Build-Benedictions

Aliases: `buildben`, `bube`

Managing Multiple (Python) Projects & Dependencies

using

```
$ bube init-proj
```

Dr. rer. nat. Martin Kuric

Germania Sacra / HisQu · Academy of Sciences Göttingen

Disclaimer

- `buildben` is very easy to use. (Goal is to make work simpler)
- This presentation is for python beginners.

But ...

- ... `buildben` solves **a lot** of *behind-the-scenes-problems* at once.
  → The logic behind `buildben` is **not beginner-friendly**.
- Some problems are hard to understand if you haven't encountered them yet...
  I myself don't understand them fully either, *I simply trust the best practices..!*
  → expect some *(un)organized chaos*...

PLEASE INTERRUPT ME AT ANY POINT!

What's `buildben`?

ChatGPT:

> *“buildben is like **Cookiecutter** plus automatic virtual-env creation, dependency locking, and helper tasks.”*

... and what's a Cookiecutter?

> *"A **Cookiecutter** is a project template that can be used to create new projects with a predefined structure and configuration. It is a tool that helps developers quickly set up new projects by providing a standardized starting point."*

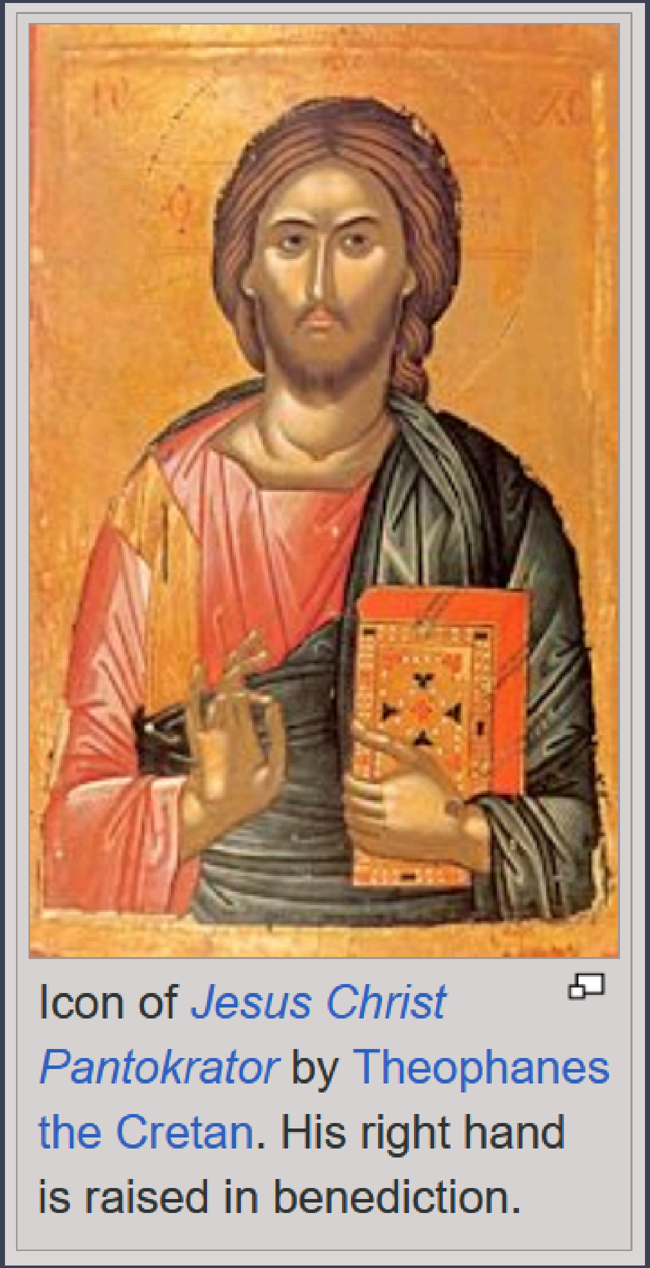
Main Modules:

Aliases: **`buildben`**, **`bube`**

- `$ bube init-proj`: Create a new **project**. — ✅ *99% Done*
- `$ bube add-experiment`: Add a new **experiment** to a project. — 🤞 *80% Done*
- `$ bube env-snapshot`: Dockerize current project for reproducibility. — 🤞 *80% Done*
- `$ bube init-database`: Create a new central **database**. — 🏗 *60% Done*

How This all Started:

- I had one big mono-repository containing multiple projects. It was a MESS.
- After splitting into smaller repos: Managing multiple separate projects is painful, too..!
  - "Let's just start developing, I can add a `setup.py` later!"
  - "When did I last update the `requirements.txt` / `setup.py`?"
  - "When anyone tries to use this code, the setup will probably break..."
- I had scripts to automate tasks, all of them poorly documented & scattered across repos!
- I needed one centralized standard to solve **all** my problems:
  - Think ahead, avoid problems, read my mind, etc.
  - Minimal interaction: No more than 1 CLI-command to do 100 things at once.
  - (*like a quick prayer doing miracles ...* 🙏😇)



Icon of *Jesus Christ Pantokrator* by Theophanes the Cretan. His right hand is raised in benediction.

From Wikipedia:

*"A **benediction** (Latin: bene, 'well' + dicere, 'to speak') is a short **invocation** for divine help, blessing and guidance [...]."*

*"**Invocation** is the act of calling upon a deity, spirit, or supernatural force, typically through prayer, ritual, or **spoken formula**, to seek guidance, assistance, or presence."*

My Projects before `buildben`:

1. Make a virtual environment (`.venv`) for each project:

```
python -m venv ".venv"  # Prevents polluting your OS with project-related chaos
source .venv/bin/activate  # Activate virtual environment
```

2. Collect my dependencies in a `"proj-requirements.txt"` file.
3. `pip`: Collects *dependencies of my dependencies* and installs everything:

```
pip install -r "proj-requirements.txt"  # Resolve Environment & install dependencies
```

4. Compile all installed dependencies + versions for further reinstalls:

```
pip freeze > "requirements.txt"  # Compile list of dependencies installed in current .venv
```

`proj-requirements.txt`

- Manually created by me: Whenever I `pip install` a new package, I add it to this file.
- Used by `pip` to *"resolve the environment"* (= collect *dependencies of dependencies*)

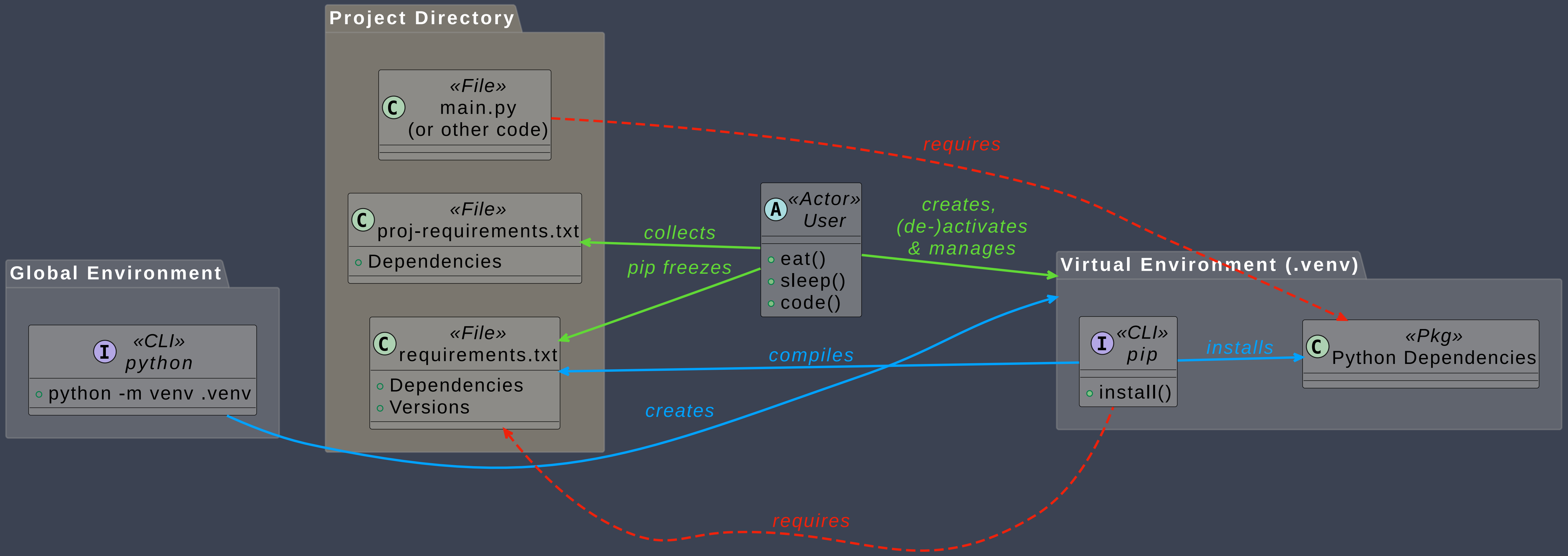
```
ipykernel
jupytext    # Convert .ipynb to .py
numpy
openpyxl    # For reading Excel files
pandas
matplotlib
seaborn     # Better plotting
pytest
```

requirements.txt

```
pip freeze > "requirements.txt"  # Compile list of dependencies installed in current .venv
```

```
asttokens==3.0.0
build==1.2.2.post1
click==8.2.1
comm==0.2.2
debugpy==1.8.14
decorator==5.2.1
ipykernel==6.29.5
ipython==9.4.0
ipython_pygments_lexers==1.1.1
jedi==0.19.2
jupyter_client==8.6.3
jupyter_core==5.8.1
matplotlib-inline==0.1.7
# ...
```

My Projects before `buildben`: Architecture



"I will add a `pyproject.toml` later..!"

My Projects before `buildben`: Setup

```
git clone "<repo-url>"          # Download
cd "<repo-name>"
python -m venv ".venv"          # Prevents polluting your OS with project-related chaos
source .venv/bin/activate       # Activate virtual environment
```

If there's only a `"requirements.txt"`:

```
pip install -r "requirements.txt"   # Install only dependencies
```

If there's a `pyproject.toml`:

```
pip install -e .                   # Editable install
```

My Projects before `buildben`: 2 Main Problems

1. **Dependencies are pinned by hand:**
   - `requirements.txt` must be manually updated.
2. **Imports rely on current working directory:**
   - `requirements.txt` only holds dependencies, not the **project structure**.
   - Cannot import anything outside the current working directory (no `import ../module`)
   - VS Code (sometimes) struggles with **refactoring** & **typing** across packages.

Further Annoyances:

1. `requirements.txt` mixes runtime and development dependencies.
2. (De-)Activating `.venv` can be forgotten or annoying.
3. Too many CLI-commands to remember & type *(especially when working with 4 Repos at the same time)*.
4. How to properly write unit-tests mid-development..?

Solutions:

Building Block	Why beginners should care	Standard
`pyproject.toml`	Single file that stores metadata and tool config	PEP 621
`pip install -e .`	Code changes are picked up without re-install	PEP 660
`src/` layout	Forces tests to run on the installed package	PyPA guide
`pip-tools`	Compiles `*requirements.txt` & syncs it with venv	(realpython.com)
`direnv`	Activates the correct virtual env when you `cd`	(direnv docs)
`just`	Saves “one-liners” like `just insco`	(just README)

`bube proj`: Workflow

1. **`$ bube proj` sets up a *ready-to-use* project directory (*Cookie-Cutter*):**
    - `pyproject.toml`: Pre-configured for `src`-layout, basic dependency list, etc.
    - `.envrc`: Tells `direnv` to create & activate virtual environment automatically.
    - `justfile`: Comes with working recipes (functions) to install, etc.
    - Many more...
2. **Use `just` recipes for everyday tasks:**
    - Installing your project: `just install-compile`
    - Resetting environment: `just reset-venv`
    - Upgrading dependencies: `just upgrade-deps`
    - You can add more yourself!

`bube proj`: Demonstration

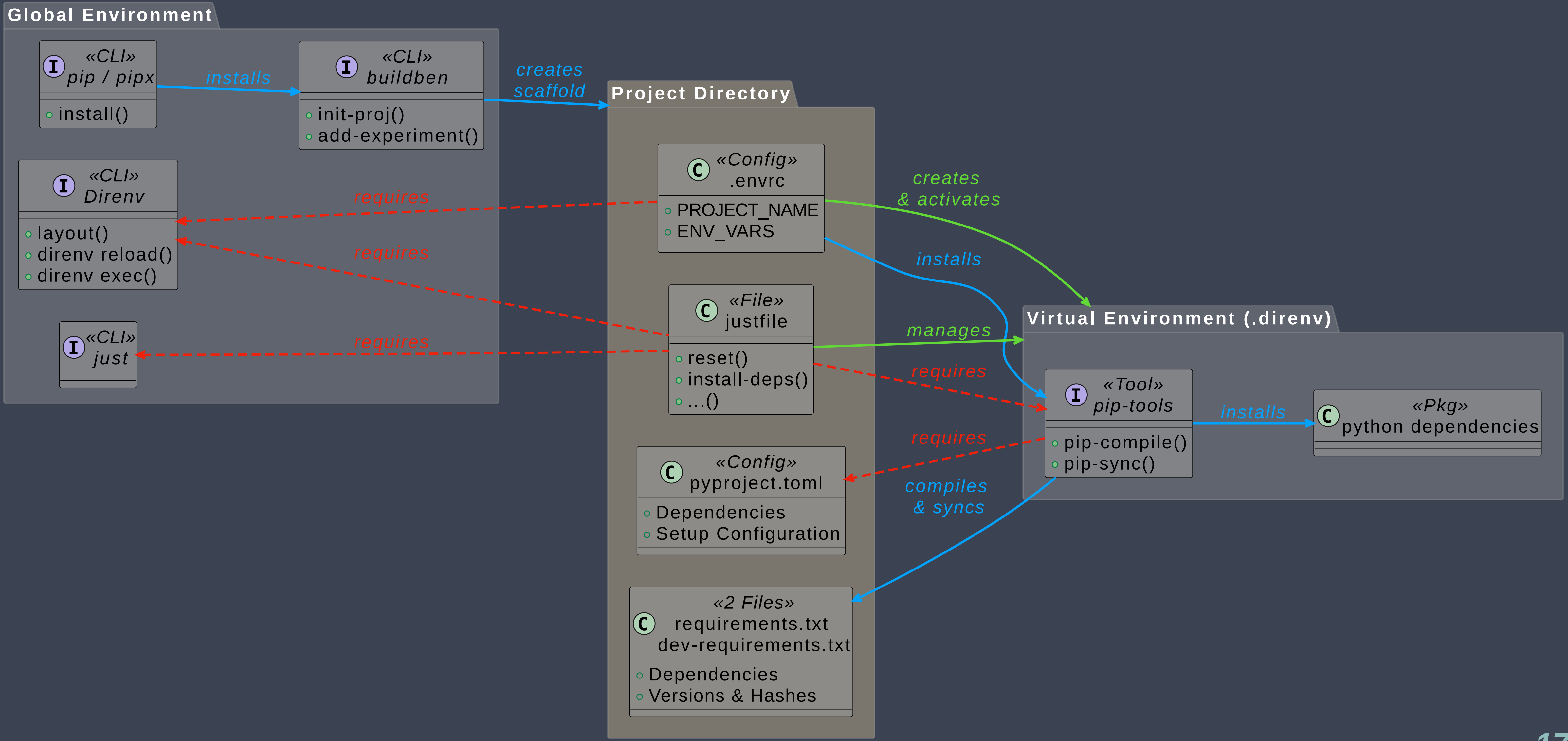
```
bube -h                                  # Show help message
bube proj -h                             # Shorthand for `buildben init-proj -h`
bube proj "sheesh" -t . -g -u "<your_github_username>"  # Cookiecutter project
cd "sheesh"                              # Change to project directory
direnv allow                             # Trust & execute .envrc
# A .direnv directory is created containing the virtual environment
just                                     # Show available recipes
just install-compile                     # Install project, compile requirements.txt

cd ..                       # Demonstrate auto-deactivation of direnv
cd bla_a                    # Demonstrate auto-activation of direnv
cd ../sheesh                # Demonstrate auto-deactivation and activation of direnv

just reset-venv             # Fully Nuke the virtual environment, start fresh!
```

`bube proj`: Minimal Example

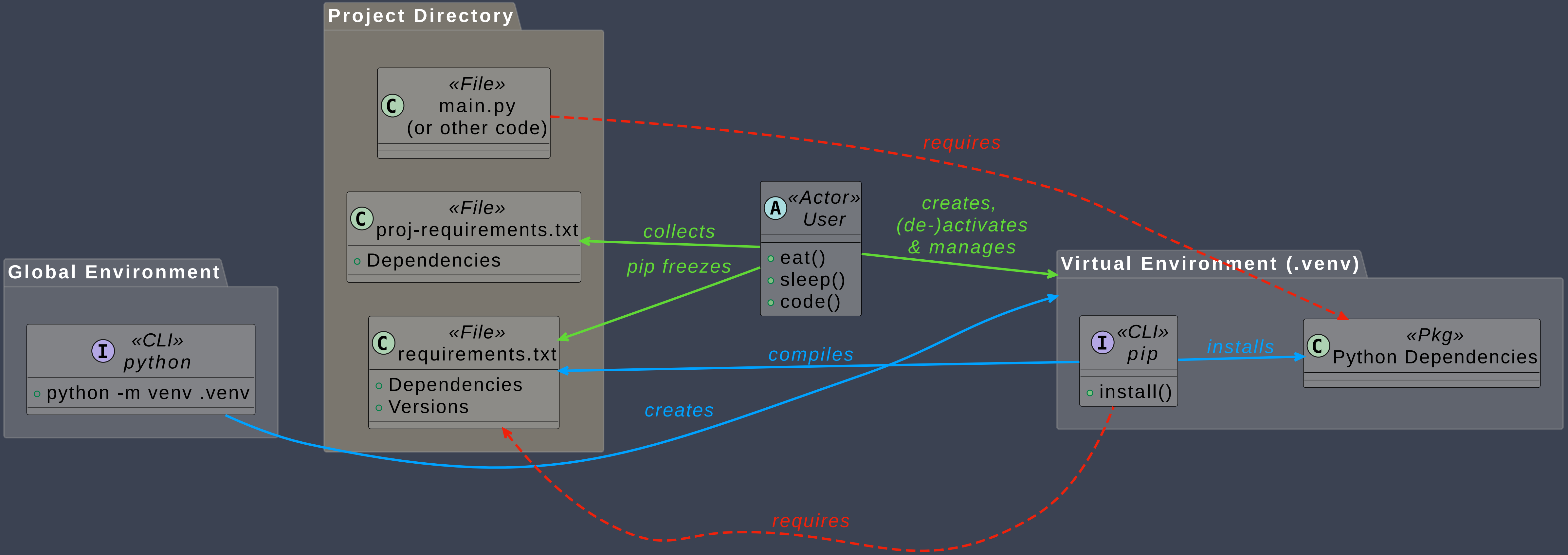
4 Lines to set up a new project.

```
bube proj "sheesh" -t . -g -u "<your_github_username>"
cd "sheesh"
direnv allow
just install-compile
```

With `buildben`:



Without `buildben`:



`just`, `justfiles` & Recipes

`just`, `justfiles` & Recipes

- A "Recipe" is a `bash` function that can be called from the command line.

```
# Docstring for the recipe (optional)
recipe-name *ARGS:
    echo "Hello, World!"
    echo "This is a recipe."
    rm {{ARGS}}     # Pass arguments
alias rcp-nm:=recipe-name  # Create an alias for the recipe
```

- All Recipes are stored in a file called `justfile` in the project root.
- `just` auto-detects the `justfile` and provides a CLI to run the recipes.

`pyproject.toml`

`pyproject.toml`

```
[project]
  name           = "<my_project>"
  version        = "0.1.0"
  description = "Short, one-line summary."
  authors        = [{ name = "<github_username>", email = "you@example.com" }]
  readme         = "README.md"
  license        = { text = "MIT" }
  requires-python = ">=3.12"
```

- Contains project metadata:
  - Dependencies (*it replaced my* `proj-requirements.txt` *file*).
  - Build system (e.g. `setuptools`, `poetry`, `uv`).
- Used by `pip` & `pip-sync` to install the project and its dependencies.
- Used by `pip-compile` to generate lock-file: `requirements.txt`.

`pyproject.toml`: Main Dependencies

- Whenever you `pip install` a package, add it to the list (like I did with my `proj-requirements.txt`)
  → Otherwise it will be forgotten when you re-install the project

```
[project]
  dependencies = [
    "numpy",
    "openpyxl",
    "matplotlib==3.5.1",     # You can pin a specific version
    "pandas>=2.0.0,<3.0.0", # You can pin a version range
    "IPython",              # Unpinned versions will be resolved by pip or pip-compile
    "...",                  # Add more dependencies here
  ]
```

`pyproject.toml`: Private / Unpublished Projects as Dependencies

- `pip install <package>` only works for packages published on PyPI.
    - (*We don't bother with PyPI, yet. Also, GitHub-Submodules are scary*)
- Any `pip`-installable projects can be added via **Git-URL** or **local path**:

```
[project.dependencies]
"<my_project> @ file://../<my_project>",                # Local path (easiest).
"<my_project2> @ git+https://github.com/HisQu/<my_project2>.git",   # Git-URL
```

- **Git version control:** Add `@<branch>`, `@<tag>` or `@<commit_hash>` after the Git-URL:

```
"<my_project3> @ git+https://github.com/HisQu/<my_project3>.git@<commit>"
```

`pyproject.toml`: Optional Dependencies

- Unnecessary dependencies risk breaking the project for users who don't need them.
- Development-tools are only needed for development, make them optional:

```toml
[project.optional-dependencies]
  dev = [      # Name of the optional dependency group
    "pytest",
    "...",     # Add more dependencies here
  ]
```

- Include optional dependencies on installation:

```
pip install -e .[dev]  # Install dependencies + development-dependencies
```

`pyproject.toml`: Packaging

- **Packaging** = Collecting all files needed to run the project into a single *distributable*.
- Python packages are usually distributed as *wheels* (`.whl` files).
- By default, Python uses `setuptools` to auto-package projects (PEP 517).
  - Other packaging-tools use `pyproject.toml`, too (`poetry`, `flit`, `uv`).
- `setuptools` scans for any , and packages them automatically.

`pyproject.toml`: Packaging Nomenclature

Component	Defintion	Contains
Module	single `.py` file	vars, funcs, classes
Package (pkg)	directory *with* `__init__.py`	modules & sub-pkgs
Namespace pkg	directory *without* `__init__.py` (PEP 517)	modules & sub-pkgs
Sub-package	nested package	modules (& sub-pkgs)
Project	collection of code units	pkgs, modules, assets, etc.

`pyproject.toml`: Packaging of `.py` files

- Modules & Packages inside `package-dir` will be copied into `".venv/lib/<my_project>"`.

```
[tool.setuptools]
package-dir = { "" = "src"}     # "<my_project>/src/" --> ".venv/lib/<my_project>"
```

- Further components are scanned

```
[tool.setuptools.packages.find]
where = ["src"]                   # Scan "<my_project>/src/" for packages (subdirectories)
```

- This determines the `import`-paths:

```
from <my_project>.<package>.<module> import <your_class>, <your_variable>
```

`pyproject.toml`: Packaging strategy of `buildben`

- `$ bube proj` returns a `pyproject.toml` with a conservative strategy:
    - Use the `src/`-layout
    - Use *a single parent directory* as the root of the project.

Why the single `src/` directory is *good practice*

- **Eliminates "works-on-my-machine" imports**: code isn't on `sys.path` until *after* installation, so tests mirror the real wheel behaviour
- **Prevents accidental shadowing**: the current working directory can't mask an already-installed package of the same name
- **Forces proper packaging earlier**: you *must* set up `package-dir` / `find` once, then forget about it—cleaner CI and fewer surprises
- **Keeps import statements short & stable**: e.g. `from my_project.subpkg.mod import Foo` just works after `pip install -e .`
- **Yes, you *could* nest multiple roots, flat-layout, mixed C-extensions…** but every extra path mapping adds maintenance cost; for most apps the single-dir rule of thumb is "99 % right, 0 % regrets"

`pyproject.toml`: Packaging non-`.py` files

- Anything that's not a `.py`-file must be explicitly added:
- The path-logic is

```
[tool.setuptools.package-data]
"<my_project>" = [
    "data/**/*.xlsx",           # Located in "src/<my_project>/data/"
    "images/**/*.{png,jpg}",  # Located in "src/<my_project>/images/"
    ]
```

`pyproject.toml`: Manual Configurations after `$ bube init-proj`

Do immediately:

- Add description, license, authors, etc. under `[project]`

Do mid-development:

- Add emerging dependencies to `[project]`
- Add emerging non-`.py` files to `[tool.setuptools.package-data]`

Don't Do *unless you know what you're doing*:

- Modify the `[build-system]` section
- Change *single directory* `src/` layout `[tool.setuptools]`, `[tool.setuptools.packages.find]`

`pip-tools`

`pip-tools` = `pip-compile` + `pip-sync`

`pip-compile`:

- Compiles a `requirements.txt` file from the `pyproject.toml` file (*unlike* `pip freeze`).
- Automatically resolves dependencies and their versions.
- Generates a `requirements.txt` file with pinned versions.

`pip-sync`:

- Synchronizes the virtual environment with multiple lock files (e.g. `*requirements.txt`):
  - Installs packages from the lock files.
  - Un-installs packages not listed in lock-files (*unlike* `pip install`).

Capability	`pip freeze`	`pip-compile`
Locks transitive deps deterministically	⚠️ best-effort	✅ topologically sorted
Separates **direct** vs **indirect** deps	❌	✅ comments show who pulled what
Generates secure `--hash=` pins	❌	✅ `--generate-hashes` flag
Selective upgrades (e.g. --upgrade-package flask)	❌	✅ built-in
Understands modern metadata (PEP 621 `pyproject.toml`)	❌	✅

`src/` -Layout

Project Structure: `src/`-Layout

```
# src layout (good)                 # flat layout (risky)
myproject/                          myproject/
├── src/                            │
│   └── myproject/                  │
│       ├── main.py                 ├── main.py
│       └── package/module.py       ├── package/module.py
├── tests/                          ├── tests/
│   └── test_module.py              │   └── test_module.py
├── README.md                       ├── README.md
```

Benefits:

- Avoids imports from working directory via `PYTHONPATH`
  → Forces tests to run on installed code: `pip install -e .` → Catches `import` bugs
- Builds **clean wheels**: Stray files never ship to PyPI

Project Structure: Inside `src/`

```
myproject/
├── src/
│   └── myproject/              # Single directory, same name as project root (Recommended)
│       ├── __init__.py         # Marks directory as package; runs on first import!
│       ├── main.py             # Optional CLI entry-point (wired in via pyproject.toml)
│       ├── shishkebab.py       # >>> import myproject.shishkebab
│       ├── clients/            # >>> import myproject.clients
│       │   ├── __init__.py     # Sub-package "clients"
│       │   ├── llm.py          # >>> import myproject.clients.llm
│       │   └── embedding.py    # >>> import myproject.clients.embedding
│       └── utils/              # >>> import myproject.utils
│           ├── __init__.py     # Sub-package "utils"
│           ├── cooltool.py     # >>> import myproject.utils.cooltool
│           └── module6.py      # >>> import myproject.utils.module6
```

Project Directory: Auxiliary Files in Project Root

```
myproject/
├── .venv/                 # Virtual environment (or .direnv!)
├── .env                   # Environment variables (& secrets)
├── .gitignore
├── .git/                  # Repository metadata
├── src/
│   └── myproject/        # Separate source code from tests!
├── tests/
│   └── test_module1.py    # Tests for module1
├── justfile               # Development tasks
├── pyproject.toml         # Project metadata, Setup!
├── requirements.txt       # Dependencies
├── requirements-dev.txt   # Development dependencies
├── README.md
├── LICENSE
```

Installation of `buildben`

Prerequisites:

- Python installed on your OS (and you know its executable in your `$PATH`)
- A Package manager (`apt`, `brew`, `winget`, etc.)

🏃Quick & Dirty:

```
git clone https://github.com/markur4/buildben.git
pip install -e buildben    # venv recommended. (Also, you might want just & direnv.)
```

🏗️ Full Install (recommended):

1. Install `pipx`:

To use `buildben` globally and to keep the OS-python clean, we recommend `pipx`.

```
sudo apt install pipx          # For Ubuntu
# brew install pipx            # For MacOS
# py -m pip install --user pipx   # For Windows (Not tested!)
pipx ensurepath                   # Add pipx to PATH, if not already done
pipx upgrade-all                  # !! Never run pipx with sudo !!
```

2. Clone & install `buildben`:

```
git clone https://github.com/markur4/buildben.git
cd buildben         # Needed, `pipx install buildben` does NOT work!
pipx install -e .   # Editable for direct modifications.
```

🏗 Full Install (recommended):

3. Install `just`:

```
sudo apt install just      # For Ubuntu
# brew install just        # For MacOS
# pipx install rust-just   # Windows requires the cross-platform version (not tested!)
```

4. Install `direnv` & hook it into your shell:

- *Either* follow the instructions for install & hook,
- *Or* run `src/buildben/setup_zsh.sh` to install both `zsh` & other useful plugins, including `direnv`.