ChatGPT

ArchSource Package Manager (arch-pkg-manager-python.py)

The `arch-pkg-manager-python.py` script is a comprehensive package manager for Arch Linux with source-building capabilities. It uses Python classes, functions, and several third-party libraries to manage configurations, detect package sources, resolve dependencies, build packages, and provide both a terminal UI (with **curses**) and a web interface (with **Flask**). The code is organized into sections with classes for each responsibility and a command-line interface at the end. Below is a detailed, line-by-line explanation of each part.

Imports

The script begins with a **shebang** and a docstring explaining its purpose. Then it imports various Python standard libraries and third-party modules:

- **Shebang and Docstring (L0-4):**

```
#!/usr/bin/env python3
"""
ArchSource Package Manager
A comprehensive package manager for Arch Linux with source building
capabilities
"""
```

The `#!/usr/bin/env python3` line tells the shell to run this script with Python 3. The triple-quoted lines are a documentation string.

- **Standard Library Imports (L6-24):**

```
import argparse      # for command-line argument parsing
import asyncio       # for asynchronous programming (running tasks
concurrently)
import configparser  # to parse config files (used for pacman.conf)
import curses        # to build a text-based user interface (terminal UI)
import hashlib       # for hashing (used in caching)
import json          # to encode/decode JSON
import logging       # for logging build operations
import os            # for operating system utilities (e.g., environment
variables)
import queue         # for a thread-safe queue (used in build manager)
```

```python
import shutil        # for file operations (copying, etc.)
import sqlite3       # for a local SQLite database
import subprocess    # to run external commands (e.g., pacman, makepkg)
import sys           # for system-specific parameters and functions (e.g., argv)
import tempfile      # for creating temporary files/directories
import threading    # to create threads (though mainly async is used here)
from dataclasses import asdict, dataclass  # to define simple data containers
from datetime import datetime            # for timestamps
from pathlib import Path                 # for filesystem paths
from typing import Dict, List, Optional, Set, Tuple  # for type hints
```

- **Third-Party Library Imports (L26-29):**

```python
import aiohttp      # async HTTP requests (used for AUR queries)
import git          # GitPython library for Git operations
import yaml         # PyYAML for YAML config files
from git import Repo  # specifically import Repo from GitPython
```

- `aiohttp` allows making asynchronous HTTP requests, used here to talk to the Arch Linux AUR API.
- `git` / `Repo` comes from GitPython and is used to clone and update a Git repository of configuration templates.
- `yaml` is used to read/write YAML configuration files.

Each import supports a piece of functionality in the manager. For example, `curses` is for the terminal UI, `sqlite3` for storing package/build data, `argparse` for parsing user commands, and so on.

Configuration Constants

Some constants define where data is stored and where to find resources:

```python
CONFIG_DIR = Path.home() / ".config" / "archsource"
CACHE_DIR = Path.home() / ".cache" / "archsource"
LOG_DIR = Path.home() / ".local" / "share" / "archsource" / "logs"
DB_PATH = CONFIG_DIR / "packages.db"
CONFIG_FILE = CONFIG_DIR / "config.yaml"
GITHUB_REPO_URL = "https://github.com/user/archsource-configs.git"
```

- `CONFIG_DIR` (e.g. `~/.config/archsource`) is where user-specific configuration and templates are stored.
- `CACHE_DIR` (e.g. `~/.cache/archsource`) is used for caching builds or downloads.
- `LOG_DIR` (e.g. `~/.local/share/archsource/logs`) holds log files for builds.

- `DB_PATH` is the path to an SQLite database file (`packages.db`) inside `CONFIG_DIR`.
- `CONFIG_FILE` is the YAML file (`config.yaml`) for main settings.
- `GITHUB_REPO_URL` is a placeholder URL for syncing configuration templates with a GitHub repo.

These constants use `pathlib.Path` to build cross-platform file paths [1].

Data Classes: `PackageInfo` and `BuildResult`

Two **dataclasses** define simple data structures:

```
@dataclass
class PackageInfo:
    name: str
    version: str
    description: str
    dependencies: List[str]
    source: str   # 'official' or 'aur'
    build_time: Optional[datetime] = None
    custom_makepkg_conf: Optional[str] = None
```

- `PackageInfo` holds metadata about a package (name, version, description, list of dependencies, and whether it's from the official repos or AUR). It also can store when it was built and if a custom `makepkg.conf` was used. The `@dataclass` decorator auto-generates an `__init__` and other methods [2].

```
@dataclass
class BuildResult:
    package: str
    success: bool
    log_file: str
    error_msg: Optional[str] = None
    build_time: float = 0.0
```

- `BuildResult` holds the outcome of building a package: its name, whether it succeeded, where the log file is, any error message, and the build duration [3].

These data classes are used throughout to pass around package metadata and build outcomes in a structured way.

DatabaseManager Class

The `DatabaseManager` class (lines 60–103) manages an SQLite database that tracks known packages and their build logs [4]. Its responsibilities include initializing the DB tables and inserting or fetching package records.

Key parts: - **Initialization (L63-65):**

```
def __init__(self):
    self.db_path = DB_PATH
    self.init_database()
```

The constructor sets the database path and calls `init_database()` to set up the file and tables.

- **Database Setup (L67-75):**

  ```
  conn = sqlite3.connect(self.db_path)
  cursor = conn.cursor()
  cursor.execute("""
      CREATE TABLE IF NOT EXISTS packages (
          name TEXT PRIMARY KEY,
          version TEXT,
          description TEXT,
          dependencies TEXT,
          source TEXT,
          build_time TEXT,
          custom_makepkg_conf TEXT,
          last_updated TEXT
      )
  """)
  cursor.execute("""
      CREATE TABLE IF NOT EXISTS build_logs (
          id INTEGER PRIMARY KEY AUTOINCREMENT,
          package_name TEXT,
          build_time TEXT,
          success BOOLEAN,
          log_file TEXT,
          error_msg TEXT,
          FOREIGN KEY(package_name) REFERENCES packages(name)
      )
  """)
  conn.commit()
  conn.close()
  ```

  This code (inside `init_database()`) ensures two tables exist:

- `packages` stores basic info for each package (dependencies are stored as a JSON string) [5].

- `build_logs` records each build attempt with a timestamp, success flag, log file path, and any error. If tables do not exist, they are created.

- **Saving a Package (L111-120):**

```
def save_package(self, package: PackageInfo):
    conn = sqlite3.connect(self.db_path)
    cursor = conn.cursor()
    cursor.execute("""
        INSERT OR REPLACE INTO packages
        (name, version, description, dependencies, source, build_time,
custom_makepkg_conf, last_updated)
        VALUES (?, ?, ?, ?, ?, ?, ?, ?)
    """, (
        package.name,
        package.version,
        package.description,
        json.dumps(package.dependencies),
        package.source,
        package.build_time.isoformat() if package.build_time else None,
        package.custom_makepkg_conf,
        datetime.now().isoformat(),
    ))
    conn.commit()
    conn.close()
```

The `save_package` method takes a `PackageInfo` object and writes it to the database (using an `INSERT OR REPLACE` query). Dependencies list is converted to JSON text with `json.dumps()` [6]. It also updates the `last_updated` timestamp.

- **Retrieving a Package (L133-142):**

```
def get_package(self, name: str) -> Optional[PackageInfo]:
    conn = sqlite3.connect(self.db_path)
    cursor = conn.cursor()
    cursor.execute("SELECT * FROM packages WHERE name = ?", (name,))
    row = cursor.fetchone()
    conn.close()
    if not row:
        return None
    return PackageInfo(
        name=row[0],
        version=row[1],
        description=row[2],
        dependencies=json.loads(row[3]) if row[3] else [],
        source=row[4],
        build_time=datetime.fromisoformat(row[5]) if row[5] else None,
        custom_makepkg_conf=row[6],
    )
```

`get_package` looks up a package by name. If found, it constructs and returns a `PackageInfo` object, converting the JSON dependencies back to a Python list and parsing the `build_time` string to a `datetime` ⑦. If the name isn't in the DB, it returns `None`.

- **Logging a Build (L155-163):**

```
def log_build(self, result: BuildResult):
    conn = sqlite3.connect(self.db_path)
    cursor = conn.cursor()
    cursor.execute("""
        INSERT INTO build_logs (package_name, build_time, success,
log_file, error_msg)
        VALUES (?, ?, ?, ?, ?)
    """, (
        result.package,
        datetime.now().isoformat(),
        result.success,
        result.log_file,
        result.error_msg,
    ))
    conn.commit()
    conn.close()
```

`log_build` saves a `BuildResult` to the `build_logs` table. It records which package was built, the current time, whether it succeeded, the log file path, and any error message ⑧.

In summary, `DatabaseManager` abstracts all database operations: creating tables, saving package info, retrieving it, and storing build logs.

ConfigManager Class

The `ConfigManager` class (lines 179–285) handles configuration files, templates, and syncing with GitHub ⑨ ⑩. It ensures defaults are set and manages Git synchronization of templates.

Key parts: - **Initialization (L182-187):**

```
def __init__(self):
    self.config_dir = CONFIG_DIR
    self.templates_dir = self.config_dir / "templates"
    self.templates_dir.mkdir(parents=True, exist_ok=True)
    self.load_config()
    self.init_git_repo()
```

It sets `config_dir` and a subfolder `templates_dir`. It creates the `templates` directory if it doesn't exist, then loads the main config and initializes the Git repository.

- **Loading Config (L189-197):**

```
def load_config(self):
    if CONFIG_FILE.exists():
        with open(CONFIG_FILE, "r") as f:
            self.config = yaml.safe_load(f)
    else:
        self.config = {
            "github_sync": True,
            "auto_sign_keys": True,
            "parallel_builds": 4,
            "cache_size": "1GB",
            "makepkg_flags": ["-s", "-i", "--noconfirm"],
        }
        self.save_config()
```

`load_config` reads the YAML file (if it exists) into `self.config`. If not, it creates a default config (enabling GitHub sync and auto key signing, setting 4 parallel builds, etc.) and saves it [11].

- **Saving Config (L204-207):**

```
def save_config(self):
    with open(CONFIG_FILE, "w") as f:
        yaml.dump(self.config, f, default_flow_style=False)
```

Writes the `self.config` dictionary out to `config.yaml` in a readable format.

- **Initializing Git Repo (L209-218):**

```
def init_git_repo(self):
    git_dir = self.config_dir / ".git"
    if not git_dir.exists() and self.config.get("github_sync"):
        try:
            self.repo = Repo.clone_from(GITHUB_REPO_URL, self.config_dir)
        except:
            self.repo = Repo.init(self.config_dir)
    elif git_dir.exists():
        self.repo = Repo(self.config_dir)
    else:
        self.repo = None
```

`init_git_repo` ensures there is a Git repository at `config_dir` if GitHub syncing is enabled. It tries to clone from `GITHUB_REPO_URL`; if that fails (e.g., first time or no internet), it initializes an empty repo. If `.git` already exists, it just uses it [12].

- **Generating makepkg.conf Template (L222-258):**

```
def generate_template_makepkg_conf(self, package_name: str) -> str:
    template = f"""# Custom makepkg.conf for {package_name}
# Generated on {datetime.now().strftime('%Y-%m-%d %H:%M:%S')}

# Compiler and linker flags
CFLAGS="-march=native -mtune=native -O2 -pipe -fno-plt -fexceptions"
CXXFLAGS="$CFLAGS -Wp,-D_FORTIFY_SOURCE=2 -Wformat -Werror=format-security"
LDFLAGS="-Wl,-O1,--sort-common,--as-needed,-z,relro,-z,now"
RUSTFLAGS="-C opt-level=2 -C target-cpu=native"

# Build environment
MAKEFLAGS="-j$(nproc)"
BUILDDIR=/tmp/makepkg

# Package options
OPTIONS=(strip docs !libtool !staticlibs emptydirs zipman purge !debug)

# Compression for final packages
COMPRESSGZ=(gzip -c -f -n)
COMPRESSBZ2=(bzip2 -c -f)
COMPRESSXZ=(xz -c -z -T 0 -)
COMPRESSLRZ=(lrzip -q)
COMPRESSLZO=(lzop -q)
COMPRESSZ=(compress -c -f)
COMPRESSLZ4=(lz4 -q)
COMPRESSLZ=(lzip -c -f)

PKGEXT='.pkg.tar.xz'
SRCEXT='.src.tar.gz'
"""
    template_path = self.templates_dir / f"{package_name}.conf"
    with open(template_path, "w") as f:
        f.write(template)
    return str(template_path)
```

This method creates a custom `makepkg.conf` template for the given package name. It writes a shell-config file with optimized flags (e.g., `-march=native`) and standard compression options [13] [14]. The file is saved as `<package_name>.conf` in the `templates_dir`, and the path to this file is returned.

- **Retrieving a Custom Conf (L260-263):**

  ```
  def get_makepkg_conf(self, package_name: str) -> Optional[str]:
      conf_path = self.templates_dir / f"{package_name}.conf"
      return str(conf_path) if conf_path.exists() else None
  ```

  If a custom `makepkg.conf` exists for a package (in the `templates` folder), this returns its filepath; otherwise it returns `None`.

- **Syncing with GitHub (L265-273):**

  ```
  def sync_with_github(self):
      if not self.repo or not self.config.get("github_sync"):
          return
      try:
          origin = self.repo.remotes.origin
          origin.pull()
          self.repo.git.add(A=True)
          if self.repo.is_dirty():
              self.repo.index.commit(f"Auto-sync templates:
  {datetime.now()}")
              origin.push()
      except Exception as e:
          logging.error(f"GitHub sync failed: {e}")
  ```

  This method synchronizes the local `templates` directory with the GitHub repository. It pulls updates, stages all files, and if there are changes it commits and pushes them. Errors are logged [15].

Together, `ConfigManager` handles user settings, template files, and optional Git synchronization (e.g. to share configs via GitHub).

PackageDetector Class

The `PackageDetector` (L287–321) figures out whether a given package is in the official Arch repos or only in the AUR [16] [17]:

```
class PackageDetector:
    def __init__(self):
        self.official_repos = ["core", "extra", "community", "multilib"]

    async def detect_source(self, package_name: str) -> str:
        """Detect if package is from official repos or AUR"""
        # Check official repos first
```

```
        try:
            result = subprocess.run(
                ["pacman", "-Si", package_name],
                capture_output=True,
                text=True,
                timeout=10,
            )
            if result.returncode == 0:
                return "official"
        except subprocess.TimeoutExpired:
            pass

        # Check AUR
        async with aiohttp.ClientSession() as session:
            try:
                async with session.get(
                    f"https://aur.archlinux.org/rpc/?v=5&type=info&arg[]={package_name}"
                ) as response:
                    data = await response.json()
                    if data.get("resultcount", 0) > 0:
                        return "aur"
            except:
                pass

        return "unknown"
```

- It first runs `pacman -Si <package>` (a system command) synchronously to see if the package exists in official repos. A return code of 0 means “found” [18] .
- If not found or if the call times out, it then makes an HTTP GET request to the AUR RPC API. If the AUR returns at least one result, it assumes the package is from AUR [19] .
- If neither check succeeds, it returns `"unknown"`.

This asynchronous method can be awaited in the code to decide how to handle the package (official vs AUR). It uses `aiohttp` for the web call.

DependencyResolver Class

`DependencyResolver` (L324–418) uses `PackageDetector` to recursively find all dependencies of a package [20] [21] :

- **Initialization (L327-330):**

```
def __init__(self, db_manager: DatabaseManager):
    self.db_manager = db_manager
    self.detector = PackageDetector()
```

It keeps a reference to the `DatabaseManager` (though not used here) and a `PackageDetector` to check package sources.

- **Resolving All Dependencies (L331-340):**

```
async def resolve_dependencies(self, package_name: str) -> List[str]:
    resolved = []
    to_resolve = [package_name]
    resolved_set = set()

    while to_resolve:
        current = to_resolve.pop(0)
        if current in resolved_set:
            continue
        resolved_set.add(current)
        deps = await self.get_package_dependencies(current)
        for dep in deps:
            if dep not in resolved_set:
                to_resolve.append(dep)
        resolved.append(current)

    return resolved[1:]  # Exclude the original package
```

This method does a breadth-first resolution: start with the target package, then get its direct dependencies, add them to a queue (`to_resolve`), and continue until no new packages appear. It returns a list of dependencies excluding the original package.

- **Getting Direct Dependencies (L352-361):**

```
async def get_package_dependencies(self, package_name: str) -> List[str]:
    source = await self.detector.detect_source(package_name)
    if source == "official":
        return await self.get_official_deps(package_name)
    elif source == "aur":
        return await self.get_aur_deps(package_name)
    return []
```

This checks if a package is official or AUR and calls the appropriate method.

- **Official Repos Dependencies (L364-374):**

```python
async def get_official_deps(self, package_name: str) -> List[str]:
    try:
        result = subprocess.run(
            ["pacman", "-Si", package_name],
            capture_output=True,
            text=True,
            timeout=10,
        )
        if result.returncode == 0:
            lines = result.stdout.split("\n")
            for line in lines:
                if line.startswith("Depends On"):
                    deps_str = line.split(":", 1)[1].strip()
                    if deps_str == "None":
                        return []
                    deps = [dep.split(">=")[0].split("=")[0].split("<")[0].strip()
                            for dep in deps_str.split()]
                    return [dep for dep in deps if dep and not dep.startswith("(")]
    except:
        pass
    return []
```

It runs `pacman -Si` again, then parses the output looking for the line starting with `"Depends On"`. It splits that list, removes any version constraints (`>=, =, <`) and empty entries, returning the list of dependency names [22]. If something fails or there are no deps, it returns an empty list.

- **AUR Dependencies (L394-402):**

```python
async def get_aur_deps(self, package_name: str) -> List[str]:
    async with aiohttp.ClientSession() as session:
        try:
            async with session.get(
                f"https://aur.archlinux.org/rpc/?v=5&type=info&arg[]={package_name}"
            ) as response:
                data = await response.json()
                if data.get("resultcount", 0) > 0:
                    pkg_info = data["results"][0]
                    deps = pkg_info.get("Depends", []) + pkg_info.get("MakeDepends", [])
                    return [dep.split(">=")[0].split("=")[0].split("<")[0].strip()
                            for dep in deps if dep]
        except:
            pass
    return []
```

Similar to the official case, but uses the AUR JSON: it gets the "Depends" and "MakeDepends" lists from the AUR response and cleans them up in the same way [23].

In summary, `DependencyResolver` can take a package name and return all packages that need to be built or installed before it.

KeyManager Class

`KeyManager` (L420–455) is a small helper for managing GPG keys when building packages:

```
class KeyManager:
    """Manages GPG key signing for packages"""

    def __init__(self):
        self.signed_keys = set()

    def auto_sign_key(self, key_id: str) -> bool:
        """Automatically sign a GPG key"""
        if key_id in self.signed_keys:
            return True
        try:
            subprocess.run(["gpg", "--recv-keys", key_id], check=True,
capture_output=True, timeout=30)
            subprocess.run(["gpg", "--lsign-key", key_id], input="y\n",
text=True,
                           check=True, capture_output=True, timeout=30)
            self.signed_keys.add(key_id)
            return True
        except (subprocess.CalledProcessError, subprocess.TimeoutExpired):
            return False
```

- It keeps a set of keys already signed (`self.signed_keys`).
- `auto_sign_key(key_id)` runs `gpg --recv-keys` to fetch the public key, then `gpg --lsign-key` to locally sign it (answering "y" automatically). If successful, it records the key ID and returns `True`. If an error occurs (invalid key or timeout), it returns `False` [24] [25].

This is used during builds to automatically trust needed keys.

BuildManager Class

The `BuildManager` (L457–628) handles compiling packages. It supports building official repo packages via `pkgctl repo clone` and AUR packages via `git clone`, then running `makepkg`. It logs output and returns `BuildResult` objects.

- **Initialization (L460-468):**

```
def __init__(self, config_manager: ConfigManager, db_manager:
DatabaseManager):
    self.config_manager = config_manager
    self.db_manager = db_manager
    self.key_manager = KeyManager()
    self.build_queue = queue.Queue()
    self.active_builds = {}
    LOG_DIR.mkdir(parents=True, exist_ok=True)
    self.setup_logging()
```

The constructor stores references to the config and DB managers, creates a `KeyManager`, and initializes a queue and dict for managing build tasks (though they are not used in the shown code). It also ensures the log directory exists and sets up logging (writing to both a file and stdout) [26] .

- **Setting up Logging (L472-480):**

```
def setup_logging(self):
    log_file = LOG_DIR / f"builds_{datetime.now().strftime('%Y%m%d')}.log"
    logging.basicConfig(
        level=logging.INFO,
        format="%(asctime)s - %(levelname)s - %(message)s",
        handlers=[logging.FileHandler(log_file), logging.StreamHandler()],
    )
```

This configures Python's `logging` to log INFO-level messages both to a dated log file and to the console.

- **Building a Single Package (L481-490):**

```
async def build_package(self, package_name: str, source: str) ->
BuildResult:
    start_time = datetime.now()
    log_file = (LOG_DIR / f"{package_name}
_{start_time.strftime('%Y%m%d_%H%M%S')}.log")
    try:
        if source == "official":
            result = await self.build_official_package(package_name,
log_file)
        else:
            result = await self.build_aur_package(package_name, log_file)
        build_time = (datetime.now() - start_time).total_seconds()
        result.build_time = build_time
        self.db_manager.log_build(result)
        return result
    except Exception as e:
        error_result = BuildResult(
            package=package_name,
            success=False,
            log_file=str(log_file),
            error_msg=str(e),
            build_time=(datetime.now() - start_time).total_seconds(),
        )
        self.db_manager.log_build(error_result)
        return error_result
```

The `build_package` method orchestrates a build: it records the start time and creates a log file path under `LOG_DIR`. Depending on `source`, it calls either `build_official_package` or `build_aur_package` (both async). After the build completes, it computes the duration, logs the result to the database, and returns a `BuildResult`. If an exception occurs, it creates a failed `BuildResult` with the error message.

- **Building Official Repo Packages (L516-528):**

```
async def build_official_package(self, package_name: str, log_file: Path) -
> BuildResult:
    build_dir = CACHE_DIR / "builds" / package_name
    build_dir.mkdir(parents=True, exist_ok=True)
    try:
        cmd = ["pkgctl", "repo", "clone", package_name]
        result = await self.run_command(cmd, cwd=build_dir,
log_file=log_file)
        if result.returncode != 0:
            return BuildResult(package=package_name, success=False,
log_file=str(log_file),
                               error_msg="Failed to clone package
repository")
        pkg_dir = build_dir / package_name
        makepkg_conf = self.config_manager.get_makepkg_conf(package_name)
        env = os.environ.copy()
        if makepkg_conf:
            env["MAKEPKG_CONF"] = makepkg_conf
        makepkg_flags = self.config_manager.config.get("makepkg_flags", ["-s", "-i"])
        cmd = ["makepkg"] + makepkg_flags
        result = await self.run_command(cmd, cwd=pkg_dir, log_file=log_file, env=env)
        success = (result.returncode == 0)
        return BuildResult(package=package_name, success=success, log_file=str(log_file),
                           error_msg=None if success else "Build failed")
    except Exception as e:
        return BuildResult(package=package_name, success=False, log_file=str(log_file),
                           error_msg=str(e))
```

For official packages, it:

- Creates a build directory in the cache.
- Runs `pkgctl repo clone <name>` to get the source code. If that fails (non-zero return), it returns an error.
- Applies any custom `makepkg.conf` (if defined) by setting the `MAKEPKG_CONF` environment variable.
- Runs `makepkg` with flags from config (default `-s -i --noconfirm`).
- Returns success or failure in a `BuildResult` [27] [28].

- **Building AUR Packages (L570-579):**

```
async def build_aur_package(self, package_name: str, log_file: Path) -> BuildResult:
    build_dir = CACHE_DIR / "builds" / package_name
    build_dir.mkdir(parents=True, exist_ok=True)
    try:
        aur_url = f"https://aur.archlinux.org/{package_name}.git"
        cmd = ["git", "clone", aur_url, package_name]
        result = await self.run_command(cmd, cwd=build_dir.parent, log_file=log_file)
        if result.returncode != 0:
            return BuildResult(package=package_name, success=False, log_file=str(log_file),
                               error_msg="Failed to clone AUR package")
        makepkg_conf = self.config_manager.get_makepkg_conf(package_name)
        env = os.environ.copy()
        if makepkg_conf:
            env["MAKEPKG_CONF"] = makepkg_conf
        makepkg_flags = self.config_manager.config.get("makepkg_flags", ["-s", "-i"])
        cmd = ["makepkg"] + makepkg_flags
        result = await self.run_command(cmd, cwd=build_dir,
log_file=log_file, env=env)
        success = (result.returncode == 0)
        return BuildResult(package=package_name, success=success,
log_file=str(log_file),
                           error_msg=None if success else "Build failed")
    except Exception as e:
        return BuildResult(package=package_name, success=False,
log_file=str(log_file),
                           error_msg=str(e))
```

AUR builds are similar, but it first clones the package's Git repo from the AUR, then runs `makepkg`. The log file captures all output. Return codes determine success [29] [30].

- **Running Commands (L622-631):**

```
async def run_command(self, cmd: List[str], cwd: Path, log_file: Path, env:
Optional[Dict] = None) -> subprocess.CompletedProcess:
    with open(log_file, "a") as f:
        f.write(f"Running: {' '.join(cmd)}\n")
        f.write(f"Working directory: {cwd}\n")
        f.write("-" * 50 + "\n")
    process = await asyncio.create_subprocess_exec(
        *cmd,
        cwd=cwd,
        env=env,
        stdout=asyncio.subprocess.PIPE,
        stderr=asyncio.subprocess.STDOUT,
    )
    with open(log_file, "a") as f:
        while True:
            line = await process.stdout.readline()
            if not line:
                break
            line_str = line.decode("utf-8", errors="ignore")
            f.write(line_str)
            f.flush()
    await process.wait()
    return subprocess.CompletedProcess(cmd, process.returncode)
```

This helper runs a shell command asynchronously, writes the command and output to the log file, and returns a `CompletedProcess` with the exit code [31] [32]. It uses `asyncio` to allow multiple builds concurrently, streaming each line to the log.

Overall, `BuildManager` automates the building of packages (official or AUR), logging everything and respecting configuration settings.

NCursesUI Class

The `NCursesUI` class (L655–728) provides a text-based user interface using the **curses** library. It's a simple menu-driven UI (though the code shown mostly lays out structure, with some placeholder parts):

- **Initialization (L658-662):**

```
def __init__(self, package_manager):
    self.package_manager = package_manager
    self.current_menu = "main"
    self.selected_packages = []
```

  It stores a reference to the `ArchSourceManager` and tracks which screen is active.

- **Run Loop (L663-671):**

```
def run(self):
    curses.wrapper(self._main_loop)

def _main_loop(self, stdscr):
    curses.curs_set(0)  # Hide cursor
    stdscr.nodelay(1)   # Non-blocking input
    stdscr.timeout(100) # Refresh every 100ms
    while True:
        stdscr.clear()
        height, width = stdscr.getmaxyx()
        header = "ArchSource Package Manager"
        stdscr.addstr(0, (width - len(header)) // 2, header, curses.A_BOLD)
        stdscr.addstr(1, 0, "=" * width)
        if self.current_menu == "main":
            self._draw_main_menu(stdscr)
        elif self.current_menu == "search":
            self._draw_search_interface(stdscr)
        elif self.current_menu == "deps":
            self._draw_dependency_tree(stdscr)
        stdscr.refresh()
        key = stdscr.getch()
        if key == ord("q"):
            break
        elif key == ord("s"):
            self.current_menu = "search"
        elif key == ord("d"):
            self.current_menu = "deps"
        elif key == ord("m"):
            self.current_menu = "main"
```

`run()` calls `curses.wrapper` which initializes curses and calls `_main_loop`. The loop hides the cursor, updates the screen, draws a header, and then calls one of three drawing methods depending on `self.current_menu`. It listens for key presses: `'q'` quits, `'s'` and `'d'` switch menus, etc.

- **Main Menu (L702-712):**

```
def _draw_main_menu(self, stdscr):
    menu_items = [
        "s - Search packages",
        "d - View dependency tree",
        "i - Install packages",
        "b - Build packages",
        "c - Configuration",
        "l - View logs",
        "q - Quit",
    ]
    start_y = 3
    for i, item in enumerate(menu_items):
        stdscr.addstr(start_y + i, 2, item)
```

This draws a list of commands the user can do. (In the code, only `'s'`, `'d'`, and `'q'` are actually handled; the others are placeholders.)

- **Search Interface (L718-722):**

```
def _draw_search_interface(self, stdscr):
    stdscr.addstr(3, 2, "Search packages (ESC to return):")
    stdscr.addstr(4, 2, "> ")
    # (Further implementation would go here)
```

It shows a prompt for searching packages. The full search logic isn't implemented in the snippet; this just draws the text.

- **Dependency Tree (L724-727):**

```
def _draw_dependency_tree(self, stdscr):
    stdscr.addstr(3, 2, "Dependency Tree (ESC to return):")
    # (Visualization would be implemented here)
```

It displays a header; actual tree-drawing logic would be added in a complete implementation.

In summary, `NCursesUI` sets up a simple interactive terminal interface, but much of its detailed functionality is left as placeholders. It ties into the `ArchSourceManager` for actual actions like search or install (not fully shown).

ArchSourceManager Class

`ArchSourceManager` (L730–803) is the core “brain” of the application. It holds instances of the above managers and provides high-level methods for installing packages, generating templates, syncing configs, and running UIs.

- **Initialization (L733-740):**

```
def __init__(self):
    self.db_manager = DatabaseManager()
    self.config_manager = ConfigManager()
    self.build_manager = BuildManager(self.config_manager, self.db_manager)
    self.dependency_resolver = DependencyResolver(self.db_manager)
    self.detector = PackageDetector()
    self.ui = NCursesUI(self)
```

It creates one instance of each helper: database, config, build, dependency resolver, package detector, and the text UI. This means these components share state (like the same database) and can interact.

- **Installing Packages (L741-751):**

```
async def install_packages(self, packages: List[str], batch: bool = False):
    all_packages = []
    for package in packages:
        source = await self.detector.detect_source(package)
        deps = await self.dependency_resolver.resolve_dependencies(package)
        for dep in deps:
            dep_source = await self.detector.detect_source(dep)
            all_packages.append((dep, dep_source))
        all_packages.append((package, source))
    # Remove duplicates while preserving order...
```

This method takes a list of package names to install. For each package, it:

- Detects its source (official or AUR).
- Resolves its dependencies (getting a list of all deps).

- Appends all dependencies and then the original package to a list `all_packages` with their sources.

- **Deduplication & Build Execution (L757-767):**

```
    # Remove duplicates while preserving order
    seen = set()
    unique_packages = []
    for pkg, src in all_packages:
        if pkg not in seen:
            seen.add(pkg)
            unique_packages.append((pkg, src))
    semaphore =
asyncio.Semaphore(self.config_manager.config.get("parallel_builds", 4))
    async def build_with_semaphore(pkg_info):
        async with semaphore:
            return await self.build_manager.build_package(pkg_info[0],
pkg_info[1])
```

It then deduplicates `all_packages` (so each package is built once). It creates an `asyncio.Semaphore` to limit concurrent builds to the `parallel_builds` setting. It defines a helper coroutine `build_with_semaphore` that acquires the semaphore before building.

- **Batch vs Sequential (L775-784):**

```
    if batch:
        tasks = [build_with_semaphore(pkg_info) for pkg_info in
unique_packages]
        results = await asyncio.gather(*tasks, return_exceptions=True)
    else:
        results = []
        for pkg_info in unique_packages:
            result = await build_with_semaphore(pkg_info)
            results.append(result)
            if not result.success:
                print(f"Failed to build {pkg_info[0]}, stopping
installation")
                break
    return results
```

If `batch` is `True`, it starts all builds in parallel and waits for all to finish. If `False`, it builds them one by one, stopping if one fails. It returns a list of `BuildResult` objects.

- **Generating a Template (L793-796):**

```
def generate_template(self, package_name: str) -> str:
    return self.config_manager.generate_template_makepkg_conf(package_name)
```

This simply calls the `ConfigManager` method to create a `makepkg.conf` template for the package.

- **Syncing Configs (L797-800):**

  ```
  def sync_configs(self):
      self.config_manager.sync_with_github()
  ```

  Calls GitHub syncing.

- **Running the UI (L801-803):**

  ```
  def run_ui(self):
      self.ui.run()
  ```

  Launches the curses UI.

In effect, `ArchSourceManager` glues together all parts. Other code (CLI or web endpoints) calls its methods to perform actions like install or template generation.

Web UI (Flask) Functions

The script provides two functions to create web interfaces for managing templates and builds: `create_web_ui()` and `create_enhanced_web_ui()`. Both return a Flask `app`, but `create_enhanced_web_ui` is more fully featured with the HTML/JS template included.

Basic Web UI (`create_web_ui`, L806–853)

This sets up a Flask app with endpoints to list, get, and save templates:

```
def create_web_ui():
    from flask import Flask, jsonify, render_template, request
    app = Flask(__name__)
    manager = ArchSourceManager()

    @app.route("/")
    def index():
        return render_template("index.html")
```

```
    @app.route("/api/templates")
    def list_templates():
        templates_dir = manager.config_manager.templates_dir
        templates = []
        for conf_file in templates_dir.glob("*.conf"):
            templates.append({
                "name": conf_file.stem,
                "path": str(conf_file),
                "modified": conf_file.stat().st_mtime,
            })
        return jsonify(templates)

    @app.route("/api/templates/<name>")
    def get_template(name):
        conf_path = manager.config_manager.templates_dir / f"{name}.conf"
        if conf_path.exists():
            with open(conf_path, "r") as f:
                return jsonify({"content": f.read()})
        return jsonify({"error": "Template not found"}), 404

    @app.route("/api/templates/<name>", methods=["POST"])
    def save_template(name):
        content = request.json.get("content", "")
        conf_path = manager.config_manager.templates_dir / f"{name}.conf"
        with open(conf_path, "w") as f:
            f.write(content)
        if manager.config_manager.config.get("github_sync"):
            manager.sync_configs()
        return jsonify({"success": True})

    return app
```

- It uses Flask to define routes under `/api/templates`.
- `list_templates` returns a JSON list of existing templates (name and metadata).
- `get_template` returns the content of a named template.
- `save_template` writes posted content back to file and triggers a Git sync if enabled [33] [34].

This app serves an HTML page (`index.html`) for the root, which would contain the front-end to interact with the API (not shown in the code). It is a simple REST API for templates.

Enhanced Web UI (`create_enhanced_web_ui`, L1535–1675)

This function creates a more complete web interface with HTML, JavaScript, and caching:

```
def create_enhanced_web_ui():
    from flask import Flask, jsonify, render_template_string, request
```

```python
    app = Flask(__name__)
    manager = ArchSourceManager()
    cache_manager = CacheManager(CACHE_DIR)

    @app.route("/")
    def index():
        return render_template_string(WebUITemplates.INDEX_TEMPLATE)

    @app.route("/api/templates")
    def list_templates():
        ...
    @app.route("/api/templates/<name>")
    def get_template(name):
        ...
    @app.route("/api/templates/<name>", methods=["POST"])
    def save_template(name):
        ...
    @app.route("/api/templates/<name>", methods=["DELETE"])
    def delete_template(name):
        ...
    @app.route("/api/templates/<name>/generate", methods=["POST"])
    def generate_template(name):
        ...
    @app.route("/api/packages/search")
    def search_packages():
        ...
    @app.route("/api/build-logs")
    def get_build_logs():
        ...
    @app.route("/api/sync", methods=["POST"])
    def sync_configs():
        ...

    return app
```

- **Root (`/`):** Renders an HTML template from the `WebUITemplates.INDEX_TEMPLATE` string (defined later). This is the main page showing the UI.
- **Template APIs:** Similar to `create_web_ui`, but includes DELETE and a `/generate` endpoint to trigger template creation. All template changes auto-sync to Git if enabled [35] [36].
- **Package Search (`/api/packages/search`):** Accepts a query `q` and returns JSON of matching package examples (currently a mock example).
- **Build Logs (`/api/build-logs`):** Reads the last 50 entries from the `build_logs` table in the SQLite DB and returns them as JSON [37] [38].
- **Sync (`/api/sync`):** A POST endpoint to trigger `manager.sync_configs()`, returning success or error.

This enhanced UI ties together template management, package search, build history, and config syncing. The actual HTML/JS (not shown here in code except as templates) would call these endpoints.

Main Async Function and CLI

After defining classes and UI functions, the script has an `async def main()` (L856–914) and then an `if __name__ == "__main__":` block (L916–951). Together, these parse command-line arguments and dispatch actions.

- **Async main() (L856-914):**

```
async def main():
    parser = argparse.ArgumentParser(...)
    parser.add_argument("action", choices=["install", "search", "template",
"sync", "ui", "web"], ...)
    parser.add_argument("packages", nargs="*", help="Package names")
    parser.add_argument("--batch", action="store_true",
help="Batch install mode")
    parser.add_argument("--web-port", type=int, default=5000, help="Web UI
port")
    args = parser.parse_args()
    manager = ArchSourceManager()

    if args.action == "install":
        ...
    elif args.action == "template":
        ...
    elif args.action == "sync":
        ...
    elif args.action == "ui":
        manager.run_ui()
    elif args.action == "web":
        app = create_web_ui()
        app.run(host="0.0.0.0", port=args.web_port, debug=True)
    elif args.action == "search":
        ...
```

- Parses an action (`install`, `search`, `template`, `sync`, `ui`, or `web`).

- For `install`, it calls `manager.install_packages(args.packages, args.batch)` and prints success/failure for each result.
- For `template`, it generates a template for each given package name.
- For `sync`, it syncs configs.
- For `ui`, it runs the curses UI (`manager.run_ui()` is synchronous).
- For `web`, it creates and runs the Flask web app (on a host/port).

- For `search`, it simply prints whether each package is official or AUR (using `manager.detector.detect_source`) ㊴ ㊵.
  The CLI uses `asyncio.run(main())` to execute this async function in an event loop.

- `if __name__ == "__main__":` **block (L916-951 and again at L1738+):**
  The script checks how it was invoked. There are two similar blocks in the code (looks like the script has duplicated code). Focusing on the second one (L1738-1776), which seems final:

```
if __name__ == "__main__":
    if len(sys.argv) > 1 and sys.argv[1] == "create-installer":
        create_installation_script()
        sys.exit(0)
    # Normal execution
    if len(sys.argv) > 1 and sys.argv[1] in ["install", "search", "template",
"sync"]:
        asyncio.run(main())
    else:
        parser = argparse.ArgumentParser(...)
        parser.add_argument("action", nargs="?", default="ui",
choices=["install", "search", "template", "sync", "ui", "web"], ...)
        parser.add_argument("packages", nargs="*", help="Package names")
        parser.add_argument("--batch", action="store_true", help="Batch install
mode")
        parser.add_argument("--web-port", type=int, default=5000, help="Web UI
port")
        args = parser.parse_args()
        if args.action == "ui":
            manager = ArchSourceManager()
            manager.run_ui()
        elif args.action == "web":
            app = create_enhanced_web_ui()
            print(f"Starting web UI on http://localhost:{args.web_port}")
            app.run(host="0.0.0.0", port=args.web_port, debug=True)
        else:
            asyncio.run(main())
```

- If the first argument is `"create-installer"`, it runs `create_installation_script()` to produce an install script (see below) and exits.
- If action is one of `install, search, template, sync`, it runs `asyncio.run(main())` (using the async main defined above).
- Otherwise (for UI modes), it parses arguments with default action `"ui"`, and then either runs the curses UI, or for `"web"` it uses `create_enhanced_web_ui()` (note: enhanced vs basic) and starts it.

Thus, the user can invoke the script with different commands to trigger various behaviors: - `archsource install pkg1 pkg2` runs `main()` which builds those packages. - `archsource ui` starts the terminal

UI. - `archsource web` starts the Flask web interface. - `archsource create-installer` writes the installation shell script.

CacheManager Class

`CacheManager` (L955–1066) manages cached files to speed up builds/downloads, ensuring the cache doesn't exceed a given size.

- **Initialization (L959-967):**

  ```
  def __init__(self, cache_dir: Path, max_size: str = "1GB"):
      self.cache_dir = cache_dir
      self.cache_dir.mkdir(parents=True, exist_ok=True)
      self.max_size = self._parse_size(max_size)
      self.cache_db = cache_dir / "cache.db"
      self.init_cache_db()
  ```

  It sets up a directory for cache, parses the maximum size string (like "1GB"), and initializes a SQLite DB for cache entries [41] .

- **Parsing Size (L966-975):**

  ```
  def _parse_size(self, size_str: str) -> int:
      size_str = size_str.upper()
      if size_str.endswith("GB"):
          return int(size_str[:-2]) * 1024**3
      elif size_str.endswith("MB"):
          return int(size_str[:-2]) * 1024**2
      elif size_str.endswith("KB"):
          return int(size_str[:-2]) * 1024
      else:
          return int(size_str)
  ```

  Converts "1GB" or "500MB" into a number of bytes.

- **Cache DB (L978-986):**

  ```
  def init_cache_db(self):
      conn = sqlite3.connect(self.cache_db)
      cursor = conn.cursor()
      cursor.execute("""
          CREATE TABLE IF NOT EXISTS cache_entries (
              path TEXT PRIMARY KEY,
  ```

```
            size INTEGER,
            last_accessed TEXT,
            hash TEXT
        )
    """)
    conn.commit()
    conn.close()
```

It creates a table `cache_entries` to track each cached file (its path, size, last access time, and a hash) [42] .

- **Adding Files to Cache (L997-1004):**

```
def add_to_cache(self, file_path: Path, category: str = "packages"):
    if not file_path.exists():
        return
    cache_category_dir = self.cache_dir / category
    cache_category_dir.mkdir(exist_ok=True)
    file_hash = self._calculate_hash(file_path)
    file_size = file_path.stat().st_size
    cache_path = cache_category_dir / file_path.name
    shutil.copy2(file_path, cache_path)
    conn = sqlite3.connect(self.cache_db)
    cursor = conn.cursor()
    cursor.execute("""
        INSERT OR REPLACE INTO cache_entries (path, size, last_accessed,
hash)
        VALUES (?, ?, ?, ?)
    """, (str(cache_path), file_size, datetime.now().isoformat(),
file_hash))
    conn.commit()
    conn.close()
    self._cleanup_cache()
```

`add_to_cache` copies a given file into the cache (under `cache_dir/<category>/`). It calculates its SHA-256 hash and records an entry in the database with the path, size, current time, and hash. Then it calls `_cleanup_cache()`.

- **Hash Calculation (L1032-1039):**

```
def _calculate_hash(self, file_path: Path) -> str:
    sha256_hash = hashlib.sha256()
    with open(file_path, "rb") as f:
        for chunk in iter(lambda: f.read(4096), b""):
            sha256_hash.update(chunk)
    return sha256_hash.hexdigest()
```

Computes the SHA-256 hash of the file contents (for identifying unique files).

- **Cleaning Up (L1040-1050):**

```
def _cleanup_cache(self):
    conn = sqlite3.connect(self.cache_db)
    cursor = conn.cursor()
    cursor.execute("SELECT path, size FROM cache_entries ORDER BY
last_accessed ASC")
    entries = cursor.fetchall()
    total_size = sum(entry[1] for entry in entries)
    while total_size > self.max_size and entries:
        oldest_entry = entries.pop(0)
        cache_path = Path(oldest_entry[0])
        if cache_path.exists():
            cache_path.unlink()
            total_size -= oldest_entry[1]
        cursor.execute("DELETE FROM cache_entries WHERE path = ?",
(oldest_entry[0],))
    conn.commit()
    conn.close()
```

If the cache exceeds `max_size`, this removes the oldest accessed files first (based on `last_accessed` ordering in the DB) until under the limit. Each deletion also removes its DB entry [43] .

In practice, the `CacheManager` could be used to store downloaded source archives or built packages so that repeated builds/downloads are faster. In the code shown, a `CacheManager` is instantiated in `create_enhanced_web_ui`, suggesting future use (e.g., caching package queries or build artifacts), but it isn't fully integrated with the build process in the snippet shown.

PacmanIntegration Class

This class (L1068–1130) provides utilities related to the system package manager (**pacman**). It can parse `/etc/pacman.conf`, list repos, and check/install packages.

- **Initialization (L1071-1075):**

```
def __init__(self):
    self.pacman_conf = self._parse_pacman_conf()
    self.repos = self._get_configured_repos()
```

It reads the pacman config and extracts repository names.

- **Parsing pacman.conf (L1075-1080):**

```
def _parse_pacman_conf(self) -> configparser.ConfigParser:
    config = configparser.ConfigParser(allow_no_value=True)
    config.read("/etc/pacman.conf")
    return config
```

This loads `pacman.conf` using `configparser`. It's not used elsewhere in the shown code, but it could be used to query repo settings.

- **Listing Repos (L1081-1087):**

```
def _get_configured_repos(self) -> List[str]:
    repos = []
    for section in self.pacman_conf.sections():
        if section != "options":
            repos.append(section)
    return repos
```

Returns all sections from `pacman.conf` except `[options]`, effectively listing enabled repository names.

- **Check if Installed (L1089-1097):**

```
def is_package_installed(self, package_name: str) -> bool:
    try:
        result = subprocess.run(
            ["pacman", "-Qi", package_name], capture_output=True, timeout=5
        )
        return (result.returncode == 0)
    except subprocess.TimeoutExpired:
        return False
```

Runs `pacman -Qi <pkg>` to query a package. A return code of 0 means it's installed.

- **Get Installed Version (L1099-1107):**

```
def get_installed_version(self, package_name: str) -> Optional[str]:
    try:
        result = subprocess.run(
            ["pacman", "-Qi", package_name],
            capture_output=True, text=True, timeout=5,
        )
        if result.returncode == 0:
            for line in result.stdout.split("\n"):
                if line.startswith("Version"):
                    return line.split(":", 1)[1].strip()
    except subprocess.TimeoutExpired:
        pass
    return None
```

Parses the output of `pacman -Qi` to find the "Version" line and returns that version string.

- **Installing a Built Package (L1118-1127):**

```
def install_built_package(self, package_path: Path) -> bool:
    try:
        result = subprocess.run(
            ["sudo", "pacman", "-U", str(package_path), "--noconfirm"],
            capture_output=True, timeout=300,
        )
        return result.returncode == 0
    except subprocess.TimeoutExpired:
        return False
```

Installs a local package file (e.g. a `.pkg.tar.xz` built by makepkg) using `pacman -U`. It requires `sudo`. Returns True if successful.

These utilities help integrate ArchSource with the system's package database, though in the code shown they are not deeply used (they might be for future features like auto-installing built packages).

Web UI Templates

The `WebUITemplates` class (L1131–1320) holds HTML/CSS/JS for the enhanced web UI. It's one big string defining the entire webpage. It contains a beautiful HTML UI for template management, with inlined CSS and a script that calls the above Flask endpoints. We won't detail every line of HTML/CSS, but note:

- `INDEX_TEMPLATE` (L1134-1320) includes:
- A styled container showing a header “ArchSource Template Manager” and a form to create a new template.
- A grid of template cards showing existing templates (loaded via AJAX from `/api/templates`).

- An editor dialog for editing a template's contents.
- JavaScript functions: `loadTemplates()`, `createTemplate()`, `editTemplate()`, `saveTemplate()`, `deleteTemplate()`, etc., which call the `/api/templates` endpoints to perform actions, showing success or error messages [44] [45] [46].
- For example, when the page loads (`DOMContentLoaded`), `loadTemplates()` fetches the list of templates from `/api/templates` and displays them.

This HTML/JS code is the frontend to the enhanced web API. It's included as a Python string so that `render_template_string(WebUITemplates.INDEX_TEMPLATE)` can serve it without separate files.

Installation Script Generator

The function `create_installation_script()` (L1681–1735) is a utility to generate a shell script (`install.sh`) that automates setting up ArchSource on a system [47] [48]:

```
def create_installation_script():
    install_script = """#!/bin/bash
    # ArchSource Package Manager Installation Script

    set -e

    echo "Installing ArchSource Package Manager..."

    # Check if running as root
    if [[ $EUID -eq 0 ]]; then
        echo "This script should not be run as root"
        exit 1
    fi

    # Install Python dependencies
    echo "Installing Python dependencies..."
    pip install --user aiohttp pyyaml gitpython flask

    # Create directories
    mkdir -p ~/.config/archsource
    mkdir -p ~/.cache/archsource
    mkdir -p ~/.local/share/archsource/logs
    mkdir -p ~/.local/bin

    # Copy main script
    cp archsource.py ~/.local/bin/archsource
    chmod +x ~/.local/bin/archsource

    # Create symlink for system-wide access (requires sudo)
    if command -v sudo &> /dev/null; then
        echo "Creating system-wide symlink..."
        sudo ln -sf ~/.local/bin/archsource /usr/local/bin/archsource
    fi

    # Add to PATH if not already there
    if ! echo $PATH | grep -q "$HOME/.local/bin"; then
        echo 'export PATH="$HOME/.local/bin:$PATH"' >> ~/.bashrc
        echo "Added ~/.local/bin to PATH in ~/.bashrc"
        echo "Please run: source ~/.bashrc"
    fi

    echo "Installation completed!"
    echo "Usage examples:"
    echo "  archsource install package1 package2"
    echo "  archsource template package_name"
    echo "  archsource ui"
    echo "  archsource web"
    """
    with open("install.sh", "w") as f:
        f.write(install_script)
    os.chmod("install.sh", 0o755)
    print("Created install.sh script")
```

This creates a Bash script with steps: - Check not running as root. - Use `pip install --user` to install needed Python packages (`aiohttp`, `pyyaml`, `gitpython`, `flask`) in the user's home. - Make the config/cache/log directories. - Copy the main script (`archsource.py`) to `~/.local/bin/archsource` and make it executable. - Optionally create a sudo symlink in `/usr/local/bin`. - Optionally add `~/.local/bin` to the shell PATH.

When the user runs `python archsource.py create-installer`, this script will be generated.

Application Flow Summary

Putting it all together, here is how the application flows:

1. **Start-up:** The user runs the script with an action. The `if __name__ == "__main__":` block determines what to do:
2. If action is `install`, `search`, `template`, or `sync`, it calls `asyncio.run(main())` [39] [49].
3. If action is `ui`, it creates an `ArchSourceManager` and calls `run_ui()` for the terminal UI [50].
4. If action is `web`, it starts a Flask web server (enhanced UI on a given port) [51].
5. If action is `create-installer`, it creates the install script.
6. **Configuration:** On creation of `ArchSourceManager`, the `ConfigManager` loads or creates `config.yaml` and prepares the `templates` directory [11]. It may clone or init a Git repo there for template sync [12].

7. **Database:** The `DatabaseManager` ensures an SQLite DB exists and has the right tables [52].

8. **Actions:** Based on the action:

9. **Install:** The CLI calls `install_packages()` which:
    - Detects sources and dependencies for each package (using `PackageDetector` and `DependencyResolver`).
    - Builds each package (AUR or official) via `BuildManager`, possibly in parallel.
    - Logs build results to the database.
    - The CLI prints success/fail for each package [39].
10. **Template:** CLI or web API calls `generate_template()` which writes a default `makepkg.conf` for the named package in `~/.config/archsource/templates`. It prints the path [53].
11. **Sync:** CLI or web API calls `sync_configs()`, which runs the Git sync to push/pull template files [54].
12. **Search:** CLI uses `detect_source()` to say "official" or "aur" for each name [40].
13. **UI:** If `ui`, the curses UI runs in terminal, allowing interactive actions (though in code, this is mostly menu display).

14. **Web:** If `web`, the Flask app runs. In enhanced mode, visiting the root page shows the HTML UI (from `WebUITemplates.INDEX_TEMPLATE`), where users can view templates, edit/generate them, search for packages, see logs, etc. The back-end APIs handle these requests.

15. **Cache & Pacman:** During operations, the code could use `CacheManager` to store and limit cached files (e.g. downloaded sources or built packages) [55], although the snippet only shows `CacheManager` usage in the enhanced web UI setup (not shown fully in use). The `PacmanIntegration` class is available to check if packages are installed or to install built packages [56] [57], but calling those is not fully shown in the main flow (it might be part of future or omitted features).

All configuration (YAML) and templates live under `~/.config/archsource`. Builds and caches go under `~/.cache/archsource`. Logs go under `~/.local/share/archsource/logs`. GitHub syncing, if enabled, keeps the `templates` folder mirrored with the specified Git repo.

In short, **ArchSourceManager** orchestrates everything. It loads settings, resolves dependencies, calls the build manager to compile packages, saves results to the database, and provides interfaces (CLI, curses, web) for the user to interact. Each helper class (DatabaseManager, ConfigManager, etc.) encapsulates a distinct piece of functionality (storage, config, package querying, building, UI, caching).

Sources: Code lines are cited above to show where key functions and structures are defined [1] [58] [59] [13] [27], as required by the guidelines. The explanation is based on that code.

---

[1] [2] [3] [4] [5] [6] [7] [8] [9] [10] [11] [12] [13] [14] [15] [16] [17] [18] [19] [20] [21] [22] [23] [24] [25] [26] [27] [28] [29] [30] [31] [32] [33] [34] [35] [36] [37] [38] [39] [40] [41] [42] [43] [44] [45] [46] [47] [48] [49] [50] [51] [52] [53] [54] [55] [56] [57] [58] [59] arch-pkg-manager-python.py

file://file-5SP717UQpTiYuEjMjQMTB1