16)                if jlab_metadata is not None:
(20)                    discovery = jlab_metadata.get("discovery", {})
(20)                    if "kernel" in discovery:
(24)                        follow_ups.append("kernel")
(20)                    if "server" in discovery:
(24)                        follow_ups.append("server")
(16)                return ActionResult(status="ok", needs_restart=follow_ups)
(12)            else:
(16)                self.log.error(f"Failed to installed {name}: code {result.returncode}\n{error}")
(16)                return ActionResult(status="error", message=error)
(4)     async def uninstall(self, extension: str) -> ActionResult:
(8)         """Uninstall the required extension.
(8)         Note:
(12)            If the user must be notified with a message (like asking to restart the
(12)            server), the result should be
(12)            {"status": "warning", "message": "<explanation for the user>"}
(8)         Args:
(12)            extension: The extension name
(8)         Returns:
(12)            The action result
(8)         """
(8)         current_loop = tornado.ioloop.IOLoop.current()
(8)         cmdline = [
(12)            sys.executable,
(12)            "-m",
(12)            "pip",
(12)            "uninstall",
(12)            "--yes",
(12)            "--no-input",
(12)            extension,
(8)         ]
(8)         jlab_metadata = None
(8)         try:
(12)            tmp_cmd = cmdline.copy()
(12)            tmp_cmd.remove("--yes")
(12)            result = await current_loop.run_in_executor(
(16)                None, partial(run, tmp_cmd, capture_output=True)
(12)            )
(12)            lines = filter(
(16)                lambda line: line.endswith("package.json"),
(16)                map(lambda line: line.strip(), result.stdout.decode("utf-8").splitlines()),  # noqa
(12)            )
(12)            for filepath in filter(
(16)                lambda f: f.name == "package.json",
(16)                map(Path, lines),
(12)            ):
(16)                data = json.loads(filepath.read_bytes())
(16)                jlab_metadata = data.get("jupyterlab")
(16)                if jlab_metadata is not None:
(20)                    break
```

```
(8)                except Exception as e:
(12)                   self.log.debug("Fail to list files to be uninstalled.", exc_info=e)
(8)                self.log.debug(f"Executing '{' '.join(cmdline)}'")
(8)                result = await current_loop.run_in_executor(
(12)                   None, partial(run, cmdline, capture_output=True)
(8)                )
(8)                self.log.debug(f"return code: {result.returncode}")
(8)                self.log.debug(f"stdout: {result.stdout.decode('utf-8')}")
(8)                error = result.stderr.decode("utf-8")
(8)                if result.returncode == 0:
(12)                   self.log.debug(f"stderr: {error}")
(12)                   follow_ups = [
(16)                       "frontend",
(12)                   ]
(12)                   if jlab_metadata is not None:
(16)                       discovery = jlab_metadata.get("discovery", {})
(16)                       if "kernel" in discovery:
(20)                           follow_ups.append("kernel")
(16)                       if "server" in discovery:
(20)                           follow_ups.append("server")
(12)                   return ActionResult(status="ok", needs_restart=follow_ups)
(8)                else:
(12)                   self.log.error(f"Failed to installed {extension}: code {result.returncode}\n{error}")
(12)                   return ActionResult(status="error", message=error)
(4)            def _normalize_name(self, name: str) -> str:
(8)                """Normalize extension name.
(8)                Remove `@` from npm scope and replace `/` and `_` by `-`.
(8)                Args:
(12)                   name: Extension name
(8)                Returns:
(12)                   Normalized name
(8)                """
(8)                return name.replace("@", "").replace("/", "-").replace("_", "-")

----------------------------------------

File 20 - readonly.py:

(0)            """Extension manager without installation capabilities."""
(0)            import sys
(0)            from typing import Dict, Optional, Tuple
(0)            from jupyterlab_server.translation_utils import translator
(0)            from .manager import ActionResult, ExtensionManager, ExtensionManagerMetadata, ExtensionPackage
(0)            class ReadOnlyExtensionManager(ExtensionManager):
(4)                """Extension manager without installation capabilities."""
(4)                @property
(4)                def metadata(self) -> ExtensionManagerMetadata:
(8)                    """Extension manager metadata."""
(8)                    return ExtensionManagerMetadata("read-only", install_path=sys.prefix)
(4)                async def get_latest_version(self, pkg: str) -> Optional[str]:
(8)                    """Return the latest available version for a given extension.
(8)                    Args:
(12)                       pkg: The extension to search for
(8)                    Returns:
(12)                       The latest available version
(8)                    """
(8)                    return None
(4)                async def list_packages(
(8)                    self, query: str, page: int, per_page: int
(4)                ) -> Tuple[Dict[str, ExtensionPackage], Optional[int]]:
(8)                    """List the available extensions.
(8)                    Args:
(12)                       query: The search extension query
(12)                       page: The result page
(12)                       per_page: The number of results per page
(8)                    Returns:
```

```
29: (12)                        The available extensions in a mapping {name: metadata}
30: (12)                        The results last page; None if the manager does not support
pagination
31: (8)                     """
32: (8)                     return {}, None
33: (4)                 async def install(self, extension: str, version: Optional[str] = None) ->
ActionResult:
34: (8)                     """Install the required extension.
35: (8)                     Note:
36: (12)                        If the user must be notified with a message (like asking to
restart the
37: (12)                        server), the result should be
38: (12)                        {"status": "warning", "message": "<explanation for the user>"}
39: (8)                     Args:
40: (12)                        extension: The extension name
41: (12)                        version: The version to install; default None (i.e. the latest
possible)
42: (8)                     Returns:
43: (12)                        The action result
44: (8)                     """
45: (8)                     trans = translator.load("jupyterlab")
46: (8)                     return ActionResult(
47: (12)                        status="error", message=trans.gettext("Extension installation not
supported.")
48: (8)                     )
49: (4)                 async def uninstall(self, extension: str) -> ActionResult:
50: (8)                     """Uninstall the required extension.
51: (8)                     Note:
52: (12)                        If the user must be notified with a message (like asking to
restart the
53: (12)                        server), the result should be
54: (12)                        {"status": "warning", "message": "<explanation for the user>"}
55: (8)                     Args:
56: (12)                        extension: The extension name
57: (8)                     Returns:
58: (12)                        The action result
59: (8)                     """
60: (8)                     trans = translator.load("jupyterlab")
61: (8)                     return ActionResult(
62: (12)                        status="error", message=trans.gettext("Extension removal not
supported.")
63: (8)                     )

----------------------------------------

File 21 - __init__.py:

1: (0)              """Extension manager for JupyterLab."""
2: (0)              import sys
3: (0)              from typing import Optional
4: (0)              from traitlets.config import Configurable
5: (0)              from .manager import ActionResult, ExtensionManager, ExtensionPackage  # noqa:
F401
6: (0)              from .pypi import PyPIExtensionManager
7: (0)              from .readonly import ReadOnlyExtensionManager
8: (0)              if sys.version_info < (3, 10):
9: (4)                  from importlib_metadata import entry_points
10: (0)             else:
11: (4)                 from importlib.metadata import entry_points
12: (0)             MANAGERS = {}
13: (0)             for entry in entry_points(group="jupyterlab.extension_manager_v1"):
14: (4)                 MANAGERS[entry.name] = entry
15: (0)             def get_readonly_manager(
16: (4)                 app_options: Optional[dict] = None,
17: (4)                 ext_options: Optional[dict] = None,
18: (4)                 parent: Optional[Configurable] = None,
19: (0)             ) -> ExtensionManager:
20: (4)                 """Read-Only Extension Manager factory"""
21: (4)                 return ReadOnlyExtensionManager(app_options, ext_options, parent)
```

```
22: (0)             def get_pypi_manager(
23: (4)                 app_options: Optional[dict] = None,
24: (4)                 ext_options: Optional[dict] = None,
25: (4)                 parent: Optional[Configurable] = None,
26: (0)             ) -> ExtensionManager:
27: (4)                 """PyPi Extension Manager factory"""
28: (4)                 return PyPIExtensionManager(app_options, ext_options, parent)

----------------------------------------

File 22 - __init__.py:

1: (0)              import getpass
2: (0)              import os
3: (0)              from pathlib import Path
4: (0)              from tempfile import mkdtemp
5: (0)              def configure_jupyter_server(c):
6: (4)                  """Helper to configure the Jupyter Server for integration testing
7: (4)                  with Galata.
8: (4)                  By default the tests will be executed in the OS temporary folder. You
9: (4)                  can override that folder by setting the environment variable
``JUPYTERLAB_GALATA_ROOT_DIR``.
10: (4)                 .. warning::
11: (8)                     Never use this configuration in production as it will remove all
security protections.
12: (4)                 """
13: (4)                 if getpass.getuser() == "jovyan":
14: (8)                     c.ServerApp.ip = "0.0.0.0"  # noqa S104
15: (4)                 c.ServerApp.port = 8888
16: (4)                 c.ServerApp.port_retries = 0
17: (4)                 c.ServerApp.open_browser = False
18: (4)                 c.LabServerApp.extra_labextensions_path = str(Path(__file__).parent)
19: (4)                 c.LabApp.workspaces_dir = mkdtemp(prefix="galata-workspaces-")
20: (4)                 c.ServerApp.root_dir = os.environ.get(
21: (8)                     "JUPYTERLAB_GALATA_ROOT_DIR", mkdtemp(prefix="galata-test-")
22: (4)                 )
23: (4)                 c.IdentityProvider.token = ""
24: (4)                 c.ServerApp.password = ""
25: (4)                 c.ServerApp.disable_check_xsrf = True
26: (4)                 c.LabApp.expose_app_in_browser = True

----------------------------------------

File 23 - announcements.py:

1: (0)              """Announcements handler for JupyterLab."""
2: (0)              import abc
3: (0)              import hashlib
4: (0)              import json
5: (0)              import xml.etree.ElementTree as ET  # noqa
6: (0)              from dataclasses import asdict, dataclass, field
7: (0)              from datetime import datetime, timezone
8: (0)              from typing import Awaitable, Optional, Tuple, Union
9: (0)              from jupyter_server.base.handlers import APIHandler
10: (0)             from jupyterlab_server.translation_utils import translator
11: (0)             from packaging.version import parse
12: (0)             from tornado import httpclient, web
13: (0)             from jupyterlab._version import __version__
14: (0)             ISO8601_FORMAT = "%Y-%m-%dT%H:%M:%S%z"
15: (0)             JUPYTERLAB_LAST_RELEASE_URL = "https://pypi.org/pypi/jupyterlab/json"
16: (0)             JUPYTERLAB_RELEASE_URL =
"https://github.com/jupyterlab/jupyterlab/releases/tag/v"
17: (0)             def format_datetime(dt_str: str):
18: (4)                 return datetime.fromisoformat(dt_str).timestamp() * 1000
19: (0)             @dataclass(frozen=True)
20: (0)             class Notification:
21: (4)                 """Notification
22: (4)                 Attributes:
23: (8)                     createdAt: Creation date
```

```
24: (8)                         message: Notification message
25: (8)                         modifiedAt: Modification date
26: (8)                         type: Notification type — ["default", "error", "info", "success",
"warning"]
27: (8)                         link: Notification link button as a tuple (label, URL)
28: (8)                         options: Notification options
29: (4)                     """
30: (4)                     createdAt: float  # noqa
31: (4)                     message: str
32: (4)                     modifiedAt: float  # noqa
33: (4)                     type: str = "default"
34: (4)                     link: Tuple[str, str] = field(default_factory=tuple)
35: (4)                     options: dict = field(default_factory=dict)
36: (0)                 class CheckForUpdateABC(abc.ABC):
37: (4)                     """Abstract class to check for update.
38: (4)                     Args:
39: (8)                         version: Current JupyterLab version
40: (4)                     Attributes:
41: (8)                         version - str: Current JupyterLab version
42: (8)                         logger - logging.Logger: Server logger
43: (4)                     """
44: (4)                     def __init__(self, version: str) -> None:
45: (8)                         self.version = version
46: (4)                     @abc.abstractmethod
47: (4)                     async def __call__(self) -> Awaitable[Union[None, str, Tuple[str,
Tuple[str, str]]]]:
48: (8)                         """Get the notification message if a new version is available.
49: (8)                         Returns:
50: (12)                            None if there is not update.
51: (12)                            or the notification message
52: (12)                            or the notification message and a tuple(label, URL link) for the
user to get more information
53: (8)                         """
54: (8)                         msg = "CheckForUpdateABC.__call__ is not implemented"
55: (8)                         raise NotImplementedError(msg)
56: (0)                 class CheckForUpdate(CheckForUpdateABC):
57: (4)                     """Default class to check for update.
58: (4)                     Args:
59: (8)                         version: Current JupyterLab version
60: (4)                     Attributes:
61: (8)                         version - str: Current JupyterLab version
62: (8)                         logger - logging.Logger: Server logger
63: (4)                     """
64: (4)                     async def __call__(self) -> Awaitable[Tuple[str, Tuple[str, str]]]:
65: (8)                         """Get the notification message if a new version is available.
66: (8)                         Returns:
67: (12)                            None if there is no update.
68: (12)                            or the notification message
69: (12)                            or the notification message and a tuple(label, URL link) for the
user to get more information
70: (8)                         """
71: (8)                         http_client = httpclient.AsyncHTTPClient()
72: (8)                         try:
73: (12)                            response = await http_client.fetch(
74: (16)                                JUPYTERLAB_LAST_RELEASE_URL,
75: (16)                                headers={"Content-Type": "application/json"},
76: (12)                            )
77: (12)                            data = json.loads(response.body).get("info")
78: (12)                            last_version = data["version"]
79: (8)                         except Exception as e:
80: (12)                            self.logger.debug("Failed to get latest version", exc_info=e)
81: (12)                            return None
82: (8)                         else:
83: (12)                            if parse(self.version) < parse(last_version):
84: (16)                                trans = translator.load("jupyterlab")
85: (16)                                return (
86: (20)                                    trans.__(f"A newer version ({last_version}) of JupyterLab
is available."),
87: (20)                                    (trans.__("Open changelog"), f"{JUPYTERLAB_RELEASE_URL}
{last_version}"),
88: (16)                            )
89: (12)                        else:
90: (16)                            return None
91: (0)             class NeverCheckForUpdate(CheckForUpdateABC):
92: (4)                 """Check update version that does nothing.
93: (4)                 This is provided for administrators that want to
94: (4)                 turn off requesting external resources.
95: (4)                 Args:
96: (8)                     version: Current JupyterLab version
97: (4)                 Attributes:
98: (8)                     version - str: Current JupyterLab version
99: (8)                     logger - logging.Logger: Server logger
100: (4)                """
101: (4)                async def __call__(self) -> Awaitable[None]:
102: (8)                    """Get the notification message if a new version is available.
103: (8)                    Returns:
104: (12)                       None if there is no update.
105: (12)                       or the notification message
106: (12)                       or the notification message and a tuple(label, URL link) for the
user to get more information
107: (8)                    """
108: (8)                    return None
109: (0)            class CheckForUpdateHandler(APIHandler):
110: (4)                """Check for Updates API handler.
111: (4)                Args:
112: (8)                    update_check: The class checking for a new version
113: (4)                """
114: (4)                def initialize(
115: (8)                    self,
116: (8)                    update_checker: Optional[CheckForUpdate] = None,
117: (4)                ) -> None:
118: (8)                    super().initialize()
119: (8)                    self.update_checker = (
120: (12)                       NeverCheckForUpdate(__version__) if update_checker is None else
update_checker
121: (8)                    )
122: (8)                    self.update_checker.logger = self.log
123: (4)                @web.authenticated
124: (4)                async def get(self):
125: (8)                    """Check for updates.
126: (8)                    Response:
127: (12)                       {
128: (16)                           "notification": Optional[Notification]
129: (12)                       }
130: (8)                    """
131: (8)                    notification = None
132: (8)                    out = await self.update_checker()
133: (8)                    if out:
134: (12)                       message, link = (out, ()) if isinstance(out, str) else out
135: (12)                       now = datetime.now(tz=timezone.utc).timestamp() * 1000.0
136: (12)                       hash_ = hashlib.sha1(message.encode()).hexdigest()  # noqa: S324
137: (12)                       notification = Notification(
138: (16)                           message=message,
139: (16)                           createdAt=now,
140: (16)                           modifiedAt=now,
141: (16)                           type="info",
142: (16)                           link=link,
143: (16)                           options={"data": {"id": hash_, "tags": ["update"]}},
144: (12)                       )
145: (8)                    self.set_status(200)
146: (8)                    self.finish(
147: (12)                       json.dumps({"notification": None if notification is None else
asdict(notification)})
148: (8)                    )
149: (0)            class NewsHandler(APIHandler):
150: (4)                """News API handler.
151: (4)                Args:
152: (8)                    news_url: The Atom feed to fetch for news
```

```
153: (4)                """
154: (4)                def initialize(
155: (8)                    self,
156: (8)                    news_url: Optional[str] = None,
157: (4)                ) -> None:
158: (8)                    super().initialize()
159: (8)                    self.news_url = news_url
160: (4)                @web.authenticated
161: (4)                async def get(self):
162: (8)                    """Get the news.
163: (8)                    Response:
164: (12)                       {
165: (16)                           "news": List[Notification]
166: (12)                       }
167: (8)                    """
168: (8)                    news = []
169: (8)                    http_client = httpclient.AsyncHTTPClient()
170: (8)                    if self.news_url is not None:
171: (12)                       trans = translator.load("jupyterlab")
172: (12)                       xml_namespaces = {"atom": "http://www.w3.org/2005/Atom"}
173: (12)                       for key, spec in xml_namespaces.items():
174: (16)                           ET.register_namespace(key, spec)
175: (12)                       try:
176: (16)                           response = await http_client.fetch(
177: (20)                               self.news_url,
178: (20)                               headers={"Content-Type": "application/atom+xml"},
179: (16)                           )
180: (16)                           tree = ET.fromstring(response.body)  # noqa S314
181: (16)                           def build_entry(node):
182: (20)                               def get_xml_text(attr: str, default: Optional[str] = None)
-> str:
183: (24)                                   node_item = node.find(f"atom:{attr}", xml_namespaces)
184: (24)                                   if node_item is not None:
185: (28)                                       return node_item.text
186: (24)                                   elif default is not None:
187: (28)                                       return default
188: (24)                                   else:
189: (28)                                       error_m = (
190: (32)                                           f"atom feed entry does not contain a required
attribute: {attr}"
191: (28)                                       )
192: (28)                                       raise KeyError(error_m)
193: (20)                               entry_title = get_xml_text("title")
194: (20)                               entry_id = get_xml_text("id")
195: (20)                               entry_updated = get_xml_text("updated")
196: (20)                               entry_published = get_xml_text("published", entry_updated)
197: (20)                               entry_summary = get_xml_text("summary", default="")
198: (20)                               links = node.findall("atom:link", xml_namespaces)
199: (20)                               if len(links) > 1:
200: (24)                                   alternate = list(filter(lambda elem: elem.get("rel")
== "alternate", links))
201: (24)                                   link_node = alternate[0] if alternate else links[0]
202: (20)                               else:
203: (24)                                   link_node = links[0] if len(links) == 1 else None
204: (20)                               entry_link = link_node.get("href") if link_node is not
None else None
205: (20)                               message = (
206: (24)                                   "\n".join([entry_title, entry_summary]) if
entry_summary else entry_title
207: (20)                               )
208: (20)                               modified_at = format_datetime(entry_updated)
209: (20)                               created_at = format_datetime(entry_published)
210: (20)                               notification = Notification(
211: (24)                                   message=message,
212: (24)                                   createdAt=created_at,
213: (24)                                   modifiedAt=modified_at,
214: (24)                                   type="info",
215: (24)                                   link=None
216: (24)                                   if entry_link is None
217: (24)                                   else (
218: (28)                                       trans.__("Open full post"),
219: (28)                                       entry_link,
220: (24)                                   ),
221: (24)                                   options={
222: (28)                                       "data": {
223: (32)                                           "id": entry_id,
224: (32)                                           "tags": ["news"],
225: (28)                                       }
226: (24)                                   },
227: (20)                               )
228: (20)                               return notification
229: (16)                           entries = map(build_entry, tree.findall("atom:entry",
xml_namespaces))
230: (16)                           news.extend(entries)
231: (12)                       except Exception as e:
232: (16)                           self.log.debug(
233: (20)                               f"Failed to get announcements from Atom feed:
{self.news_url}",
234: (20)                               exc_info=e,
235: (16)                           )
236: (8)                    self.set_status(200)
237: (8)                    self.finish(json.dumps({"news": list(map(asdict, news))}))
238: (0)            news_handler_path = r"/lab/api/news"
239: (0)            check_update_handler_path = r"/lab/api/update"

----------------------------------------

File 24 - build_handler.py:

1: (0)              """Tornado handlers for frontend config storage."""
2: (0)              import json
3: (0)              from concurrent.futures import ThreadPoolExecutor
4: (0)              from threading import Event
5: (0)              from jupyter_server.base.handlers import APIHandler
6: (0)              from jupyter_server.extension.handler import ExtensionHandlerMixin
7: (0)              from tornado import gen, web
8: (0)              from tornado.concurrent import run_on_executor
9: (0)              from jupyterlab.commands import AppOptions, _ensure_options, build,
build_check, clean
10: (0)             class Builder:
11: (4)                 building = False
12: (4)                 executor = ThreadPoolExecutor(max_workers=5)
13: (4)                 canceled = False
14: (4)                 _canceling = False
15: (4)                 _kill_event = None
16: (4)                 _future = None
17: (4)                 def __init__(self, core_mode, app_options=None):
18: (8)                     app_options = _ensure_options(app_options)
19: (8)                     self.log = app_options.logger
20: (8)                     self.core_mode = core_mode
21: (8)                     self.app_dir = app_options.app_dir
22: (8)                     self.core_config = app_options.core_config
23: (8)                     self.labextensions_path = app_options.labextensions_path
24: (4)                 @gen.coroutine
25: (4)                 def get_status(self):
26: (8)                     if self.core_mode:
27: (12)                        raise gen.Return({"status": "stable", "message": ""})
28: (8)                     if self.building:
29: (12)                        raise gen.Return({"status": "building", "message": ""})
30: (8)                     try:
31: (12)                        messages = yield self._run_build_check(
32: (16)                            self.app_dir, self.log, self.core_config,
self.labextensions_path
33: (12)                        )
34: (12)                        status = "needed" if messages else "stable"
35: (12)                        if messages:
36: (16)                            self.log.warning("Build recommended")
37: (16)                            [self.log.warning(m) for m in messages]
```

```
38: (12)                        else:
39: (16)                            self.log.info("Build is up to date")
40: (8)                     except ValueError:
41: (12)                        self.log.warning("Could not determine jupyterlab build status
without nodejs")
42: (12)                        status = "stable"
43: (12)                        messages = []
44: (8)                     raise gen.Return({"status": status, "message": "\n".join(messages)})
45: (4)                 @gen.coroutine
46: (4)                 def build(self):
47: (8)                     if self._canceling:
48: (12)                        msg = "Cancel in progress"
49: (12)                        raise ValueError(msg)
50: (8)                     if not self.building:
51: (12)                        self.canceled = False
52: (12)                        self._future = future = gen.Future()
53: (12)                        self.building = True
54: (12)                        self._kill_event = evt = Event()
55: (12)                        try:
56: (16)                            yield self._run_build(
57: (20)                                self.app_dir, self.log, evt, self.core_config,
self.labextensions_path
58: (16)                            )
59: (16)                            future.set_result(True)
60: (12)                        except Exception as e:
61: (16)                            if str(e) == "Aborted":
62: (20)                                future.set_result(False)
63: (16)                            else:
64: (20)                                future.set_exception(e)
65: (12)                        finally:
66: (16)                            self.building = False
67: (8)                     try:
68: (12)                        yield self._future
69: (8)                     except Exception as e:
70: (12)                        raise e
71: (4)                 @gen.coroutine
72: (4)                 def cancel(self):
73: (8)                     if not self.building:
74: (12)                        msg = "No current build"
75: (12)                        raise ValueError(msg)
76: (8)                     self._canceling = True
77: (8)                     yield self._future
78: (8)                     self._canceling = False
79: (8)                     self.canceled = True
80: (4)                 @run_on_executor
81: (4)                 def _run_build_check(self, app_dir, logger, core_config,
labextensions_path):
82: (8)                     return build_check(
83: (12)                        app_options=AppOptions(
84: (16)                            app_dir=app_dir,
85: (16)                            logger=logger,
86: (16)                            core_config=core_config,
87: (16)                            labextensions_path=labextensions_path,
88: (12)                        )
89: (8)                     )
90: (4)                 @run_on_executor
91: (4)                 def _run_build(self, app_dir, logger, kill_event, core_config,
labextensions_path):
92: (8)                     app_options = AppOptions(
93: (12)                        app_dir=app_dir,
94: (12)                        logger=logger,
95: (12)                        kill_event=kill_event,
96: (12)                        core_config=core_config,
97: (12)                        labextensions_path=labextensions_path,
98: (8)                     )
99: (8)                     try:
100: (12)                       return build(app_options=app_options)
101: (8)                    except Exception:
102: (12)                       if self._kill_event.is_set():
```

```
(16)                        return
(12)                    self.log.warning("Build failed, running a clean and rebuild")
(12)                    clean(app_options=app_options)
(12)                    return build(app_options=app_options)
(0)         class BuildHandler(ExtensionHandlerMixin, APIHandler):
(4)             def initialize(self, builder=None, name=None):
(8)                 super().initialize(name=name)
(8)                 self.builder = builder
(4)             @web.authenticated
(4)             @gen.coroutine
(4)             def get(self):
(8)                 data = yield self.builder.get_status()
(8)                 self.finish(json.dumps(data))
(4)             @web.authenticated
(4)             @gen.coroutine
(4)             def delete(self):
(8)                 self.log.warning("Canceling build")
(8)                 try:
(12)                    yield self.builder.cancel()
(8)                 except Exception as e:
(12)                    raise web.HTTPError(500, str(e)) from None
(8)                 self.set_status(204)
(4)             @web.authenticated
(4)             @gen.coroutine
(4)             def post(self):
(8)                 self.log.debug("Starting build")
(8)                 try:
(12)                    yield self.builder.build()
(8)                 except Exception as e:
(12)                    raise web.HTTPError(500, str(e)) from None
(8)                 if self.builder.canceled:
(12)                    raise web.HTTPError(400, "Build canceled")
(8)                 self.log.debug("Build succeeded")
(8)                 self.set_status(200)
(0)         build_path = r"/lab/api/build"
```

File 25 - error_handler.py:

```
(0)         """An error handler for JupyterLab."""
(0)         from jupyter_server.base.handlers import JupyterHandler
(0)         from jupyter_server.extension.handler import ExtensionHandlerMixin
(0)         from tornado import web
(0)         TEMPLATE = """
(0)         <!DOCTYPE HTML>
(0)         <html>
(0)         <head>
(4)             <meta charset="utf-8">
(4)             <title>JupyterLab Error</title>
(0)         </head>
(0)         <body>
(0)         <h1>JupyterLab Error<h1>
(0)         %s
(0)         </body>
(0)         """
(0)         class ErrorHandler(ExtensionHandlerMixin, JupyterHandler):
(4)             def initialize(self, messages=None, name=None):
(8)                 super().initialize(name=name)
(8)                 self.messages = messages
(4)             @web.authenticated
(4)             @web.removeslash
(4)             def get(self):
(8)                 msgs = ["<h2>%s</h2>" % msg for msg in self.messages]
(8)                 self.write(TEMPLATE % "\n".join(msgs))
```

File 26 - extension_manager_handler.py:

```
(0)             """Tornado handlers for extension management."""
(0)             import dataclasses
(0)             import json
(0)             from urllib.parse import urlencode, urlunparse
(0)             from jupyter_server.base.handlers import APIHandler
(0)             from tornado import web
(0)             from jupyterlab.extensions.manager import ExtensionManager
(0)             class ExtensionHandler(APIHandler):
(4)                 def initialize(self, manager: ExtensionManager):
(8)                     super().initialize()
(8)                     self.manager = manager
(4)                 @web.authenticated
(4)                 async def get(self):
(8)                     """GET query returns info on extensions
(8)                     Query arguments:
(12)                        refresh: [optional] Force refreshing the list of extensions - ["0", "1"]; default 0
(12)                        query: [optional] Query to search for extensions - default None (i.e. returns installed extensions)
(12)                        page: [optional] Result page - default 1 (min. 1)
(12)                        per_page: [optional] Number of results per page - default 30 (max. 100)
(8)                     """
(8)                     query = self.get_argument("query", None)
(8)                     page = max(1, int(self.get_argument("page", "1")))
(8)                     per_page = min(100, int(self.get_argument("per_page", "30")))
(8)                     if self.get_argument("refresh", "0") == "1":
(12)                        await self.manager.refresh(query, page, per_page)
(8)                     extensions, last_page = await self.manager.list_extensions(query, page, per_page)
(8)                     self.set_status(200)
(8)                     if last_page is not None:
(12)                        links = []
(12)                        query_args = {"page": last_page, "per_page": per_page}
(12)                        if query is not None:
(16)                            query_args["query"] = query
(12)                        last = urlunparse(
(16)                            (
(20)                                self.request.protocol,
(20)                                self.request.host,
(20)                                self.request.path,
(20)                                "",
(20)                                urlencode(query_args, doseq=True),
(20)                                "",
(16)                            )
(12)                        )
(12)                        links.append(f'<{last}>; rel="last"')
(12)                        if page > 1:
(16)                            query_args["page"] = max(1, page - 1)
(16)                            prev = urlunparse(
(20)                                (
(24)                                    self.request.protocol,
(24)                                    self.request.host,
(24)                                    self.request.path,
(24)                                    "",
(24)                                    urlencode(query_args, doseq=True),
(24)                                    "",
(20)                                )
(16)                            )
(16)                            links.append(f'<{prev}>; rel="prev"')
(12)                        if page < last_page:
(16)                            query_args["page"] = min(page + 1, last_page)
(16)                            next_ = urlunparse(
(20)                                (
(24)                                    self.request.protocol,
(24)                                    self.request.host,
(24)                                    self.request.path,
(24)                                    "",
65: (24)                                    urlencode(query_args, doseq=True),
66: (24)                                    "",
67: (20)                                )
68: (16)                            )
69: (16)                            links.append(f'<{next_}>; rel="next"')
70: (12)                        query_args["page"] = 1
71: (12)                        first = urlunparse(
72: (16)                            (
73: (20)                                self.request.protocol,
74: (20)                                self.request.host,
75: (20)                                self.request.path,
76: (20)                                "",
77: (20)                                urlencode(query_args, doseq=True),
78: (20)                                "",
79: (16)                            )
80: (12)                        )
81: (12)                        links.append(f'<{first}>; rel="first"')
82: (12)                        self.set_header("Link", ", ".join(links))
83: (8)                     self.finish(json.dumps(list(map(dataclasses.asdict, extensions))))
84: (4)                 @web.authenticated
85: (4)                 async def post(self):
86: (8)                     """POST query performs an action on a specific extension
87: (8)                     Body arguments:
88: (12)                        {
89: (16)                            "cmd": Action to perform - ["install", "uninstall", "enable",
"disable"]
90: (16)                            "extension_name": Extension name
91: (16)                            "extension_version": [optional] Extension version (used only
for install action)
92: (12)                        }
93: (8)                     """
94: (8)                     data = self.get_json_body()
95: (8)                     cmd = data["cmd"]
96: (8)                     name = data["extension_name"]
97: (8)                     version = data.get("extension_version")
98: (8)                     if cmd not in ("install", "uninstall", "enable", "disable") or not
name:
99: (12)                        raise web.HTTPError(
100: (16)                           422,
101: (16)                           f"Could not process instruction {cmd!r} with extension name
{name!r}",
102: (12)                       )
103: (8)                    ret_value = None
104: (8)                    try:
105: (12)                       if cmd == "install":
106: (16)                           ret_value = await self.manager.install(name, version)
107: (12)                       elif cmd == "uninstall":
108: (16)                           ret_value = await self.manager.uninstall(name)
109: (12)                       elif cmd == "enable":
110: (16)                           ret_value = await self.manager.enable(name)
111: (12)                       elif cmd == "disable":
112: (16)                           ret_value = await self.manager.disable(name)
113: (8)                    except Exception as e:
114: (12)                       raise web.HTTPError(500, str(e)) from e
115: (8)                    if ret_value.status == "error":
116: (12)                       self.set_status(500)
117: (8)                    else:
118: (12)                       self.set_status(201)
119: (8)                    self.finish(json.dumps(dataclasses.asdict(ret_value)))
120: (0)            extensions_handler_path = r"/lab/api/extensions"

----------------------------------------

File 27 - plugin_manager_handler.py:

1: (0)              """Tornado handlers for plugin management."""
2: (0)              import dataclasses
3: (0)              import json
4: (0)              from jupyter_server.base.handlers import APIHandler
```

```
5: (0)              from tornado import web
6: (0)              from jupyterlab.extensions.manager import PluginManager
7: (0)              class PluginHandler(APIHandler):
8: (4)                  def initialize(self, manager: PluginManager):
9: (8)                      super().initialize()
10: (8)                     self.manager = manager
11: (4)                 @web.authenticated
12: (4)                 async def get(self):
13: (8)                     """GET query returns info on plugins locks"""
14: (8)                     locks = await self.manager.plugin_locks()
15: (8)                     self.set_status(200)
16: (8)                     self.finish(json.dumps(locks))
17: (4)                 @web.authenticated
18: (4)                 async def post(self):
19: (8)                     """POST query performs an action on a specific plugin
20: (8)                     Body arguments:
21: (12)                        {
22: (16)                            "cmd": Action to perform - ["enable", "disable"]
23: (16)                            "plugin_name": Plugin name
24: (12)                        }
25: (8)                     """
26: (8)                     data = self.get_json_body()
27: (8)                     cmd = data["cmd"]
28: (8)                     name = data["plugin_name"]
29: (8)                     if cmd not in ("enable", "disable") or not name:
30: (12)                        raise web.HTTPError(
31: (16)                            422,
32: (16)                            f"Could not process instruction {cmd!r} with plugin name
{name!r}",
33: (12)                        )
34: (8)                     ret_value = None
35: (8)                     try:
36: (12)                        if cmd == "enable":
37: (16)                            ret_value = await self.manager.enable(name)
38: (12)                        elif cmd == "disable":
39: (16)                            ret_value = await self.manager.disable(name)
40: (8)                     except Exception as e:
41: (12)                        raise web.HTTPError(500, str(e)) from e
42: (8)                     if ret_value.status == "error":
43: (12)                        self.set_status(500)
44: (8)                     else:
45: (12)                        self.set_status(201)
46: (8)                     self.finish(json.dumps(dataclasses.asdict(ret_value)))
47: (0)              plugins_handler_path = r"/lab/api/plugins"

----------------------------------------

File 28 - __init__.py:

1: (0)

----------------------------------------

File 29 - conftest.py:

1: (0)              import pytest
2: (0)              from jupyterlab import __version__
3: (0)              from jupyterlab.handlers.announcements import (
4: (4)                  CheckForUpdate,
5: (4)                  CheckForUpdateHandler,
6: (4)                  NewsHandler,
7: (4)                  check_update_handler_path,
8: (4)                  news_handler_path,
9: (0)              )
10: (0)             @pytest.fixture
11: (0)             def labserverapp(jp_serverapp, make_labserver_extension_app):
12: (4)                 app = make_labserver_extension_app()
13: (4)                 app._link_jupyter_server_extension(jp_serverapp)
14: (4)                 app.handlers.extend(
15: (8)                     [
16: (12)                        (
17: (16)                            r"/custom/(.*)(?<!\.js)$",
18: (16)                            jp_serverapp.web_app.settings["static_handler_class"],
19: (16)                            {
20: (20)                                "path":
jp_serverapp.web_app.settings["static_custom_path"],
21: (20)                                "no_cache_paths": ["/"],  # don't cache anything in custom
22: (16)                            },
23: (12)                        ),
24: (12)                        (
25: (16)                            check_update_handler_path,
26: (16)                            CheckForUpdateHandler,
27: (16)                            {
28: (20)                                "update_checker": CheckForUpdate(__version__),
29: (16)                            },
30: (12)                        ),
31: (12)                        (
32: (16)                            news_handler_path,
33: (16)                            NewsHandler,
34: (16)                            {
35: (20)                                "news_url": "https://dummy.io/feed.xml",
36: (16)                            },
37: (12)                        ),
38: (8)                     ]
39: (4)                 )
40: (4)                 app.initialize()
41: (4)                 return app
```

File 30 - echo_kernel.py:

```
1: (0)              import logging
2: (0)              from ipykernel.kernelapp import IPKernelApp
3: (0)              from ipykernel.kernelbase import Kernel
4: (0)              class EchoKernel(Kernel):
5: (4)                  implementation = "Echo"
6: (4)                  implementation_version = "1.0"
7: (4)                  language = "echo"
8: (4)                  language_version = "0.1"
9: (4)                  language_info = {
10: (8)                     "name": "echo",
11: (8)                     "mimetype": "text/plain",
12: (8)                     "file_extension": ".txt",
13: (4)                 }
14: (4)                 banner = "Echo kernel - as useful as a parrot"
15: (4)                 def do_execute(
16: (8)                     self, code, silent, store_history=True, user_expressions=None,
allow_stdin=False
17: (4)                 ):
18: (8)                     if not silent:
19: (12)                        stream_content = {"name": "stdout", "text": code}
20: (12)                        self.send_response(self.iopub_socket, "stream", stream_content)
21: (12)                        if allow_stdin and code and code.find("input(") != -1:
22: (16)                            self._input_request(
23: (20)                                "Echo Prompt",
24: (20)                                self._parent_ident["shell"],
25: (20)                                self.get_parent(channel="shell"),
26: (20)                                password=False,
27: (16)                            )
28: (8)                     return {
29: (12)                        "status": "ok",
30: (12)                        "execution_count": self.execution_count,
31: (12)                        "payload": [],
32: (12)                        "user_expressions": {},
33: (8)                     }
34: (0)             class EchoKernelApp(IPKernelApp):
35: (4)                 kernel_class = EchoKernel
```

```
36: (0)             if __name__ == "__main__":
37: (4)                 logging.disable(logging.ERROR)
38: (4)                 EchoKernelApp.launch_instance()

----------------------------------------

File 31 - test_announcements.py:

1: (0)              import hashlib
2: (0)              import json
3: (0)              from unittest.mock import patch
4: (0)              from . import fake_client_factory
5: (0)              FAKE_ATOM_FEED = b"""<?xml version="1.0" encoding="utf-8"?><feed
xmlns="http://www.w3.org/2005/Atom" ><generator uri="https://jekyllrb.com/"
version="3.9.2">Jekyll</generator><link href="https://jupyterlab.github.io/assets/feed.xml"
rel="self" type="application/atom+xml" /><link href="https://jupyterlab.github.io/assets/"
rel="alternate" type="text/html" /><updated>2022-11-02T15:14:50+00:00</updated>
<id>https://jupyterlab.github.io/assets/feed.xml</id><title type="html">JupyterLab News</title>
<subtitle>Subscribe to get news about JupyterLab.</subtitle><entry><title type="html">Thanks for
using JupyterLab</title><link
href="https://jupyterlab.github.io/assets/posts/2022/11/02/demo.html" rel="alternate"
type="text/html" title="Thanks for using JupyterLab" /><published>2022-11-
02T14:00:00+00:00</published><updated>2022-11-02T14:00:00+00:00</updated>
<id>https://jupyterlab.github.io/assets/posts/2022/11/02/demo</id><content type="html"
xml:base="https://jupyterlab.github.io/assets/posts/2022/11/02/demo.html">&lt;h1
id=&quot;welcome&quot;&gt;Welcome&lt;/h1&gt;
6: (0)              &lt;p&gt;Thanks a lot for your interest in JupyterLab.&lt;/p&gt;</content>
<author><name></name></author><category term="posts" /><summary type="html">Big thanks to you,
beloved JupyterLab user.</summary></entry></feed>"""
7: (0)              FAKE_JUPYTERLAB_PYPI_JSON = b"""{ "info": { "version": "1000.0.0" } }"""
8: (0)              @patch("tornado.httpclient.AsyncHTTPClient", new_callable=fake_client_factory)
9: (0)              async def test_NewsHandler_get_success(mock_client, labserverapp, jp_fetch):
10: (4)                 mock_client.body = FAKE_ATOM_FEED
11: (4)                 response = await jp_fetch("lab", "api", "news", method="GET")
12: (4)                 assert response.code == 200
13: (4)                 payload = json.loads(response.body)
14: (4)                 assert payload["news"] == [
15: (8)                     {
16: (12)                        "createdAt": 1667397600000.0,
17: (12)                        "message": "Thanks for using JupyterLab\nBig thanks to you,
beloved JupyterLab user.",
18: (12)                        "modifiedAt": 1667397600000.0,
19: (12)                        "type": "info",
20: (12)                        "link": [
21: (16)                            "Open full post",
22: (16)
"https://jupyterlab.github.io/assets/posts/2022/11/02/demo.html",
23: (12)                        ],
24: (12)                        "options": {
25: (16)                            "data": {
26: (20)                                "id":
"https://jupyterlab.github.io/assets/posts/2022/11/02/demo",
27: (20)                                "tags": ["news"],
28: (16)                            }
29: (12)                        },
30: (8)                     }
31: (4)                 ]
32: (0)             @patch("tornado.httpclient.AsyncHTTPClient", new_callable=fake_client_factory)
33: (0)             async def test_NewsHandler_get_failure(mock_client, labserverapp, jp_fetch):
34: (4)                 response = await jp_fetch("lab", "api", "news", method="GET")
35: (4)                 assert response.code == 200
36: (4)                 payload = json.loads(response.body)
37: (4)                 assert payload["news"] == []
38: (0)             @patch("tornado.httpclient.AsyncHTTPClient", new_callable=fake_client_factory)
39: (0)             async def test_CheckForUpdateHandler_get_pypi_success(mock_client,
labserverapp, jp_fetch):
40: (4)                 mock_client.body = FAKE_JUPYTERLAB_PYPI_JSON
41: (4)                 response = await jp_fetch("lab", "api", "update", method="GET")
42: (4)                 message = "A newer version (1000.0.0) of JupyterLab is available."
```

```
43: (4)                 assert response.code == 200
44: (4)                 payload = json.loads(response.body)
45: (4)                 assert payload["notification"]["message"] == message
46: (4)                 assert payload["notification"]["link"] == [
47: (8)                     "Open changelog",
48: (8)                     "https://github.com/jupyterlab/jupyterlab/releases/tag/v1000.0.0",
49: (4)                 ]
50: (4)                 assert payload["notification"]["options"] == {
51: (8)                     "data": {"id": hashlib.sha1(message.encode()).hexdigest(), "tags":
["update"]}  # noqa: S324
52: (4)                 }
53: (0)             @patch("tornado.httpclient.AsyncHTTPClient", new_callable=fake_client_factory)
54: (0)             async def test_CheckForUpdateHandler_get_failure(mock_client, labserverapp,
jp_fetch):
55: (4)                 response = await jp_fetch("lab", "api", "update", method="GET")
56: (4)                 assert response.code == 200
57: (4)                 payload = json.loads(response.body)
58: (4)                 assert payload["notification"] is None
59: (0)             FAKE_NO_SUMMARY_ATOM_FEED = b"""<?xml version='1.0' encoding='UTF-8'?><feed
xmlns="http://www.w3.org/2005/Atom" xml:lang="en">
<id>https://jupyterlab.github.io/assets/feed.xml</id><title>JupyterLab News</title><updated>2023-
05-02T19:01:33.669598+00:00</updated><author><name>John Doe</name><email>john@example.de</email>
</author><link href="https://jupyterlab.github.io/assets/feed.xml" rel="self"
type="application/atom+xml"/><link href="https://jupyterlab.github.io/assets/" rel="alternate"
type="text/html"/><generator uri="https://lkiesow.github.io/python-feedgen"
version="0.9.0">python-feedgen</generator><logo>http://ex.com/logo.jpg</logo><subtitle>Subscribe
to get news about JupyterLab.</subtitle><entry>
<id>https://jupyterlab.github.io/assets/posts/2022/11/02/demo</id><title>Thanks for using
JupyterLab</title><updated>2022-11-02T14:00:00+00:00</updated><link
href="https://jupyterlab.github.io/assets/posts/2022/11/02/demo.html" rel="alternate"
type="text/html" title="Thanks for using JupyterLab"/><published>2022-11-
02T14:00:00+00:00</published></entry></feed>"""
60: (0)             @patch("tornado.httpclient.AsyncHTTPClient", new_callable=fake_client_factory)
61: (0)             async def test_NewsHandler_get_missing_summary(mock_client, labserverapp,
jp_fetch):
62: (4)                 mock_client.body = FAKE_NO_SUMMARY_ATOM_FEED
63: (4)                 response = await jp_fetch("lab", "api", "news", method="GET")
64: (4)                 assert response.code == 200
65: (4)                 payload = json.loads(response.body)
66: (4)                 assert payload["news"] == [
67: (8)                     {
68: (12)                        "createdAt": 1667397600000.0,
69: (12)                        "message": "Thanks for using JupyterLab",
70: (12)                        "modifiedAt": 1667397600000.0,
71: (12)                        "type": "info",
72: (12)                        "link": [
73: (16)                            "Open full post",
74: (16)
"https://jupyterlab.github.io/assets/posts/2022/11/02/demo.html",
75: (12)                        ],
76: (12)                        "options": {
77: (16)                            "data": {
78: (20)                                "id":
"https://jupyterlab.github.io/assets/posts/2022/11/02/demo",
79: (20)                                "tags": ["news"],
80: (16)                            }
81: (12)                        },
82: (8)                     }
83: (4)                 ]
84: (0)             FAKE_MULTI_ENTRY_LINKS_ATOM_FEED = b"""<?xml version='1.0' encoding='UTF-8'?>
<feed xmlns="http://www.w3.org/2005/Atom" xml:lang="en">
<id>https://jupyterlab.github.io/assets/feed.xml</id><title>JupyterLab News</title><updated>2023-
05-02T19:59:44.332080+00:00</updated><author><name>John Doe</name><email>john@example.de</email>
</author><link href="https://jupyterlab.github.io/assets/feed.xml" rel="self"
type="application/atom+xml"/><link href="https://jupyterlab.github.io/assets/" rel="alternate"
type="text/html"/><generator uri="https://lkiesow.github.io/python-feedgen"
version="0.9.0">python-feedgen</generator><logo>http://ex.com/logo.jpg</logo><subtitle>Subscribe
to get news about JupyterLab.</subtitle><entry>
<id>https://jupyterlab.github.io/assets/posts/2022/11/02/demo</id><title>Thanks for using
```

```
JupyterLab</title><updated>2022-11-02T14:00:00+00:00</updated><link
href="https://jupyterlab.github.io/assets/posts/2022/11/02/demo_self.html" rel="self"
type="text/html" title="Thanks for using JupyterLab"/><link
href="https://jupyterlab.github.io/assets/posts/2022/11/02/demo.html" rel="alternate"
type="text/html" title="Thanks for using JupyterLab"/><summary>Big thanks to you, beloved
JupyterLab user.</summary><published>2022-11-02T14:00:00+00:00</published></entry></feed>"""
85: (0)             @patch("tornado.httpclient.AsyncHTTPClient", new_callable=fake_client_factory)
86: (0)             async def test_NewsHandler_multi_entry_links(mock_client, labserverapp,
jp_fetch):
87: (4)                 mock_client.body = FAKE_MULTI_ENTRY_LINKS_ATOM_FEED
88: (4)                 response = await jp_fetch("lab", "api", "news", method="GET")
89: (4)                 assert response.code == 200
90: (4)                 payload = json.loads(response.body)
91: (4)                 assert payload["news"] == [
92: (8)                     {
93: (12)                        "createdAt": 1667397600000.0,
94: (12)                        "message": "Thanks for using JupyterLab\nBig thanks to you,
beloved JupyterLab user.",
95: (12)                        "modifiedAt": 1667397600000.0,
96: (12)                        "type": "info",
97: (12)                        "link": [
98: (16)                            "Open full post",
99: (16)
"https://jupyterlab.github.io/assets/posts/2022/11/02/demo.html",
100: (12)                       ],
101: (12)                       "options": {
102: (16)                           "data": {
103: (20)                               "id":
"https://jupyterlab.github.io/assets/posts/2022/11/02/demo",
104: (20)                               "tags": ["news"],
105: (16)                           }
106: (12)                       },
107: (8)                    }
108: (4)                ]
109: (0)            FAKE_NO_PUBLISHED_ATOM_FEED = b"""<?xml version='1.0' encoding='UTF-8'?><feed
xmlns="http://www.w3.org/2005/Atom" xml:lang="en">
<id>https://jupyterlab.github.io/assets/feed.xml</id><title>JupyterLab News</title><updated>2023-
05-02T19:32:08.566055+00:00</updated><author><name>John Doe</name><email>john@example.de</email>
</author><link href="https://jupyterlab.github.io/assets/feed.xml" rel="self"
type="application/atom+xml"/><link href="https://jupyterlab.github.io/assets/" rel="alternate"
type="text/html"/><generator uri="https://lkiesow.github.io/python-feedgen"
version="0.9.0">python-feedgen</generator><logo>http://ex.com/logo.jpg</logo><subtitle>Subscribe
to get news about JupyterLab.</subtitle><entry>
<id>https://jupyterlab.github.io/assets/posts/2022/11/02/demo</id><title>Thanks for using
JupyterLab</title><updated>2022-11-02T14:00:00+00:00</updated><link
href="https://jupyterlab.github.io/assets/posts/2022/11/02/demo.html" rel="alternate"
type="text/html" title="Thanks for using JupyterLab"/><summary>Big thanks to you, beloved
JupyterLab user.</summary></entry></feed>"""
110: (0)            @patch("tornado.httpclient.AsyncHTTPClient", new_callable=fake_client_factory)
111: (0)            async def test_NewsHandler_no_published(mock_client, labserverapp, jp_fetch):
112: (4)                mock_client.body = FAKE_NO_PUBLISHED_ATOM_FEED
113: (4)                response = await jp_fetch("lab", "api", "news", method="GET")
114: (4)                assert response.code == 200
115: (4)                payload = json.loads(response.body)
116: (4)                assert payload["news"] == [
117: (8)                    {
118: (12)                       "createdAt": 1667397600000.0,
119: (12)                       "message": "Thanks for using JupyterLab\nBig thanks to you,
beloved JupyterLab user.",
120: (12)                       "modifiedAt": 1667397600000.0,
121: (12)                       "type": "info",
122: (12)                       "link": [
123: (16)                           "Open full post",
124: (16)
"https://jupyterlab.github.io/assets/posts/2022/11/02/demo.html",
125: (12)                       ],
126: (12)                       "options": {
127: (16)                           "data": {
128: (20)                               "id":
```

```
"https://jupyterlab.github.io/assets/posts/2022/11/02/demo",
129: (20)                               "tags": ["news"],
130: (16)                           }
131: (12)                       },
132: (8)                    }
133: (4)                ]
134: (0)            FAKE_LINK_NO_REL_ATOM_FEED = b"""<?xml version='1.0' encoding='UTF-8'?><feed
xmlns="http://www.w3.org/2005/Atom" xml:lang="en">
<id>https://jupyterlab.github.io/assets/feed.xml</id><title>JupyterLab News</title><updated>2023-
05-03T17:06:43.950978+00:00</updated><author><name>John Doe</name><email>john@example.de</email>
</author><link href="https://jupyterlab.github.io/assets/feed.xml" rel="self"
type="application/atom+xml"/><link href="https://jupyterlab.github.io/assets/" rel="alternate"
type="text/html"/><generator uri="https://lkiesow.github.io/python-feedgen"
version="0.9.0">python-feedgen</generator><logo>http://ex.com/logo.jpg</logo><subtitle>Subscribe
to get news about JupyterLab.</subtitle><entry>
<id>https://jupyterlab.github.io/assets/posts/2022/11/02/demo</id><title>Thanks for using
JupyterLab</title><updated>2022-11-02T14:00:00+00:00</updated><link
href="https://jupyterlab.github.io/assets/posts/2022/11/02/demo.html" type="text/html"
title="Thanks for using JupyterLab"/><summary>Big thanks to you, beloved JupyterLab user.
</summary><published>2022-11-02T14:00:00+00:00</published></entry></feed>"""
135: (0)            @patch("tornado.httpclient.AsyncHTTPClient", new_callable=fake_client_factory)
136: (0)            async def test_NewsHandler_link_no_rel(mock_client, labserverapp, jp_fetch):
137: (4)                mock_client.body = FAKE_LINK_NO_REL_ATOM_FEED
138: (4)                response = await jp_fetch("lab", "api", "news", method="GET")
139: (4)                assert response.code == 200
140: (4)                payload = json.loads(response.body)
141: (4)                assert payload["news"] == [
142: (8)                    {
143: (12)                       "createdAt": 1667397600000.0,
144: (12)                       "message": "Thanks for using JupyterLab\nBig thanks to you,
beloved JupyterLab user.",
145: (12)                       "modifiedAt": 1667397600000.0,
146: (12)                       "type": "info",
147: (12)                       "link": [
148: (16)                           "Open full post",
149: (16)
"https://jupyterlab.github.io/assets/posts/2022/11/02/demo.html",
150: (12)                       ],
151: (12)                       "options": {
152: (16)                           "data": {
153: (20)                               "id":
"https://jupyterlab.github.io/assets/posts/2022/11/02/demo",
154: (20)                               "tags": ["news"],
155: (16)                           }
156: (12)                       },
157: (8)                    }
158: (4)                ]
159: (0)            FAKE_NO_LINK_ATOM_FEED = b"""<?xml version='1.0' encoding='UTF-8'?><feed
xmlns="http://www.w3.org/2005/Atom" xml:lang="en">
<id>https://jupyterlab.github.io/assets/feed.xml</id><title>JupyterLab News</title><updated>2023-
05-03T17:06:43.950978+00:00</updated><author><name>John Doe</name><email>john@example.de</email>
</author><link href="https://jupyterlab.github.io/assets/feed.xml" rel="self"
type="application/atom+xml"/><link href="https://jupyterlab.github.io/assets/" rel="alternate"
type="text/html"/><generator uri="https://lkiesow.github.io/python-feedgen"
version="0.9.0">python-feedgen</generator><logo>http://ex.com/logo.jpg</logo><subtitle>Subscribe
to get news about JupyterLab.</subtitle><entry>
<id>https://jupyterlab.github.io/assets/posts/2022/11/02/demo</id><title>Thanks for using
JupyterLab</title><updated>2022-11-02T14:00:00+00:00</updated><summary>Big thanks to you, beloved
JupyterLab user.</summary><published>2022-11-02T14:00:00+00:00</published></entry></feed>"""
160: (0)            @patch("tornado.httpclient.AsyncHTTPClient", new_callable=fake_client_factory)
161: (0)            async def test_NewsHandler_no_links(mock_client, labserverapp, jp_fetch):
162: (4)                mock_client.body = FAKE_NO_LINK_ATOM_FEED
163: (4)                response = await jp_fetch("lab", "api", "news", method="GET")
164: (4)                assert response.code == 200
165: (4)                payload = json.loads(response.body)
166: (4)                assert payload["news"] == [
167: (8)                    {
168: (12)                       "createdAt": 1667397600000.0,
169: (12)                       "message": "Thanks for using JupyterLab\nBig thanks to you,
beloved JupyterLab user.",
170: (12)                        "modifiedAt": 1667397600000.0,
171: (12)                        "type": "info",
172: (12)                        "link": None,
173: (12)                        "options": {
174: (16)                            "data": {
175: (20)                                "id":
"https://jupyterlab.github.io/assets/posts/2022/11/02/demo",
176: (20)                                "tags": ["news"],
177: (16)                            }
178: (12)                        },
179: (8)                     }
180: (4)                 ]

----------------------------------------

File 32 - test_app.py:

1: (0)              """A lab app that runs a sub process for a demo or a test."""
2: (0)              import atexit
3: (0)              import json
4: (0)              import os
5: (0)              import shutil
6: (0)              import sys
7: (0)              import tempfile
8: (0)              try:
9: (4)                  from importlib.resources import files
10: (0)             except ImportError:
11: (4)                 from importlib_resources import files
12: (0)             from os import path as osp
13: (0)             from os.path import join as pjoin
14: (0)             from stat import S_IRGRP, S_IROTH, S_IRUSR
15: (0)             from tempfile import TemporaryDirectory
16: (0)             from unittest.mock import patch
17: (0)             import jupyter_core
18: (0)             import jupyterlab_server
19: (0)             from ipykernel.kernelspec import write_kernel_spec
20: (0)             from jupyter_server.serverapp import ServerApp
21: (0)             from jupyterlab_server.process_app import ProcessApp
22: (0)             from traitlets import default
23: (0)             HERE = osp.realpath(osp.dirname(__file__))
24: (0)             def _create_template_dir():
25: (4)                 template_dir = tempfile.mkdtemp(prefix="mock_static")
26: (4)                 index_filepath = osp.join(template_dir, "index.html")
27: (4)                 with open(index_filepath, "w") as fid:
28: (8)                     fid.write(
29: (12)                        """
30: (0)             <!DOCTYPE HTML>
31: (0)             <html>
32: (0)             <head>
33: (4)                 <meta charset="utf-8">
34: (4)                 <title>{% block title %}Jupyter Lab Test{% endblock %}</title>
35: (4)                 <meta http-equiv="X-UA-Compatible" content="IE=edge" />
36: (4)                 <meta name="viewport" content="width=device-width, initial-scale=1.0">
37: (4)                 {% block meta %}
38: (4)                 {% endblock %}
39: (0)             </head>
40: (0)             <body>
41: (2)               <h1>JupyterLab Test Application</h1>
42: (2)               <div id="site">
43: (4)                 {% block site %}
44: (4)                 {% endblock site %}
45: (2)               </div>
46: (2)               {% block after_site %}
47: (2)               {% endblock after_site %}
48: (0)             </body>
49: (0)             </html>"""
50: (8)                     )
51: (4)                 return template_dir
```

```
(0)            def _create_static_dir():
(4)                static_dir = tempfile.mkdtemp(prefix="mock_static")
(4)                return static_dir
(0)            def _create_schemas_dir():
(4)                """Create a temporary directory for schemas."""
(4)                root_dir = tempfile.mkdtemp(prefix="mock_schemas")
(4)                extension_dir = osp.join(root_dir, "@jupyterlab", "apputils-extension")
(4)                os.makedirs(extension_dir)
(4)                schema_package = jupyterlab_server.__name__
(4)                schema_path = "tests/schemas/@jupyterlab/apputils-extension/themes.json"
(4)                themes = files(schema_package).joinpath(schema_path).read_bytes()
(4)                with open(osp.join(extension_dir, "themes.json"), "w") as fid:
(8)                    fid.write(themes.decode("utf-8"))
(4)                atexit.register(lambda: shutil.rmtree(root_dir, True))
(4)                return root_dir
(0)            def _create_user_settings_dir():
(4)                """Create a temporary directory for workspaces."""
(4)                root_dir = tempfile.mkdtemp(prefix="mock_user_settings")
(4)                atexit.register(lambda: shutil.rmtree(root_dir, True))
(4)                return root_dir
(0)            def _create_workspaces_dir():
(4)                """Create a temporary directory for workspaces."""
(4)                root_dir = tempfile.mkdtemp(prefix="mock_workspaces")
(4)                atexit.register(lambda: shutil.rmtree(root_dir, True))
(4)                return root_dir
(0)            class TestEnv:
(4)                """Set Jupyter path variables to a temporary directory
(4)                Useful as a context manager or with explicit start/stop
(4)                """
(4)                def start(self):
(8)                    self.test_dir = td = TemporaryDirectory()
(8)                    self.env_patch = patch.dict(
(12)                       os.environ,
(12)                       {
(16)                           "JUPYTER_CONFIG_DIR": pjoin(td.name, "jupyter"),
(16)                           "JUPYTER_DATA_DIR": pjoin(td.name, "jupyter_data"),
(16)                           "JUPYTER_RUNTIME_DIR": pjoin(td.name, "jupyter_runtime"),
(16)                           "IPYTHONDIR": pjoin(td.name, "ipython"),
(12)                       },
(8)                    )
(8)                    self.env_patch.start()
(8)                    self.path_patch = patch.multiple(
(12)                       jupyter_core.paths,
(12)                       SYSTEM_JUPYTER_PATH=[pjoin(td.name, "share", "jupyter")],
(12)                       ENV_JUPYTER_PATH=[pjoin(td.name, "env", "share", "jupyter")],
(12)                       SYSTEM_CONFIG_PATH=[pjoin(td.name, "etc", "jupyter")],
(12)                       ENV_CONFIG_PATH=[pjoin(td.name, "env", "etc", "jupyter")],
(8)                    )
(8)                    self.path_patch.start()
(4)                def stop(self):
(8)                    self.env_patch.stop()
(8)                    self.path_patch.stop()
(8)                    try:
(12)                       self.test_dir.cleanup()
(8)                    except OSError:
(12)                       pass
(4)                def __enter__(self):
(8)                    self.start()
(8)                    return self.test_dir.name
(4)                def __exit__(self, *exc_info):
(8)                    self.stop()
(0)            class ProcessTestApp(ProcessApp):
(4)                """A process app for running tests, includes a mock contents directory."""
(4)                allow_origin = "*"
(4)                def initialize_templates(self):
(8)                    self.static_paths = [_create_static_dir()]
(8)                    self.template_paths = [_create_template_dir()]
(4)                def initialize_settings(self):
(8)                    self.env_patch = TestEnv()
```

```
(8)                 self.env_patch.start()
(8)                 ProcessApp.__init__(self)
(8)                 self.settings["allow_origin"] = ProcessTestApp.allow_origin
(8)                 self.static_dir = self.static_paths[0]
(8)                 self.template_dir = self.template_paths[0]
(8)                 self.schemas_dir = _create_schemas_dir()
(8)                 self.user_settings_dir = _create_user_settings_dir()
(8)                 self.workspaces_dir = _create_workspaces_dir()
(8)                 self._install_default_kernels()
(8)                 self.settings["kernel_manager"].default_kernel_name = "echo"
(8)                 super().initialize_settings()
(4)             def _install_kernel(self, kernel_name, kernel_spec):
(8)                 """Install a kernel spec to the data directory.
(8)                 Parameters
(8)                 ----------
(8)                 kernel_name: str
(12)                    Name of the kernel.
(8)                 kernel_spec: dict
(12)                    The kernel spec for the kernel
(8)                 """
(8)                 paths = jupyter_core.paths
(8)                 kernel_dir = pjoin(paths.jupyter_data_dir(), "kernels", kernel_name)
(8)                 os.makedirs(kernel_dir)
(8)                 with open(pjoin(kernel_dir, "kernel.json"), "w") as f:
(12)                    f.write(json.dumps(kernel_spec))
(4)             def _install_default_kernels(self):
(8)                 self._install_kernel(
(12)                    kernel_name="echo",
(12)                    kernel_spec={
(16)                        "argv": [
(20)                            sys.executable,
(20)                            "-m",
(20)                            "jupyterlab.tests.echo_kernel",
(20)                            "-f",
(20)                            "{connection_file}",
(16)                        ],
(16)                        "display_name": "Echo Kernel",
(16)                        "language": "echo",
(12)                    },
(8)                 )
(8)                 paths = jupyter_core.paths
(8)                 ipykernel_dir = pjoin(paths.jupyter_data_dir(), "kernels", "ipython")
(8)                 write_kernel_spec(ipykernel_dir)
(4)             def _process_finished(self, future):
(8)                 self.serverapp.http_server.stop()
(8)                 self.serverapp.io_loop.stop()
(8)                 self.env_patch.stop()
(8)                 try:
(12)                    os._exit(future.result())
(8)                 except Exception as e:
(12)                    self.log.error(str(e))
(12)                    os._exit(1)
(0)         class RootedServerApp(ServerApp):
(4)             @default("root_dir")
(4)             def _default_root_dir(self):
(8)                 """Create a temporary directory with some file structure."""
(8)                 root_dir = tempfile.mkdtemp(prefix="mock_root")
(8)                 os.mkdir(osp.join(root_dir, "src"))
(8)                 with open(osp.join(root_dir, "src", "temp.txt"), "w") as fid:
(12)                    fid.write("hello")
(8)                 readonly_filepath = osp.join(root_dir, "src", "readonly-temp.txt")
(8)                 with open(readonly_filepath, "w") as fid:
(12)                    fid.write("hello from a readonly file")
(8)                 os.chmod(readonly_filepath, S_IRUSR | S_IRGRP | S_IROTH)
(8)                 atexit.register(lambda: shutil.rmtree(root_dir, True))
(8)                 return root_dir
```

File 33 - test_build_api.py:

```
(0)              """Test the kernels service API."""
(0)              import asyncio
(0)              import json
(0)              import os
(0)              from tempfile import TemporaryDirectory
(0)              import pytest
(0)              import tornado
(0)              def expected_http_error(error, expected_code, expected_message=None):
(4)                  """Check that the error matches the expected output error."""
(4)                  e = error.value
(4)                  if isinstance(e, tornado.web.HTTPError):
(8)                      if expected_code != e.status_code:
(12)                         return False
(8)                      if expected_message is not None and expected_message != str(e):
(12)                         return False
(8)                      return True
(4)                  elif any(
(8)                      [
(12)                         isinstance(e, tornado.httpclient.HTTPClientError),
(12)                         isinstance(e, tornado.httpclient.HTTPError),
(8)                      ]
(4)                  ):
(8)                      if expected_code != e.code:
(12)                         return False
(8)                      if expected_message:
(12)                         message = json.loads(e.response.body.decode())["message"]
(12)                         if expected_message != message:
(16)                             return False
(8)                      return True
(0)              @pytest.fixture
(0)              def build_api_tester(jp_serverapp, labapp, fetch_long):
(4)                  return BuildAPITester(labapp, fetch_long)
(0)              class BuildAPITester:
(4)                  """Wrapper for build REST API requests"""
(4)                  url = "lab/api/build"
(4)                  def __init__(self, labapp, fetch_long):
(8)                      self.labapp = labapp
(8)                      self.fetch = fetch_long
(4)                  async def _req(self, verb, path, body=None):
(8)                      return await self.fetch(self.url + path, method=verb, body=body)
(4)                  async def getStatus(self):
(8)                      return await self._req("GET", "")
(4)                  async def build(self):
(8)                      return await self._req("POST", "", json.dumps({}))
(4)                  async def clear(self):
(8)                      return await self._req("DELETE", "")
(0)              @pytest.mark.slow
(0)              class TestBuildAPI:
(4)                  def tempdir(self):
(8)                      td = TemporaryDirectory()
(8)                      self.tempdirs.append(td)
(8)                      return td.name
(4)                  def setUp(self):
(8)                      self.tempdirs = []
(8)                      @self.addCleanup
(8)                      def cleanup_tempdirs():
(12)                         for d in self.tempdirs:
(16)                             d.cleanup()
(4)                  async def test_get_status(self, build_api_tester):
(8)                      """Make sure there are no kernels running at the start"""
(8)                      r = await build_api_tester.getStatus()
(8)                      res = r.body.decode()
(8)                      resp = json.loads(res)
(8)                      assert "status" in resp
(8)                      assert "message" in resp
(4)                  @pytest.mark.skipif(os.name == "nt", reason="Currently failing on
windows")
```

```
67: (4)                 async def test_build(self, build_api_tester):
68: (8)                     r = await build_api_tester.build()
69: (8)                     assert r.code == 200
70: (4)                 @pytest.mark.skipif(os.name == "nt", reason="Currently failing on windows")
71: (4)                 async def test_clear(self, build_api_tester):
72: (8)                     with pytest.raises(tornado.httpclient.HTTPClientError) as e:
73: (12)                        r = await build_api_tester.clear()
74: (12)                        res = r.body.decode()
75: (8)                     assert expected_http_error(e, 500)
76: (8)                     loop = asyncio.get_event_loop()
77: (8)                     asyncio.ensure_future(build_api_tester.build(), loop=loop)  # noqa RUF006
78: (8)                     while True:
79: (12)                        r = await build_api_tester.getStatus()
80: (12)                        res = r.body.decode()
81: (12)                        resp = json.loads(res)
82: (12)                        if resp["status"] == "building":
83: (16)                            break
84: (8)                     r = await build_api_tester.clear()
85: (8)                     assert r.code == 204
```

File 34 - test_custom_css_handler.py:

```
1: (0)              import os
2: (0)              import pytest
3: (0)              CUSTOM_CSS = """body #top-panel-wrapper,
4: (2)                background-color: #aecad4 !important;
5: (0)              }
6: (0)              body h1 {
7: (2)                font-size: 22px;
8: (2)                margin-bottom: 40px;
9: (2)                color: #10929e;
10: (2)               text-decoration: underline;
11: (0)             }"""
12: (0)             @pytest.fixture
13: (0)             def jp_server_config(jp_server_config, tmp_path):
14: (4)                 config = jp_server_config.copy()
15: (4)                 config["LabApp"]["custom_css"] = True
16: (4)                 return config
17: (0)             async def test_CustomCssHandler(tmp_path, jp_serverapp, labserverapp, jp_fetch):
18: (4)                 custom_path = tmp_path / "config" / "custom"
19: (4)                 assert str(custom_path) in jp_serverapp.web_app.settings["static_custom_path"]
20: (4)                 custom_path.mkdir(parents=True, exist_ok=True)
21: (4)                 (custom_path / "custom.css").write_text(CUSTOM_CSS)
22: (4)                 response = await jp_fetch("custom", "custom.css", method="GET")
23: (4)                 assert response.code == 200
24: (4)                 assert response.body.decode().replace(os.linesep, "\n") == CUSTOM_CSS
```

File 35 - test_extensions.py:

```
1: (0)              import json
2: (0)              from unittest.mock import Mock, patch
3: (0)              import pytest
4: (0)              from traitlets.config import Config, Configurable
5: (0)              from jupyterlab.extensions import PyPIExtensionManager, ReadOnlyExtensionManager
6: (0)              from jupyterlab.extensions.manager import ExtensionManager, ExtensionPackage, PluginManager
7: (0)              from . import fake_client_factory
8: (0)              @pytest.mark.parametrize(
9: (4)                  "version, expected",
10: (4)                 (
11: (8)                         ("1", "1"),
12: (8)                         ("1.0", "1.0"),
13: (8)                         ("1.0.0", "1.0.0"),
14: (8)                         ("1.0.0a52", "1.0.0-alpha.52"),
15: (8)                         ("1.0.0b3", "1.0.0-beta.3"),
16: (8)                         ("1.0.0rc22", "1.0.0-rc.22"),
17: (8)                         ("1.0.0rc23.post2", "1.0.0-rc.23"),
18: (8)                         ("1.0.0rc24.dev2", "1.0.0-rc.24"),
19: (8)                         ("1.0.0rc25.post4.dev2", "1.0.0-rc.25"),
20: (4)                     ),
21: (0)                 )
22: (0)                 def test_ExtensionManager_get_semver_version(version, expected):
23: (4)                     assert ExtensionManager.get_semver_version(version) == expected
24: (0)                 async def test_ExtensionManager_list_extensions_installed(monkeypatch):
25: (4)                     extension1 = ExtensionPackage("extension1", "Extension 1 description", "",
"prebuilt")
26: (4)                     async def mock_installed(*args, **kwargs):
27: (8)                         return {"extension1": extension1}
28: (4)                     monkeypatch.setattr(ReadOnlyExtensionManager, "_get_installed_extensions",
mock_installed)
29: (4)                     manager = ReadOnlyExtensionManager()
30: (4)                     extensions = await manager.list_extensions()
31: (4)                     assert extensions == ([extension1], 1)
32: (0)                 async def test_ExtensionManager_list_extensions_query(monkeypatch):
33: (4)                     extension1 = ExtensionPackage("extension1", "Extension 1 description", "",
"prebuilt")
34: (4)                     extension2 = ExtensionPackage("extension2", "Extension 2 description", "",
"prebuilt")
35: (4)                     async def mock_list(*args, **kwargs):
36: (8)                         return {"extension1": extension1, "extension2": extension2}, None
37: (4)                     monkeypatch.setattr(ReadOnlyExtensionManager, "list_packages", mock_list)
38: (4)                     manager = ReadOnlyExtensionManager()
39: (4)                     extensions = await manager.list_extensions("ext")
40: (4)                     assert extensions == ([extension1, extension2], 1)
41: (0)                 @patch("tornado.httpclient.AsyncHTTPClient", new_callable=fake_client_factory)
42: (0)                 async def test_ExtensionManager_list_extensions_query_allow(mock_client,
monkeypatch):
43: (4)                     extension1 = ExtensionPackage("extension1", "Extension 1 description", "",
"prebuilt")
44: (4)                     extension2 = ExtensionPackage("extension2", "Extension 2 description", "",
"prebuilt")
45: (4)                     mock_client.body = json.dumps({"allowed_extensions": [{"name":
"extension1"}]}).encode()
46: (4)                     async def mock_list(*args, **kwargs):
47: (8)                         return {"extension1": extension1, "extension2": extension2}, None
48: (4)                     monkeypatch.setattr(ReadOnlyExtensionManager, "list_packages", mock_list)
49: (4)                     manager = ReadOnlyExtensionManager(
50: (8)                         ext_options={"allowed_extensions_uris": {"http://dummy-allowed-
extension"}},
51: (4)                     )
52: (4)                     extensions = await manager.list_extensions("ext")
53: (4)                     assert extensions == ([extension1], 1)
54: (0)                 @patch("tornado.httpclient.AsyncHTTPClient", new_callable=fake_client_factory)
55: (0)                 async def test_ExtensionManager_list_extensions_query_block(mock_client,
monkeypatch):
56: (4)                     extension1 = ExtensionPackage("extension1", "Extension 1 description", "",
"prebuilt")
57: (4)                     extension2 = ExtensionPackage("extension2", "Extension 2 description", "",
"prebuilt")
58: (4)                     mock_client.body = json.dumps({"blocked_extensions": [{"name":
"extension1"}]}).encode()
59: (4)                     async def mock_list(*args, **kwargs):
60: (8)                         return {"extension1": extension1, "extension2": extension2}, None
61: (4)                     monkeypatch.setattr(ReadOnlyExtensionManager, "list_packages", mock_list)
62: (4)                     manager = ReadOnlyExtensionManager(
63: (8)                         ext_options={"blocked_extensions_uris": {"http://dummy-blocked-
extension"}}
64: (4)                     )
65: (4)                     extensions = await manager.list_extensions("ext")
```

```
66: (4)                 assert extensions == ([extension2], 1)
67: (0)             @patch("tornado.httpclient.AsyncHTTPClient", new_callable=fake_client_factory)
68: (0)             async def test_ExtensionManager_list_extensions_query_allow_block(mock_client,
monkeypatch):
69: (4)                 extension1 = ExtensionPackage("extension1", "Extension 1 description", "",
"prebuilt")
70: (4)                 extension2 = ExtensionPackage("extension2", "Extension 2 description", "",
"prebuilt")
71: (4)                 mock_client.body = json.dumps(
72: (8)                     {
73: (12)                        "allowed_extensions": [{"name": "extension1"}],
74: (12)                        "blocked_extensions": [{"name": "extension1"}],
75: (8)                     }
76: (4)                 ).encode()
77: (4)                 async def mock_list(*args, **kwargs):
78: (8)                     return {"extension1": extension1, "extension2": extension2}, None
79: (4)                 monkeypatch.setattr(ReadOnlyExtensionManager, "list_packages", mock_list)
80: (4)                 manager = ReadOnlyExtensionManager(
81: (8)                     ext_options={
82: (12)                        "allowed_extensions_uris": {"http://dummy-allowed-extension"},
83: (12)                        "blocked_extensions_uris": {"http://dummy-blocked-extension"},
84: (8)                     }
85: (4)                 )
86: (4)                 extensions = await manager.list_extensions("ext")
87: (4)                 assert extensions == ([extension1], 1)
88: (0)             async def test_ExtensionManager_install():
89: (4)                 manager = ReadOnlyExtensionManager()
90: (4)                 result = await manager.install("extension1")
91: (4)                 assert result.status == "error"
92: (4)                 assert result.message == "Extension installation not supported."
93: (0)             async def test_ExtensionManager_uninstall():
94: (4)                 manager = ReadOnlyExtensionManager()
95: (4)                 result = await manager.uninstall("extension1")
96: (4)                 assert result.status == "error"
97: (4)                 assert result.message == "Extension removal not supported."
98: (0)             @patch("jupyterlab.extensions.pypi.xmlrpc.client")
99: (0)             async def test_PyPiExtensionManager_list_extensions_query(mocked_rpcclient):
100: (4)                extension1 = ExtensionPackage(
101: (8)                    name="jupyterlab-git",
102: (8)                    description="A JupyterLab extension for version control using git",
103: (8)                    homepage_url="https://github.com/jupyterlab/jupyterlab-git",
104: (8)                    pkg_type="prebuilt",
105: (8)                    latest_version="0.37.1",
106: (8)                    author="Jupyter Development Team",
107: (8)                    license="BSD-3-Clause",
108: (8)                    package_manager_url="https://pypi.org/project/jupyterlab-git/",
109: (4)                )
110: (4)                extension2 = ExtensionPackage(
111: (8)                    name="jupyterlab-github",
112: (8)                    description="JupyterLab viewer for GitHub repositories",
113: (8)                    homepage_url="https://github.com/jupyterlab/jupyterlab-
github/blob/main/README.md",
114: (8)                    pkg_type="prebuilt",
115: (8)                    latest_version="3.0.1",
116: (8)                    author="Ian Rose",
117: (8)                    license="BSD-3-Clause",
118: (8)                    bug_tracker_url="https://github.com/jupyterlab/jupyterlab-
github/issues",
119: (8)                    package_manager_url="https://pypi.org/project/jupyterlab-github/",
120: (8)                    repository_url="https://github.com/jupyterlab/jupyterlab-github",
121: (4)                )
122: (4)                proxy = Mock(
123: (8)                    browse=Mock(
124: (12)                       return_value=[
125: (16)                           ["jupyterlab-git", "0.33.0"],
126: (16)                           ["jupyterlab-git", "0.34.0"],
127: (16)                           ["jupyterlab-git", "0.34.1"],
128: (16)                           ["jupyterlab-git", "0.37.0"],
129: (16)                           ["jupyterlab-git", "0.37.1"],
(16)                ["jupyterlab-github", "3.0.0"],
(16)                ["jupyterlab-github", "3.0.1"],
(12)            ]
(8)         ),
(4)     )
(4)     mocked_rpcclient.ServerProxy = Mock(return_value=proxy)
(4)     manager = PyPIExtensionManager()
(4)     async def mock_pkg_metadata(n, l, b):  # noqa
(8)         return (
(12)            {
(16)                "name": "jupyterlab-git",
(16)                "version": "0.37.1",
(16)                "stable_version": None,
(16)                "bugtrack_url": None,
(16)                "package_url": "https://pypi.org/project/jupyterlab-git/",
(16)                "release_url": "https://pypi.org/project/jupyterlab-git/0.37.1/",
(16)                "docs_url": None,
(16)                "home_page": "https://github.com/jupyterlab/jupyterlab-git",
(16)                "download_url": "",
(16)                "project_url": "",
(16)                "project_urls": {},
(16)                "author": "Jupyter Development Team",
(16)                "author_email": "",
(16)                "maintainer": "",
(16)                "maintainer_email": "",
(16)                "summary": "A JupyterLab extension for version control using git",
(16)                "license": "BSD-3-Clause",
(16)                "keywords": "Jupyter,JupyterLab,JupyterLab3,jupyterlab-extension,Git",
(16)                "platform": "Linux",
(16)                "classifiers": [
(20)                    "Framework :: Jupyter",
(20)                    "Framework :: Jupyter :: JupyterLab",
(20)                    "Framework :: Jupyter :: JupyterLab :: 3",
(20)                    "Framework :: Jupyter :: JupyterLab :: Extensions",
(20)                    "Framework :: Jupyter :: JupyterLab :: Extensions :: Prebuilt",
(20)                    "Intended Audience :: Developers",
(20)                    "Intended Audience :: Science/Research",
(20)                    "License :: OSI Approved :: BSD License",
(20)                    "Programming Language :: Python",
(20)                    "Programming Language :: Python :: 3",
(20)                    "Programming Language :: Python :: 3.10",
(20)                    "Programming Language :: Python :: 3.6",
(20)                    "Programming Language :: Python :: 3.7",
(20)                    "Programming Language :: Python :: 3.8",
(20)                    "Programming Language :: Python :: 3.9",
(16)                ],
(16)                "requires": [],
(16)                "requires_dist": [
(20)                    "jupyter-server",
(20)                    "nbdime (~=3.1)",
(20)                    "nbformat",
(20)                    "packaging",
(20)                    "pexpect",
(20)                    "coverage ; extra == 'dev'",
(20)                    "jupyter-packaging (~=0.7.9) ; extra == 'dev'",
(20)                    "jupyterlab (~=3.0) ; extra == 'dev'",
(20)                    "pre-commit ; extra == 'dev'",
(20)                    "pytest ; extra == 'dev'",
(20)                    "pytest-asyncio ; extra == 'dev'",
(20)                    "pytest-cov ; extra == 'dev'",
(20)                    "pytest-tornasync ; extra == 'dev'",
(20)                    "coverage ; extra == 'tests'",
(20)                    "jupyter-packaging (~=0.7.9) ; extra == 'tests'",
(20)                    "jupyterlab (~=3.0) ; extra == 'tests'",
(20)                    "pre-commit ; extra == 'tests'",
(20)                     "pytest ; extra == 'tests'",
(20)                     "pytest-asyncio ; extra == 'tests'",
(20)                     "pytest-cov ; extra == 'tests'",
(20)                     "pytest-tornasync ; extra == 'tests'",
(20)                     "hybridcontents ; extra == 'tests'",
(20)                     "jupytext ; extra == 'tests'",
(16)                 ],
(16)                 "provides": [],
(16)                 "provides_dist": [],
(16)                 "obsoletes": [],
(16)                 "obsoletes_dist": [],
(16)                 "requires_python": "<4,>=3.6",
(16)                 "requires_external": [],
(16)                 "_pypi_ordering": 55,
(16)                 "downloads": {"last_day": -1, "last_week": -1, "last_month": -1},
(16)                 "cheesecake_code_kwalitee_id": None,
(16)                 "cheesecake_documentation_id": None,
(16)                 "cheesecake_installability_id": None,
(12)             }
(12)             if n == "jupyterlab-git"
(12)             else {
(16)                 "name": "jupyterlab-github",
(16)                 "version": "3.0.1",
(16)                 "stable_version": None,
(16)                 "bugtrack_url": None,
(16)                 "package_url": "https://pypi.org/project/jupyterlab-github/",
(16)                 "release_url": "https://pypi.org/project/jupyterlab-github/3.0.1/",
(16)                 "docs_url": None,
(16)                 "home_page": "",
(16)                 "download_url": "",
(16)                 "project_url": "",
(16)                 "project_urls": {
(20)                     "Homepage": "https://github.com/jupyterlab/jupyterlab-github/blob/main/README.md",
(20)                     "Bug Tracker": "https://github.com/jupyterlab/jupyterlab-github/issues",
(20)                     "Source Code": "https://github.com/jupyterlab/jupyterlab-github",
(16)                 },
(16)                 "author": "Ian Rose",
(16)                 "author_email": "jupyter@googlegroups.com",
(16)                 "maintainer": "",
(16)                 "maintainer_email": "",
(16)                 "summary": "JupyterLab viewer for GitHub repositories",
(16)                 "license": "BSD-3-Clause",
(16)                 "keywords": "Jupyter,JupyterLab,JupyterLab3",
(16)                 "platform": "Linux",
(16)                 "classifiers": [
(20)                     "Framework :: Jupyter",
(20)                     "Framework :: Jupyter :: JupyterLab",
(20)                     "Framework :: Jupyter :: JupyterLab :: 3",
(20)                     "Framework :: Jupyter :: JupyterLab :: Extensions",
(20)                     "Framework :: Jupyter :: JupyterLab :: Extensions :: Prebuilt",
(20)                     "License :: OSI Approved :: BSD License",
(20)                     "Programming Language :: Python",
(20)                     "Programming Language :: Python :: 3",
(20)                     "Programming Language :: Python :: 3.6",
(20)                     "Programming Language :: Python :: 3.7",
(20)                     "Programming Language :: Python :: 3.8",
(20)                     "Programming Language :: Python :: 3.9",
(16)                 ],
(16)                 "requires": [],
(16)                 "requires_dist": ["jupyterlab (~=3.0)"],
(16)                 "provides": [],
(16)                 "provides_dist": [],
(16)                 "obsoletes": [],
258: (16)                           "obsoletes_dist": [],
259: (16)                           "requires_python": ">=3.6",
260: (16)                           "requires_external": [],
261: (16)                           "_pypi_ordering": 12,
262: (16)                           "downloads": {"last_day": -1, "last_week": -1, "last_month":
-1},
263: (16)                           "cheesecake_code_kwalitee_id": None,
264: (16)                           "cheesecake_documentation_id": None,
265: (16)                           "cheesecake_installability_id": None,
266: (12)                       }
267: (8)                    )
268: (4)                manager._fetch_package_metadata = mock_pkg_metadata
269: (4)                extensions = await manager.list_extensions("git")
270: (4)                assert extensions == ([extension1, extension2], 1)
271: (0)            async def test_PyPiExtensionManager_custom_server_url():
272: (4)                BASE_URL = "https://mylocal.pypi.server/pypi"  # noqa
273: (4)                parent = Configurable(config=Config({"PyPIExtensionManager": {"base_url":
BASE_URL}}))
274: (4)                manager = PyPIExtensionManager(parent=parent)
275: (4)                assert manager.base_url == BASE_URL
276: (0)            LEVELS = ["user", "sys_prefix", "system"]
277: (0)            @pytest.mark.parametrize("level", LEVELS)
278: (0)            async def test_PyPiExtensionManager_custom_level(level):
279: (4)                parent = Configurable(config=Config({"PyPIExtensionManager": {"level":
level}}))
280: (4)                manager = PyPIExtensionManager(parent=parent)
281: (4)                assert manager.level == level
282: (0)            @pytest.mark.parametrize("level", LEVELS)
283: (0)            async def test_PyPiExtensionManager_inherits_custom_level(level):
284: (4)                parent = Configurable(config=Config({"PluginManager": {"level": level}}))
285: (4)                manager = PyPIExtensionManager(parent=parent)
286: (4)                assert manager.level == level
287: (0)            @pytest.mark.parametrize("level", LEVELS)
288: (0)            async def test_PluginManager_custom_level(level):
289: (4)                parent = Configurable(config=Config({"PluginManager": {"level": level}}))
290: (4)                manager = PluginManager(parent=parent)
291: (4)                assert manager.level == level
292: (0)            async def test_PluginManager_default_level():
293: (4)                manager = PluginManager()
294: (4)                assert manager.level == "sys_prefix"

----------------------------------------

File 36 - test_jupyterlab.py:

1: (0)              """Test installation of JupyterLab extensions"""
2: (0)              import glob
3: (0)              import json
4: (0)              import logging
5: (0)              import os
6: (0)              import platform
7: (0)              import shutil
8: (0)              import subprocess
9: (0)              import sys
10: (0)             from os.path import join as pjoin
11: (0)             from pathlib import Path
12: (0)             from tempfile import TemporaryDirectory
13: (0)             from unittest import TestCase
14: (0)             from unittest.mock import patch
15: (0)             import pytest
16: (0)             from jupyter_core import paths
17: (0)             from jupyterlab import commands
18: (0)             from jupyterlab.commands import (
19: (4)                 DEV_DIR,
20: (4)                 AppOptions,
21: (4)                 _compare_ranges,
22: (4)                 _test_overlap,
23: (4)                 build,
24: (4)                 build_check,
    check_extension,
    disable_extension,
    enable_extension,
    get_app_info,
    get_app_version,
    install_extension,
    link_package,
    list_extensions,
    uninstall_extension,
    unlink_package,
    update_extension,
)
from jupyterlab.coreconfig import CoreConfig, _get_default_core_data
here = os.path.dirname(os.path.abspath(__file__))
def touch(file, mtime=None):
    """ensure a file exists, and set its modification time
    returns the modification time of the file
    """
    dirname = os.path.dirname(file)
    if not os.path.exists(dirname):
        os.makedirs(dirname)
    open(file, "a").close()
    if mtime:
        atime = os.stat(file).st_atime
        os.utime(file, (atime, mtime))
    return os.stat(file).st_mtime
class AppHandlerTest(TestCase):
    def tempdir(self):
        td = TemporaryDirectory()
        self.tempdirs.append(td)
        return td.name
    def setUp(self):
        self.tempdirs = []
        self.devnull = open(os.devnull, "w")  # noqa
        @self.addCleanup
        def cleanup_tempdirs():
            for d in self.tempdirs:
                d.cleanup()
        self.test_dir = self.tempdir()
        self.data_dir = pjoin(self.test_dir, "data")
        self.config_dir = pjoin(self.test_dir, "config")
        self.pkg_names = {}
        for name in ["extension", "incompat", "package", "mimeextension"]:
            src = pjoin(here, "mock_packages", name)
            def ignore(dname, files):
                if "node_modules" in dname:
                    files = []
                if "node_modules" in files:
                    files.remove("node_modules")
                return dname, files
            dest = pjoin(self.test_dir, name)
            shutil.copytree(src, dest, ignore=ignore)
            if not os.path.exists(pjoin(dest, "node_modules")):
                os.makedirs(pjoin(dest, "node_modules"))
            setattr(self, "mock_" + name, dest)
            with open(pjoin(dest, "package.json")) as fid:
                data = json.load(fid)
            self.pkg_names[name] = data["name"]
        self.patches = []
        p = patch.dict(
            "os.environ",
            {
                "JUPYTER_CONFIG_DIR": self.config_dir,
                "JUPYTER_DATA_DIR": self.data_dir,
                "JUPYTERLAB_DIR": pjoin(self.data_dir, "lab"),
            },
        )
        self.patches.append(p)
        for mod in [paths]:
```

```
(12)                if hasattr(mod, "ENV_JUPYTER_PATH"):
(16)                    p = patch.object(mod, "ENV_JUPYTER_PATH", [self.data_dir])
(16)                    self.patches.append(p)
(12)                if hasattr(mod, "ENV_CONFIG_PATH"):
(16)                    p = patch.object(mod, "ENV_CONFIG_PATH", [self.config_dir])
(16)                    self.patches.append(p)
(12)                if hasattr(mod, "CONFIG_PATH"):
(16)                    p = patch.object(mod, "CONFIG_PATH", self.config_dir)
(16)                    self.patches.append(p)
(12)                if hasattr(mod, "BUILD_PATH"):
(16)                    p = patch.object(mod, "BUILD_PATH", self.data_dir)
(16)                    self.patches.append(p)
(8)             for p in self.patches:
(12)                p.start()
(12)                self.addCleanup(p.stop)
(8)             self.assertEqual(paths.ENV_CONFIG_PATH, [self.config_dir])
(8)             self.assertEqual(paths.ENV_JUPYTER_PATH, [self.data_dir])
(8)             self.assertEqual(
(12)                Path(commands.get_app_dir()).resolve(), (Path(self.data_dir) / "lab").resolve()
(8)             )
(8)             self.app_dir = commands.get_app_dir()
(8)             self.pinned_packages = ["jupyterlab-test-extension@1.0", "jupyterlab-test-extension@2.0"]
(0)     class TestExtension(AppHandlerTest):
(4)         def test_install_extension(self):
(8)             assert install_extension(self.mock_extension) is True
(8)             path = pjoin(self.app_dir, "extensions", "*.tgz")
(8)             assert glob.glob(path)
(8)             extensions = get_app_info()["extensions"]
(8)             name = self.pkg_names["extension"]
(8)             assert name in extensions
(8)             assert check_extension(name)
(4)         def test_install_twice(self):
(8)             assert install_extension(self.mock_extension) is True
(8)             path = pjoin(self.app_dir, "extensions", "*.tgz")
(8)             assert install_extension(self.mock_extension) is True
(8)             assert glob.glob(path)
(8)             extensions = get_app_info()["extensions"]
(8)             name = self.pkg_names["extension"]
(8)             assert name in extensions
(8)             assert check_extension(name)
(4)         def test_install_mime_renderer(self):
(8)             install_extension(self.mock_mimeextension)
(8)             name = self.pkg_names["mimeextension"]
(8)             assert name in get_app_info()["extensions"]
(8)             assert check_extension(name)
(8)             assert uninstall_extension(name) is True
(8)             assert name not in get_app_info()["extensions"]
(8)             assert not check_extension(name)
(4)         def test_install_incompatible(self):
(8)             with pytest.raises(ValueError) as excinfo:
(12)                install_extension(self.mock_incompat)
(8)             assert "Conflicting Dependencies" in str(excinfo.value)
(8)             assert not check_extension(self.pkg_names["incompat"])
(4)         def test_install_failed(self):
(8)             path = self.mock_package
(8)             with pytest.raises(ValueError):
(12)                install_extension(path)
(8)             with open(pjoin(path, "package.json")) as fid:
(12)                data = json.load(fid)
(8)             extensions = get_app_info()["extensions"]
(8)             name = data["name"]
(8)             assert name not in extensions
(8)             assert not check_extension(name)
(4)         def test_validation(self):
(8)             path = self.mock_extension
(8)             os.remove(pjoin(path, "index.js"))
(8)             with pytest.raises(ValueError):
```

```
(12)            install_extension(path)
(8)         assert not check_extension(self.pkg_names["extension"])
(8)         path = self.mock_mimeextension
(8)         os.remove(pjoin(path, "index.js"))
(8)         with pytest.raises(ValueError):
(12)            install_extension(path)
(8)         assert not check_extension(self.pkg_names["mimeextension"])
(4)     def test_uninstall_extension(self):
(8)         assert install_extension(self.mock_extension) is True
(8)         name = self.pkg_names["extension"]
(8)         assert check_extension(name)
(8)         assert uninstall_extension(self.pkg_names["extension"]) is True
(8)         path = pjoin(self.app_dir, "extensions", "*.tgz")
(8)         assert not glob.glob(path)
(8)         extensions = get_app_info()["extensions"]
(8)         assert name not in extensions
(8)         assert not check_extension(name)
(4)     def test_uninstall_all_extensions(self):
(8)         install_extension(self.mock_extension)
(8)         install_extension(self.mock_mimeextension)
(8)         ext_name = self.pkg_names["extension"]
(8)         mime_ext_name = self.pkg_names["mimeextension"]
(8)         assert check_extension(ext_name) is True
(8)         assert check_extension(mime_ext_name) is True
(8)         assert uninstall_extension(all_=True) is True
(8)         extensions = get_app_info()["extensions"]
(8)         assert ext_name not in extensions
(8)         assert mime_ext_name not in extensions
(4)     @pytest.mark.slow
(4)     def test_uninstall_core_extension(self):
(8)         assert uninstall_extension("@jupyterlab/console-extension") is True
(8)         app_dir = self.app_dir
(8)         build()
(8)         with open(pjoin(app_dir, "staging", "package.json")) as fid:
(12)            data = json.load(fid)
(8)         extensions = data["jupyterlab"]["extensions"]
(8)         assert "@jupyterlab/console-extension" not in extensions
(8)         assert not check_extension("@jupyterlab/console-extension")
(8)         assert install_extension("@jupyterlab/console-extension") is True
(8)         build()
(8)         with open(pjoin(app_dir, "staging", "package.json")) as fid:
(12)            data = json.load(fid)
(8)         extensions = data["jupyterlab"]["extensions"]
(8)         assert "@jupyterlab/console-extension" in extensions
(8)         assert check_extension("@jupyterlab/console-extension")
(4)     def test_install_and_uninstall_pinned(self):
(8)         """
(8)         You should be able to install different versions of the same extension with different
(8)         pinned names and uninstall them with those names.
(8)         """
(8)         NAMES = ["test-1", "test-2"]  # noqa
(8)         assert install_extension(self.pinned_packages[0], pin=NAMES[0])
(8)         assert install_extension(self.pinned_packages[1], pin=NAMES[1])
(8)         extensions = get_app_info()["extensions"]
(8)         assert NAMES[0] in extensions
(8)         assert NAMES[1] in extensions
(8)         assert check_extension(NAMES[0])
(8)         assert check_extension(NAMES[1])
(8)         assert uninstall_extension(NAMES[0])
(8)         assert uninstall_extension(NAMES[1])
(8)         extensions = get_app_info()["extensions"]
(8)         assert NAMES[0] not in extensions
(8)         assert NAMES[1] not in extensions
(8)         assert not check_extension(NAMES[0])
(8)         assert not check_extension(NAMES[1])
(4)     @pytest.mark.skipif(
(8)         platform.system() == "Windows", reason="running npm pack fails on windows CI"
228: (4)                )
229: (4)                def test_install_and_uninstall_pinned_folder(self):
230: (8)                    """
231: (8)                    Same as above test, but installs from a local folder instead of from
npm.
232: (8)                    """
233: (8)                    base_dir = Path(self.tempdir())
234: (8)                    packages = [
235: (12)                       subprocess.run(
236: (16)                           ["npm", "pack", name],  # noqa S603 S607
237: (16)                           stdout=subprocess.PIPE,
238: (16)                           text=True,
239: (16)                           check=True,
240: (16)                           cwd=str(base_dir),
241: (12)                       ).stdout.strip()
242: (12)                       for name in self.pinned_packages
243: (8)                    ]
244: (8)                    shutil.unpack_archive(str(base_dir / packages[0]), str(base_dir /
"1"))
245: (8)                    shutil.unpack_archive(str(base_dir / packages[1]), str(base_dir /
"2"))
246: (8)                    self.pinned_packages = [str(base_dir / "1" / "package"), str(base_dir
/ "2" / "package")]
247: (8)                    self.test_install_and_uninstall_pinned()
248: (4)                def test_link_extension(self):
249: (8)                    path = self.mock_extension
250: (8)                    name = self.pkg_names["extension"]
251: (8)                    link_package(path)
252: (8)                    linked = get_app_info()["linked_packages"]
253: (8)                    assert name not in linked
254: (8)                    assert name in get_app_info()["extensions"]
255: (8)                    assert check_extension(name)
256: (8)                    assert unlink_package(path) is True
257: (8)                    linked = get_app_info()["linked_packages"]
258: (8)                    assert name not in linked
259: (8)                    assert name not in get_app_info()["extensions"]
260: (8)                    assert not check_extension(name)
261: (4)                def test_link_package(self):
262: (8)                    path = self.mock_package
263: (8)                    name = self.pkg_names["package"]
264: (8)                    assert link_package(path) is True
265: (8)                    linked = get_app_info()["linked_packages"]
266: (8)                    assert name in linked
267: (8)                    assert name not in get_app_info()["extensions"]
268: (8)                    assert check_extension(name)
269: (8)                    assert unlink_package(path)
270: (8)                    linked = get_app_info()["linked_packages"]
271: (8)                    assert name not in linked
272: (8)                    assert not check_extension(name)
273: (4)                def test_unlink_package(self):
274: (8)                    target = self.mock_package
275: (8)                    assert link_package(target) is True
276: (8)                    assert unlink_package(target) is True
277: (8)                    linked = get_app_info()["linked_packages"]
278: (8)                    name = self.pkg_names["package"]
279: (8)                    assert name not in linked
280: (8)                    assert not check_extension(name)
281: (4)                def test_list_extensions(self):
282: (8)                    assert install_extension(self.mock_extension) is True
283: (8)                    list_extensions()
284: (4)                def test_app_dir(self):
285: (8)                    app_dir = self.tempdir()
286: (8)                    options = AppOptions(app_dir=app_dir)
287: (8)                    assert install_extension(self.mock_extension, app_options=options) is
True
288: (8)                    path = pjoin(app_dir, "extensions", "*.tgz")
289: (8)                    assert glob.glob(path)
290: (8)                    extensions = get_app_info(app_options=options)["extensions"]
291: (8)                    ext_name = self.pkg_names["extension"]
```

```
(8)                         assert ext_name in extensions
(8)                         assert check_extension(ext_name, app_options=options)
(8)                         assert uninstall_extension(self.pkg_names["extension"],
app_options=options) is True
(8)                         path = pjoin(app_dir, "extensions", "*.tgz")
(8)                         assert not glob.glob(path)
(8)                         extensions = get_app_info(app_options=options)["extensions"]
(8)                         assert ext_name not in extensions
(8)                         assert not check_extension(ext_name, app_options=options)
(8)                         assert link_package(self.mock_package, app_options=options) is True
(8)                         linked = get_app_info(app_options=options)["linked_packages"]
(8)                         pkg_name = self.pkg_names["package"]
(8)                         assert pkg_name in linked
(8)                         assert check_extension(pkg_name, app_options=options)
(8)                         assert unlink_package(self.mock_package, app_options=options) is True
(8)                         linked = get_app_info(app_options=options)["linked_packages"]
(8)                         assert pkg_name not in linked
(8)                         assert not check_extension(pkg_name, app_options=options)
(4)                     def test_app_dir_use_sys_prefix(self):
(8)                         app_dir = self.tempdir()
(8)                         options = AppOptions(app_dir=app_dir)
(8)                         if os.path.exists(self.app_dir):
(12)                            os.removedirs(self.app_dir)
(8)                         assert install_extension(self.mock_extension) is True
(8)                         path = pjoin(app_dir, "extensions", "*.tgz")
(8)                         assert not glob.glob(path)
(8)                         extensions = get_app_info(app_options=options)["extensions"]
(8)                         ext_name = self.pkg_names["extension"]
(8)                         assert ext_name in extensions
(8)                         assert check_extension(ext_name, app_options=options)
(4)                     def test_app_dir_disable_sys_prefix(self):
(8)                         app_dir = self.tempdir()
(8)                         options = AppOptions(app_dir=app_dir, use_sys_dir=False)
(8)                         if os.path.exists(self.app_dir):
(12)                            os.removedirs(self.app_dir)
(8)                         assert install_extension(self.mock_extension) is True
(8)                         path = pjoin(app_dir, "extensions", "*.tgz")
(8)                         assert not glob.glob(path)
(8)                         extensions = get_app_info(app_options=options)["extensions"]
(8)                         ext_name = self.pkg_names["extension"]
(8)                         assert ext_name not in extensions
(8)                         assert not check_extension(ext_name, app_options=options)
(4)                     def test_app_dir_shadowing(self):
(8)                         app_dir = self.tempdir()
(8)                         sys_dir = self.app_dir
(8)                         app_options = AppOptions(app_dir=app_dir)
(8)                         if os.path.exists(sys_dir):
(12)                            os.removedirs(sys_dir)
(8)                         assert install_extension(self.mock_extension) is True
(8)                         sys_path = pjoin(sys_dir, "extensions", "*.tgz")
(8)                         assert glob.glob(sys_path)
(8)                         app_path = pjoin(app_dir, "extensions", "*.tgz")
(8)                         assert not glob.glob(app_path)
(8)                         extensions = get_app_info(app_options=app_options)["extensions"]
(8)                         ext_name = self.pkg_names["extension"]
(8)                         assert ext_name in extensions
(8)                         assert check_extension(ext_name, app_options=app_options)
(8)                         assert install_extension(self.mock_extension, app_options=app_options)
is True
(8)                         assert glob.glob(app_path)
(8)                         extensions = get_app_info(app_options=app_options)["extensions"]
(8)                         assert ext_name in extensions
(8)                         assert check_extension(ext_name, app_options=app_options)
(8)                         assert uninstall_extension(self.pkg_names["extension"],
app_options=app_options) is True
(8)                         assert not glob.glob(app_path)
(8)                         assert glob.glob(sys_path)
(8)                         extensions = get_app_info(app_options=app_options)["extensions"]
(8)                         assert ext_name in extensions
```

```
358: (8)                        assert check_extension(ext_name, app_options=app_options)
359: (8)                        assert uninstall_extension(self.pkg_names["extension"],
app_options=app_options) is True
360: (8)                        assert not glob.glob(app_path)
361: (8)                        assert not glob.glob(sys_path)
362: (8)                        extensions = get_app_info(app_options=app_options)["extensions"]
363: (8)                        assert ext_name not in extensions
364: (8)                        assert not check_extension(ext_name, app_options=app_options)
365: (4)                    @pytest.mark.slow
366: (4)                    def test_build(self):
367: (8)                        assert install_extension(self.mock_extension) is True
368: (8)                        build()
369: (8)                        entry = pjoin(self.app_dir, "staging", "build", "index.out.js")
370: (8)                        with open(entry) as fid:
371: (12)                           data = fid.read()
372: (8)                        assert self.pkg_names["extension"] in data
373: (8)                        entry = pjoin(self.app_dir, "static", "index.out.js")
374: (8)                        with open(entry) as fid:
375: (12)                           data = fid.read()
376: (8)                        assert self.pkg_names["extension"] in data
377: (4)                    @pytest.mark.slow
378: (4)                    @pytest.mark.skipif(not os.path.exists(DEV_DIR), reason="Not in git
checkout")
379: (4)                    def test_build_splice_packages(self):
380: (8)                        app_options = AppOptions(splice_source=True)
381: (8)                        assert install_extension(self.mock_extension) is True
382: (8)                        build(app_options=app_options)
383: (8)                        assert "-spliced" in get_app_version(app_options)
384: (8)                        entry = pjoin(self.app_dir, "staging", "build", "index.out.js")
385: (8)                        with open(entry) as fid:
386: (12)                           data = fid.read()
387: (8)                        assert self.pkg_names["extension"] in data
388: (8)                        entry = pjoin(self.app_dir, "static", "index.out.js")
389: (8)                        with open(entry) as fid:
390: (12)                           data = fid.read()
391: (8)                        assert self.pkg_names["extension"] in data
392: (4)                    @pytest.mark.slow
393: (4)                    def test_build_custom(self):
394: (8)                        assert install_extension(self.mock_extension) is True
395: (8)                        build(name="foo", version="1.0", static_url="bar")
396: (8)                        entry = pjoin(self.app_dir, "static", "index.out.js")
397: (8)                        with open(entry) as fid:
398: (12)                           data = fid.read()
399: (8)                        assert self.pkg_names["extension"] in data
400: (8)                        pkg = pjoin(self.app_dir, "static", "package.json")
401: (8)                        with open(pkg) as fid:
402: (12)                           data = json.load(fid)
403: (8)                        assert data["jupyterlab"]["name"] == "foo"
404: (8)                        assert data["jupyterlab"]["version"] == "1.0"
405: (8)                        assert data["jupyterlab"]["staticUrl"] == "bar"
406: (4)                    @pytest.mark.slow
407: (4)                    def test_build_custom_minimal_core_config(self):
408: (8)                        default_config = CoreConfig()
409: (8)                        core_config = CoreConfig()
410: (8)                        core_config.clear_packages()
411: (8)                        logger = logging.getLogger("jupyterlab_test_logger")
412: (8)                        logger.setLevel("DEBUG")
413: (8)                        app_dir = self.tempdir()
414: (8)                        options = AppOptions(
415: (12)                           app_dir=app_dir,
416: (12)                           core_config=core_config,
417: (12)                           logger=logger,
418: (12)                           use_sys_dir=False,
419: (8)                        )
420: (8)                        extensions = (
421: (12)                           "@jupyterlab/application-extension",
422: (12)                           "@jupyterlab/apputils-extension",
423: (8)                        )
424: (8)                        singletons = (
425: (12)                           "@jupyterlab/application",
426: (12)                           "@jupyterlab/apputils",
427: (12)                           "@jupyterlab/coreutils",
428: (12)                           "@jupyterlab/services",
429: (8)                        )
430: (8)                        for name in extensions:
431: (12)                           semver = default_config.extensions[name]
432: (12)                           core_config.add(name, semver, extension=True)
433: (8)                        for name in singletons:
434: (12)                           semver = default_config.singletons[name]
435: (12)                           core_config.add(name, semver)
436: (8)                        assert install_extension(self.mock_extension, app_options=options) is
True
437: (8)                        build(app_options=options)
438: (8)                        entry = pjoin(app_dir, "static", "index.out.js")
439: (8)                        with open(entry) as fid:
440: (12)                           data = fid.read()
441: (8)                        assert self.pkg_names["extension"] in data
442: (8)                        pkg = pjoin(app_dir, "static", "package.json")
443: (8)                        with open(pkg) as fid:
444: (12)                           data = json.load(fid)
445: (8)                        assert sorted(data["jupyterlab"]["extensions"].keys()) == [
446: (12)                           "@jupyterlab/application-extension",
447: (12)                           "@jupyterlab/apputils-extension",
448: (12)                           "@jupyterlab/mock-extension",
449: (8)                        ]
450: (8)                        assert data["jupyterlab"]["mimeExtensions"] == {}
451: (8)                        for pkg in data["jupyterlab"]["singletonPackages"]:
452: (12)                           if pkg.startswith("@jupyterlab/"):
453: (16)                               assert pkg in singletons
454: (4)                    def test_disable_extension(self):
455: (8)                        options = AppOptions(app_dir=self.tempdir())
456: (8)                        assert install_extension(self.mock_extension, app_options=options) is
True
457: (8)                        assert disable_extension(self.pkg_names["extension"],
app_options=options) is True
458: (8)                        info = get_app_info(app_options=options)
459: (8)                        name = self.pkg_names["extension"]
460: (8)                        assert info["disabled"].get(name) is True
461: (8)                        assert not check_extension(name, app_options=options)
462: (8)                        assert check_extension(name, installed=True, app_options=options)
463: (8)                        assert disable_extension("@jupyterlab/notebook-extension",
app_options=options) is True
464: (8)                        info = get_app_info(app_options=options)
465: (8)                        assert info["disabled"].get("@jupyterlab/notebook-extension") is True
466: (8)                        assert not check_extension("@jupyterlab/notebook-extension",
app_options=options)
467: (8)                        assert check_extension(
468: (12)                           "@jupyterlab/notebook-extension", installed=True,
app_options=options
469: (8)                        )
470: (8)                        assert info["disabled"].get(name) is True
471: (8)                        assert not check_extension(name, app_options=options)
472: (8)                        assert check_extension(name, installed=True, app_options=options)
473: (4)                    def test_enable_extension(self):
474: (8)                        options = AppOptions(app_dir=self.tempdir())
475: (8)                        assert install_extension(self.mock_extension, app_options=options) is
True
476: (8)                        assert disable_extension(self.pkg_names["extension"],
app_options=options) is True
477: (8)                        assert enable_extension(self.pkg_names["extension"],
app_options=options) is True
478: (8)                        info = get_app_info(app_options=options)
479: (8)                        assert "@jupyterlab/notebook-extension" not in info["disabled"]
480: (8)                        name = self.pkg_names["extension"]
481: (8)                        assert info["disabled"].get(name, False) is False
482: (8)                        assert check_extension(name, app_options=options)
483: (8)                        assert disable_extension("@jupyterlab/notebook-extension",
app_options=options) is True
```

```
        assert check_extension(name, app_options=options)
        assert not check_extension("@jupyterlab/notebook-extension", app_options=options)
    @pytest.mark.slow
    def test_build_check(self):
        assert build_check()
        assert install_extension(self.mock_extension) is True
        assert link_package(self.mock_package) is True
        build()
        assert not build_check()
        assert install_extension(self.mock_mimeextension) is True
        assert build_check()
        assert uninstall_extension(self.pkg_names["mimeextension"]) is True
        assert not build_check()
        pkg_path = pjoin(self.mock_extension, "package.json")
        with open(pkg_path) as fid:
            data = json.load(fid)
        with open(pkg_path, "rb") as fid:
            orig = fid.read()
        data["foo"] = "bar"
        with open(pkg_path, "w") as fid:
            json.dump(data, fid)
        assert build_check()
        assert build_check()
        with open(pkg_path, "wb") as fid:
            fid.write(orig)
        assert not build_check()
        pkg_path = pjoin(self.mock_package, "index.js")
        with open(pkg_path, "rb") as fid:
            orig = fid.read()
        with open(pkg_path, "wb") as fid:
            fid.write(orig + b'\nconsole.log("hello");')
        assert build_check()
        assert build_check()
        with open(pkg_path, "wb") as fid:
            fid.write(orig)
        assert not build_check()
    def test_compatibility(self):
        assert _test_overlap("^0.6.0", "^0.6.1")
        assert _test_overlap(">0.1", "0.6")
        assert _test_overlap("~0.5.0", "~0.5.2")
        assert _test_overlap("0.5.2", "^0.5.0")
        assert not _test_overlap("^0.5.0", "^0.6.0")
        assert not _test_overlap("~1.5.0", "^1.6.0")
        assert _test_overlap("*", "0.6") is None
        assert _test_overlap("<0.6", "0.1") is None
        assert _test_overlap("^1 || ^2", "^1")
        assert _test_overlap("^1 || ^2", "^2")
        assert _test_overlap("^1", "^1 || ^2")
        assert _test_overlap("^2", "^1 || ^2")
        assert _test_overlap("^1 || ^2", "^2 || ^3")
        assert not _test_overlap("^1 || ^2", "^3 || ^4")
        assert not _test_overlap("^2", "^1 || ^3")
    def test_compare_ranges(self):
        assert _compare_ranges("^1 || ^2", "^1") == 0
        assert _compare_ranges("^1 || ^2", "^2 || ^3") == 0
        assert _compare_ranges("^1 || ^2", "^3 || ^4") == 1
        assert _compare_ranges("^3 || ^4", "^1 || ^2") == -1
        assert _compare_ranges("^2 || ^3", "^1 || ^4") is None
    def test_install_compatible(self):
        core_data = _get_default_core_data()
        current_app_dep = core_data["dependencies"]["@jupyterlab/application"]
        def _gen_dep(ver):
            return {"dependencies": {"@jupyterlab/application": ver}}
        def _mock_metadata(registry, name, logger):
            assert name == "mockextension"
            return {
                "name": name,
                "versions": {
```

```
(20)                    "0.9.0": _gen_dep(current_app_dep),
(20)                    "1.0.0": _gen_dep(current_app_dep),
(20)                    "1.1.0": _gen_dep(current_app_dep),
(20)                    "2.0.0": _gen_dep("^2000.0.0"),
(20)                    "2.0.0-b0": _gen_dep(current_app_dep),
(20)                    "2.1.0-b0": _gen_dep("^2000.0.0"),
(20)                    "2.1.0": _gen_dep("^2000.0.0"),
(16)                },
(12)            }
(8)         def _mock_extract(self, source, tempdir, *args, **kwargs):
(12)            data = {
(16)                "name": source,
(16)                "version": "2.1.0",
(16)                "jupyterlab": {"extension": True},
(16)                "jupyterlab_extracted_files": ["index.js"],
(12)            }
(12)            data.update(_gen_dep("^2000.0.0"))
(12)            info = {
(16)                "source": source,
(16)                "is_dir": False,
(16)                "data": data,
(16)                "name": source,
(16)                "version": data["version"],
(16)                "filename": "mockextension.tgz",
(16)                "path": pjoin(tempdir, "mockextension.tgz"),
(12)            }
(12)            return info
(8)         class Success(Exception):  # noqa
(12)            pass
(8)         def _mock_install(self, name, *args, **kwargs):
(12)            assert name in ("mockextension", "mockextension@1.1.0")
(12)            if name == "mockextension@1.1.0":
(16)                raise Success()
(12)            return orig_install(self, name, *args, **kwargs)
(8)         p1 = patch.object(commands, "_fetch_package_metadata", _mock_metadata)
(8)         p2 = patch.object(commands._AppHandler, "_extract_package", _mock_extract)
(8)         p3 = patch.object(commands._AppHandler, "_install_extension", _mock_install)
(8)         with p1, p2:
(12)            orig_install = commands._AppHandler._install_extension
(12)            with p3, pytest.raises(Success):
(16)                assert install_extension("mockextension") is True
(4)     def test_update_single(self):
(8)         installed = []
(8)         def _mock_install(self, name, *args, **kwargs):
(12)            installed.append(name[0] + name[1:].split("@")[0])
(12)            return {"name": name, "is_dir": False, "path": "foo/bar/" + name}
(8)         def _mock_latest(self, name):
(12)            return "10000.0.0"
(8)         p1 = patch.object(commands._AppHandler, "_install_extension", _mock_install)
(8)         p2 = patch.object(commands._AppHandler, "_latest_compatible_package_version", _mock_latest)
(8)         assert install_extension(self.mock_extension) is True
(8)         assert install_extension(self.mock_mimeextension) is True
(8)         with p1, p2:
(12)            assert update_extension(self.pkg_names["extension"]) is True
(8)         assert installed == [self.pkg_names["extension"]]
(4)     def test_update_missing_extension(self):
(8)         assert update_extension("foo") is False
(4)     def test_update_multiple(self):
(8)         installed = []
(8)         def _mock_install(self, name, *args, **kwargs):
(12)            installed.append(name[0] + name[1:].split("@")[0])
(12)            return {"name": name, "is_dir": False, "path": "foo/bar/" + name}
(8)         def _mock_latest(self, name):
(12)            return "10000.0.0"
(8)         p1 = patch.object(commands._AppHandler, "_install_extension",
_mock_install)
617: (8)                    p2 = patch.object(commands._AppHandler,
"_latest_compatible_package_version", _mock_latest)
618: (8)                    install_extension(self.mock_extension)
619: (8)                    install_extension(self.mock_mimeextension)
620: (8)                    with p1, p2:
621: (12)                       assert update_extension(self.pkg_names["extension"]) is True
622: (12)                       assert update_extension(self.pkg_names["mimeextension"]) is True
623: (8)                    assert installed == [self.pkg_names["extension"],
self.pkg_names["mimeextension"]]
624: (4)                def test_update_all(self):
625: (8)                    updated = []
626: (8)                    def _mock_update(self, name, *args, **kwargs):
627: (12)                       updated.append(name[0] + name[1:].split("@")[0])
628: (12)                       return True
629: (8)                    original_app_info = commands._AppHandler._get_app_info
630: (8)                    def _mock_app_info(self):
631: (12)                       info = original_app_info(self)
632: (12)                       info["local_extensions"] = []
633: (12)                       return info
634: (8)                    assert install_extension(self.mock_extension) is True
635: (8)                    assert install_extension(self.mock_mimeextension) is True
636: (8)                    p1 = patch.object(commands._AppHandler, "_update_extension",
_mock_update)
637: (8)                    p2 = patch.object(commands._AppHandler, "_get_app_info",
_mock_app_info)
638: (8)                    with p1, p2:
639: (12)                       assert update_extension(None, all_=True) is True
640: (8)                    assert sorted(updated) == [self.pkg_names["extension"],
self.pkg_names["mimeextension"]]
641: (0)            def test_load_extension(jp_serverapp, make_lab_app):
642: (4)                app = make_lab_app()
643: (4)                stderr = sys.stderr
644: (4)                app._link_jupyter_server_extension(jp_serverapp)
645: (4)                app.initialize()
646: (4)                sys.stderr = stderr

----------------------------------------

File 37 - test_registry.py:

1: (0)              """Test yarn registry replacement"""
2: (0)              import logging
3: (0)              import subprocess
4: (0)              from os.path import join as pjoin
5: (0)              from unittest.mock import patch
6: (0)              from jupyterlab import commands
7: (0)              from .test_jupyterlab import AppHandlerTest
8: (0)              class TestAppHandlerRegistry(AppHandlerTest):
9: (4)                  def test_node_not_available(self):
10: (8)                     with patch("jupyterlab.commands.which") as which:
11: (12)                        which.side_effect = ValueError("Command not found")
12: (12)                        logger = logging.getLogger("jupyterlab")
13: (12)                        config = commands._yarn_config(logger)
14: (12)                        which.assert_called_once_with("node")
15: (12)                        self.assertDictEqual(config, {"yarn config": {}, "npm config":
{}})
16: (4)                 def test_yarn_config(self):
17: (8)                     with patch("subprocess.check_output") as check_output:
18: (12)                        yarn_registry = "https://private.yarn/manager"
19: (12)                        check_output.return_value = b"\n".join(
20: (16)                            [
21: (20)                                b'{"type":"info","data":"yarn config"}',
22: (20)                                b'{"type":"inspect","data":{"registry":"'
23: (20)                                + bytes(yarn_registry, "utf-8")
24: (20)                                + b'"}}',
25: (20)                                b'{"type":"info","data":"npm config"}',
26: (20)                                b'{"type":"inspect","data":{"registry":"'
27: (20)                                + bytes(yarn_registry, "utf-8")
28: (20)                                + b'"}}',
29: (16)                            ]
30: (12)                        )
31: (12)                        logger = logging.getLogger("jupyterlab")
32: (12)                        config = commands._yarn_config(logger)
33: (12)                        self.assertDictEqual(
34: (16)                            config,
35: (16)                            {
36: (20)                                "yarn config": {"registry": yarn_registry},
37: (20)                                "npm config": {"registry": yarn_registry},
38: (16)                            },
39: (12)                        )
40: (4)                 def test_yarn_config_failure(self):
41: (8)                     with patch("subprocess.check_output") as check_output:
42: (12)                        check_output.side_effect = subprocess.CalledProcessError(
43: (16)                            1, ["yarn", "config", "list"], b"", stderr=b"yarn config failed."
44: (12)                        )
45: (12)                        logger = logging.getLogger("jupyterlab")
46: (12)                        config = commands._yarn_config(logger)
47: (12)                        self.assertDictEqual(config, {"yarn config": {}, "npm config": {}})
48: (4)                 def test_get_registry(self):
49: (8)                     with patch("subprocess.check_output") as check_output:
50: (12)                        yarn_registry = "https://private.yarn/manager"
51: (12)                        check_output.return_value = b"\n".join(
52: (16)                            [
53: (20)                                b'{"type":"info","data":"yarn config"}',
54: (20)                                b'{"type":"inspect","data":{"registry":"'
55: (20)                                + bytes(yarn_registry, "utf-8")
56: (20)                                + b'"}}',
57: (20)                                b'{"type":"info","data":"npm config"}',
58: (20)                                b'{"type":"inspect","data":{"registry":"'
59: (20)                                + bytes(yarn_registry, "utf-8")
60: (20)                                + b'"}}',
61: (16)                            ]
62: (12)                        )
63: (12)                        handler = commands.AppOptions()
64: (12)                        self.assertEqual(handler.registry, yarn_registry)
65: (4)                 def test_populate_staging(self):
66: (8)                     with patch("subprocess.check_output") as check_output:
67: (12)                        yarn_registry = "https://private.yarn/manager"
68: (12)                        check_output.return_value = b"\n".join(
69: (16)                            [
70: (20)                                b'{"type":"info","data":"yarn config"}',
71: (20)                                b'{"type":"inspect","data":{"registry":"'
72: (20)                                + bytes(yarn_registry, "utf-8")
73: (20)                                + b'"}}',
74: (20)                                b'{"type":"info","data":"npm config"}',
75: (20)                                b'{"type":"inspect","data":{"registry":"'
76: (20)                                + bytes(yarn_registry, "utf-8")
77: (20)                                + b'"}}',
78: (16)                            ]
79: (12)                        )
80: (12)                        staging = pjoin(self.app_dir, "staging")
81: (12)                        handler = commands._AppHandler(commands.AppOptions())
82: (12)                        handler._populate_staging()
83: (12)                        lock_path = pjoin(staging, "yarn.lock")
84: (12)                        with open(lock_path) as f:
85: (16)                            lock = f.read()
86: (12)                        self.assertNotIn(commands.YARN_DEFAULT_REGISTRY, lock)
87: (12)                        self.assertNotIn(yarn_registry, lock)

----------------------------------------

File 38 - __init__.py:

1: (0)              from typing import NamedTuple
2: (0)              class Response(NamedTuple):
```

```
3: (4)                  """Fake tornado response."""
4: (4)                  body: bytes
5: (0)              def fake_client_factory():
6: (4)                  class FakeClient:
7: (8)                      """Fake AsyncHTTPClient
8: (8)                      body can be set in the test to a custom value.
9: (8)                      """
10: (8)                     body = b""
11: (8)                     async def fetch(*args, **kwargs):
12: (12)                        return Response(FakeClient.body)
13: (4)                 return FakeClient

----------------------------------------

File 39 - mock_package.py:

1: (0)              import json
2: (0)              import os.path as osp
3: (0)              HERE = osp.abspath(osp.dirname(__file__))
4: (0)              with open(osp.join(HERE, "package.json")) as fid:
5: (4)                  data = json.load(fid)
6: (0)              def _jupyter_labextension_paths():
7: (4)                  return [{"src": data["jupyterlab"].get("outputDir", "static"), "dest":
data["name"]}]

----------------------------------------

File 40 - setup.py:

1: (0)              import json
2: (0)              import os.path as osp
3: (0)              name = "mock-package"
4: (0)              HERE = osp.abspath(osp.dirname(__file__))
5: (0)              with open(osp.join(HERE, "package.json")) as fid:
6: (4)                  data = json.load(fid)
7: (0)              from setuptools import setup  # noqa
8: (0)              setup(name=name, version=data["version"], py_modules=[name])

----------------------------------------

File 41 - jlab_mock_consumer.py:

1: (0)              import json
2: (0)              import os.path as osp
3: (0)              HERE = osp.abspath(osp.dirname(__file__))
4: (0)              with open(osp.join(HERE, "package.json")) as fid:
5: (4)                  data = json.load(fid)
6: (0)              def _jupyter_labextension_paths():
7: (4)                  return [{"src": data["jupyterlab"].get("outputDir", "static"), "dest":
data["name"]}]

----------------------------------------

File 42 - setup.py:

1: (0)              import json
2: (0)              import os.path as osp
3: (0)              from glob import glob
4: (0)              name = "jlab_mock_consumer"
5: (0)              HERE = osp.abspath(osp.dirname(__file__))
6: (0)              with open(osp.join(HERE, "package.json")) as fid:
7: (4)                  data = json.load(fid)
8: (0)              from setuptools import setup  # noqa
9: (0)              js_name = data["name"]
10: (0)             setup(
11: (4)                 name=name,
12: (4)                 version=data["version"],
13: (4)                 py_modules=[name],
14: (4)                 data_files=[
15: (8)                     (f"share/jupyter/labextensions/{js_name}",
glob("static/package.json")),
16: (8)                     (f"share/jupyter/labextensions/{js_name}/static",
glob("static/static/*")),
17: (4)                 ],
18: (0)             )

----------------------------------------

File 43 - jlab_mock_provider.py:

1: (0)              import json
2: (0)              import os.path as osp
3: (0)              HERE = osp.abspath(osp.dirname(__file__))
4: (0)              with open(osp.join(HERE, "static", "package.json")) as fid:
5: (4)                  data = json.load(fid)
6: (0)              def _jupyter_labextension_paths():
7: (4)                  return [{"src": data["jupyterlab"].get("outputDir", "static"), "dest":
data["name"]}]

----------------------------------------

File 44 - setup.py:

1: (0)              import json
2: (0)              import os.path as osp
3: (0)              from glob import glob
4: (0)              name = "jlab_mock_provider"
5: (0)              HERE = osp.abspath(osp.dirname(__file__))
6: (0)              with open(osp.join(HERE, "package.json")) as fid:
7: (4)                  data = json.load(fid)
8: (0)              from setuptools import setup  # noqa
9: (0)              js_name = data["name"]
10: (0)             setup(
11: (4)                 name=name,
12: (4)                 version=data["version"],
13: (4)                 py_modules=[name],
14: (4)                 data_files=[
15: (8)                     (f"share/jupyter/labextensions/{js_name}",
glob("static/package.json")),
16: (8)                     (f"share/jupyter/labextensions/{js_name}/static",
glob("static/static/*")),
17: (4)                 ],
18: (0)             )

----------------------------------------

File 45 - setup.py:

1: (0)              from os import path
2: (0)              from setuptools import setup
3: (0)              version = "3.0.2"
4: (0)              name = "test_hyphens"
5: (0)              module_name = "test_hyphens"
6: (0)              lab_ext_name = "test-hyphens"
7: (0)              HERE = path.abspath(path.dirname(__file__))
8: (0)              lab_path = path.join(HERE, module_name, "labextension")
9: (0)              data_files_spec = [("share/jupyter/labextensions/" + lab_ext_name, lab_path,
"**")]
10: (0)             setup_args = {"name": name, "version": version, "packages": [module_name]}
11: (0)             try:
12: (4)                 from jupyter_packaging import get_data_files, npm_builder, wrap_installers
13: (4)                 post_develop = npm_builder(build_cmd="build:labextension",
build_dir=lab_path, npm=["jlpm"])
14: (4)                 cmdclass = wrap_installers(post_develop=post_develop)
15: (4)                 setup_args.update(
16: (8)                     {
17: (12)                        "cmdclass": cmdclass,
18: (12)                        "data_files": get_data_files(data_files_spec),
19: (8)                     }
20: (4)                 )
21: (0)             except ImportError:
22: (4)                 pass
23: (0)             setup(**setup_args)

----------------------------------------

File 46 - __init__.py:

1: (0)              def _jupyter_labextension_paths():
2: (4)                  return [{"src": "labextension", "dest": "test-hyphens"}]

----------------------------------------

File 47 - setup.py:

1: (0)              from os import path
2: (0)              from setuptools import setup
3: (0)              version = "3.0.2"
4: (0)              name = "test-hyphens-underscore"
5: (0)              module_name = "test_hyphens_underscore"
6: (0)              lab_ext_name = "test-hyphens-underscore"
7: (0)              HERE = path.abspath(path.dirname(__file__))
8: (0)              lab_path = path.join(HERE, module_name, "labextension")
9: (0)              data_files_spec = [("share/jupyter/labextensions/" + lab_ext_name, lab_path,
"**")]
10: (0)             setup_args = {"name": name, "version": version, "packages": [module_name]}
11: (0)             try:
12: (4)                 from jupyter_packaging import get_data_files, npm_builder, wrap_installers
13: (4)                 post_develop = npm_builder(build_cmd="build:labextension",
build_dir=lab_path, npm=["jlpm"])
14: (4)                 cmdclass = wrap_installers(post_develop=post_develop)
15: (4)                 setup_args.update(
16: (8)                     {
17: (12)                        "cmdclass": cmdclass,
18: (12)                        "data_files": get_data_files(data_files_spec),
19: (8)                     }
20: (4)                 )
21: (0)             except ImportError:
22: (4)                 pass
23: (0)             setup(**setup_args)

----------------------------------------

File 48 - __init__.py:

1: (0)              def _jupyter_labextension_paths():
2: (4)                  return [{"src": "labextension", "dest": "test-hyphens-underscore"}]

----------------------------------------

File 49 - setup.py:

1: (0)              from os import path
2: (0)              from setuptools import setup
3: (0)              version = "3.0.2"
4: (0)              name = "test_no_hyphens"
5: (0)              module_name = "test_no_hyphens"
6: (0)              lab_ext_name = "test_no_hyphens"
7: (0)              HERE = path.abspath(path.dirname(__file__))
8: (0)              lab_path = path.join(HERE, module_name, "labextension")
9: (0)              data_files_spec = [("share/jupyter/labextensions/" + lab_ext_name, lab_path,
"**")]
10: (0)             setup_args = {"name": name, "version": version, "packages": [module_name]}
11: (0)             try:
12: (4)                 from jupyter_packaging import get_data_files, npm_builder, wrap_installers
13: (4)                 post_develop = npm_builder(build_cmd="build:labextension",
build_dir=lab_path, npm=["jlpm"])
```

```
14: (4)                 cmdclass = wrap_installers(post_develop=post_develop)
15: (4)                 setup_args.update({"cmdclass": cmdclass, "data_files":
get_data_files(data_files_spec)})
16: (0)             except ImportError:
17: (4)                 pass
18: (0)             setup(**setup_args)

----------------------------------------

File 50 - __init__.py:

1: (0)              def _jupyter_labextension_paths():
2: (4)                  return [{"src": "labextension", "dest": "test_no_hyphens"}]

----------------------------------------

File 51 -
SANJOYNATHQHENOMENOLOGYGEOMETRIFYINGTRIGONOMETRYCOMBINER_aligner_20_characters_for_pythons_codes.p
y:

1: (0)              import os
2: (0)              from datetime import datetime
3: (0)              def get_file_info(root_folder):
4: (4)                  file_info_list = []
5: (4)                  for root, dirs, files in os.walk(root_folder):
6: (8)                      for file in files:
7: (12)                         try:
8: (16)                             if file.endswith('.py'):
9: (20)                                 file_path = os.path.join(root, file)
10: (20)                                creation_time =
datetime.fromtimestamp(os.path.getctime(file_path))
11: (20)                                modified_time =
datetime.fromtimestamp(os.path.getmtime(file_path))
12: (20)                                file_extension = os.path.splitext(file)[1].lower()
13: (20)                                file_info_list.append([file, file_path, creation_time,
modified_time, file_extension, root])
14: (12)                        except Exception as e:
15: (16)                            print(f"Error processing file {file}: {e}")
16: (4)                 file_info_list.sort(key=lambda x: (x[2], x[3], len(x[0]), x[4]))  # Sort
by creation, modification time, name length, extension
17: (4)                 return file_info_list
18: (0)             def process_file(file_info_list):
19: (4)                 combined_output = []
20: (4)                 for idx, (file_name, file_path, creation_time, modified_time,
file_extension, root) in enumerate(file_info_list):
21: (8)                     with open(file_path, 'r', encoding='utf-8', errors='ignore') as f:
22: (12)                        content = f.read()
23: (12)                        content = "\n".join([line for line in content.split('\n') if
line.strip() and not line.strip().startswith("#")])
24: (12)                        content = content.replace('\t', '    ')
25: (12)                        processed_lines = []
26: (12)                        for i, line in enumerate(content.split('\n')):
27: (16)                            leading_spaces = len(line) - len(line.lstrip(' '))
28: (16)                            line_number_str = f"{i+1}: ({leading_spaces})"
29: (16)                            padding = ' ' * (20 - len(line_number_str))
30: (16)                            processed_line = f"{line_number_str}{padding}{line}"
31: (16)                            processed_lines.append(processed_line)
32: (12)                        content_with_line_numbers = "\n".join(processed_lines)
33: (12)                        combined_output.append(f"File {idx + 1} - {file_name}:\n")
34: (12)                        combined_output.append(content_with_line_numbers)
35: (12)                        combined_output.append("\n" + "-"*40 + "\n")
36: (4)                 return combined_output
37: (0)             root_folder_path = '.'  # Set this to the desired folder
38: (0)             file_info_list = get_file_info(root_folder_path)
39: (0)             combined_output = process_file(file_info_list)
40: (0)             output_file =
'SANJOYNATHQHENOMENOLOGYGEOMETRIFYINGTRIGONOMETRY_combined_python_files_20_chars.txt'
41: (0)             with open(output_file, 'w', encoding='utf-8') as logfile:
42: (4)                 logfile.write("\n".join(combined_output))
```

```
43: (0)              print(f"Processed file info logged to {output_file}")

----------------------------------------
```