

Anže's Blog

Python, Django, and the Web

29 Aug 2024

UV with Django

In a hurry? Jump to the relevant section:

- [Create a new Django project with uv](#)
- [Add uv to an existing project](#)
- [Dev dependencies](#)
- [Running uv in production](#)
- [Running uv in CI](#)
- [Upgrading dependencies](#)

What is uv?

[Astral](#) made a huge summer splash in the Python community last week when they released `uv` [0.3.0](#).

`uv` is a Python package manager written in rust that has just gained the ability to be a project management tool (like [Pipenv/PDM](#)), tool management ([pipx](#)), python installer ([pyenv](#)), and more!

I was very eager to try it out on my Django projects, but the initial 0.3.0 release was designed for managing installable Python

packages, so there were [a few rough edges](#) when using it to manage Django app dependencies. Only a week later, these issues have been addressed, defaults were switched around, and `uv 0.4.0` [now supports Python projects](#) like your Django application out of the box!

Using uv to create a new Django project

First, we'll need to get `uv`. See [the official docs](#) for all the installation options, but the easiest way to get it on Linux and MacOS is to run:

```
curl -LsSf https://astral.sh/uv/install.sh | sh
```

Now that we have `uv`, we can use it to start a new project with:

```
> uv init hello-django  
Initialized project `hello-django` at `/Users/anze/Coding/hello-d
```



Make sure you are using `uv 0.4.0` or newer; older versions will presume you are creating an installable Python package, and you'll see "error: Failed to prepare distributions" errors when trying to run your project. You can upgrade your `uv` version by running the install command above.

You can now `cd` into the `hello-django` folder and see that `uv` created three files for us:

```
README.md  
hello.py # we won't need this so feel free to rm it.  
pyproject.toml
```

The `pyproject.toml` file is the most interesting one since it defines two important properties: `requires-python` and `dependencies`. The former defines which Python version we will be using, and the latter defines the project dependencies.

```
[project]
name = "hello-django"
version = "0.1.0"
description = "Add your description here"
readme = "README.md"
requires-python = ">=3.13"
dependencies = []
```

Our Django projects won't really care about the `name`, `version`, `description`, and `readme` properties, so just leave them as is.

Speaking of project dependencies. It's about time we install Django with `uv add django` !

```
> uv add django
Using Python 3.13.0
Creating virtualenv at: .venv
Resolved 5 packages in 186ms
Prepared 3 packages in 3ms
Installed 3 packages in 235ms
+ asgiref==3.8.1
+ django==5.1
+ sqlparse==0.5.1
```

We can see that `uv` created a virtual environment (`.venv` folder), and installed Django with its two dependencies (`asgiref` and `sqlparse`). If we inspect the `pyproject.toml` file, we can see that Django was added to the dependencies list:

```
...
dependencies = [
```

```
"django>=5.1",  
]
```

The Django version has no upper bonds, so we can easily upgrade it when newer versions of Django come out (using `uv lock --upgrade`).

This doesn't mean our current dependencies aren't locked tight. Our whole dependency tree has its versions specified in the `uv.lock` file. The lock file is a [cross-platform lock file](#), so it should be safe to install on any operating system!

Now that we have Django installed, we can run Django's `startproject` command:

```
uv run django-admin startproject hello .
```

This initialized our Django project, including the `manage.py` file, `hello/settings.py` , etc.

We can start the Django development server with the following:

```
uv run manage.py runserver
```

Using uv with an existing Django project

If you already have a Django project, you can still use the `uv init` command to switch to uv.

```
cd to_existing_project  
> uv init .
```

```
Initialized project `hello-django` at `/Users/anze/Coding/blog/h
```

If your project already has a `pyproject.toml` file defined the command might fail with:

```
error: Project is already initialized in `/Users/anze/Coding/blog
```



In that case, you'll need to add a project table to your `pyproject.toml` file:

```
[project]
name = "hello-django"
version = "0.1.0"
description = "Add your description here"
readme = "README.md"
requires-python = ">=3.13"
dependencies = []
```

And now, you should be able to add your existing dependencies to the `pyproject.toml`, either manually or with the `uv add` command. After all the dependencies are specified in `pyproject.toml` you can run `uv sync` to make sure everything is installed in your environment.

Adding dev dependencies

`uv` also supports adding development dependencies to the project:

```
> uv add --dev pytest pytest-django
Resolved 11 packages in 8ms
Prepared 5 packages in 0.91ms
Installed 5 packages in 12ms
+ iniconfig==2.0.0
+ packaging==24.1
```

```
+ pluggy==1.5.0
+ pytest==8.3.2
+ pytest-django==4.8.0
```

You can now run your tests with

```
uv run pytest
```

The dev dependencies are saved in the `tool.uv.dev-dependencies` list in your `pyproject.toml` :

```
[tool.uv]
dev-dependencies = [
    "pytest>=8.3.2",
    "pytest-django>=4.8.0",
]
```

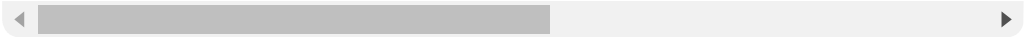
Installing dependencies in production

`uv sync` installs development dependencies by default. Because of this, it's good practice to instruct `uv` not to install dev dependencies in production using the `--no-dev` flag:

```
> uv sync --no-dev --locked
Resolved 11 packages in 2ms
Uninstalled 5 packages in 35ms
- iniconfig==2.0.0
- packaging==24.1
- pluggy==1.5.0
- pytest==8.3.2
- pytest-django==4.8.0
```

The `--locked` flag is recommended because it makes sure your lock file is in sync with your dependency definitions. If it is not `uv sync` will fail:


```
> uv run --locked python manage.py check
error: The lockfile at `uv.lock` needs to be updated, but `--loc
```



This way, only the package in the versions specified in your lock file will get installed, making your builds properly reproducible!

If you use `uv run` without `uv sync` to run your program in production, include the `--no-dev` and `--locked` flags the same way as you would with `uv sync` :

```
> uv run --no-dev --locked gunicorn fedidevs.wsgi
[2024-10-11 10:49:48 +0100] [9243] [INFO] Starting gunicorn 23.0
[2024-10-11 10:49:48 +0100] [9243] [INFO] Listening at: unix:fed
[2024-10-11 10:49:48 +0100] [9243] [INFO] Using worker: gthread
[2024-10-11 10:49:48 +0100] [9280] [INFO] Booting worker with pi
```



Hint: If you don't mind installing dependencies and running your app in a single step, you can omit the `uv sync` command.

`uv run --no-dev --locked` will ensure that all packages are installed before running.

Installing dependencies in CI

Like in production, you also want to make sure your tests are as reproducible as possible so the `--locked` flag is good practice, but since your dev dependencies likely include tools for testing you usually don't add `--no-dev` like you do in production:

```
> uv sync --locked
```

GitHub Actions

If you are running your tests on GitHub actions then using [Astral's official setup-uv action](#) is the easiest way:

```
jobs:
  test:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v4
      - name: Install uv
        uses: astral-sh/setup-uv@v3
        with:
          enable-cache: true
          version: "latest"
      - name: Install dependencies
        run: uv sync --locked
      - name: Collect static files
        run: uv run python manage.py collectstatic
      - name: Run Tests
        run: uv run pytest
```

I like to have a separate `uv sync` step in tests so that it's clearer how long it took to install dependencies and how long to run any of the commands.

If you don't want `uv` to also manage the Python version, you can use the [setup-python action](#):

```
jobs:
  test:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v4
      - name: Setup Python
        uses: actions/setup-python@v5
```



```
with:
  python-version: '3.13'
- name: Install uv
  uses: astral-sh/setup-uv@v3
  with:
    enable-cache: true
    version: "latest"
```

Using .python-version

To avoid having multiple sources of truth for the Python version, you can create a `.python-version` file with the desired version and both `uv` and `setup-python` will use it.

```
> cat .python-version
3.13.0
```

Upgrading dependencies

To upgrade your dependencies, run `uv lock --upgrade`, and it will update your `uv.lock` file based on the constraints in your dependencies definition:

```
> uv lock --upgrade
Resolved 100 packages in 784ms
Updated charset-normalizer v3.3.2 -> v3.4.0
Updated coverage v7.6.1 -> v7.6.2
Updated django v5.1.1 -> v5.1.2
Updated django-cotton v1.1.2 -> v1.1.3
Updated gevent v24.2.1 -> v24.10.2
Updated model-bakery v1.19.5 -> v1.20.0
Updated openai v1.51.1 -> v1.51.2
Updated sentry-sdk v2.15.0 -> v2.16.0
Updated textual v0.82.0 -> v0.83.0
Updated zope-interface v7.0.3 -> v7.1.0
```

Hint: Keep your dependency definitions in the `pyproject.toml` file without upper bounds to make updating packages as easy as possible:

```
dependencies = [  
    "django>=5.1.1",  
]
```

If your project breaks after running `uv lock --upgrade` you can add the upper bound to the problematic package. As an example, if you have issues with Django 5.1 you can do:

```
dependencies = [  
    "django<5.1",  
]
```

This way, you can temporarily keep Django on the latest 5.0.x version while you wait for the issue to be resolved upstream or if the upgrade will require a larger investment (in the case of Django 5.1, this is unlikely!).

Once the issue has been resolved, you must manually remove the upper bound from `pyproject.toml` for `uv lock --upgrade` to upgrade the version in `uv.lock`.

Avoiding `uv run`

Writing `uv run` gets very old very fast, but there are a few options to make the experience a bit nicer.

1. Aliases

You can alias `uv run` to something shorter like `uvr` :

```
alias uvr="uv run"
```

Or for Django use cases, you can define `uvm` like so:

```
alias uvm="uv run python manage.py"
```

So that you can run the `manage.py` file with only three letters:

```
uvm runserver
```

2. Adjusting the shebang line in `manage.py`

You can change the `#!/usr/bin/env python` at the top of `manage.py` into `#!/usr/bin/env -S uv run` to force invocations to use `uv run`.

```
./manage.py runserver
```

I learned about this trick from Jeff Triplett's blog [Python UV run with shebangs](#). ❤️

Using `uv run --with`

`uv run` has another trick up its sleeve: an optional `--with` parameter that allows you to run your project with a different package version. This is super useful if you want to quickly verify if your project works with a newer (or older) version of Django:

```
> uv run python manage.py version
5.1
> uv run --with 'django<5' manage.py version
4.2.15
> uv run --with 'django<5' pytest # to run your tests on the latest
```

Fin

This is a really exciting time for Python and Python packaging! If you want to learn more about `uv` check out the following links:

- [The official documentation](#)
- Simon Willison wrote some [notes on uv](#)
- Jeff Triplet also wrote about the [uv updates](#)

If you have any other tricks or ideas to simplify your Django workflows, [let me know](#), and I'll add your suggestions to the blog post!

- Thank you [Hynek](#) for [letting me know about](#) `--locked` ❤️!

Join the  [Newsletter](#) or subscribe through [RSS](#).

Get in touch through [Email](#), [Mastodon](#), [Twitter](#), or [LinkedIn](#).

This site is ❤️ [open source](#)!