django-email-relay

Josh Thomas

Apr 09, 2024

CONTENTS

`django-email-relay` enables Django projects without direct access to a preferred SMTP server to use that server for email dispatch.

It consists of two parts:

1. A Django app with a custom email backend that stores emails in a central database queue. This is what you will use on all the distributed Django projects that you would like to give access to the preferred SMTP server.
2. A relay service that reads from this queue to orchestrate email sending. It is available as either a standalone Docker image or a management command to be used within a Django project that does have access to the preferred SMTP server.

CHAPTER

ONE

TABLE OF CONTENTS

1.1 Why?

At The Westervelt Company, we primarily host our Django applications in the cloud. The majority of the emails we send are to internal Westervelt employees. Prior to developing and using `django-email-relay`, we were using an external Email Service Provider (ESP) to send these emails. This worked well enough, but we ran into a few issues:

- Emails sent by our applications had a tendency to sometimes be marked as spam or otherwise filtered by our company's email provider, which makes using an ESP essentially pointless.
- As a way to combat phishing emails, we treat and process internal and external emails differently. This meant that in order for our applications' transactional emails to be treated as internal, adjustments and exceptions would need to be made, which our security team was not comfortable with.

We have an internal SMTP server that can be used for any application deployed on-premises, which bypasses most of these issues. However, it is not, and there are no plans to make it publicly accessible – either through opening firewall ports or by using a service like Tailscale. This meant that we needed to find another way to route emails from our cloud-hosted Django applications to this internal SMTP server.

After discussing with our infrastructure and security team, we thought about what would be the simplest and most straightforward to develop and deploy while also not compromising on security. Taking inspiration from another Django package, `django-mailer`, we decided that a solution utilizing a central database queue that our cloud-hosted Django applications can use to store emails to be sent and a relay service that can be run on-premises that reads from that queue would fulfill those requirements. This is what `django-email-relay` is.

1.1.1 Benefits

- By utilizing an internal SMTP server for email dispatch, `django-email-relay` reduces the security risks associated with using external ESPs, aligning with stringent corporate security policies.
- The relay service is available as either a standalone Docker image or a Django management command, giving you the flexibility to choose the deployment method that suits your infrastructure.
- Emails are stored in the database as part of a transaction. If a transaction fails, the associated email records can be rolled back, ensuring data consistency and integrity.
- By using an internal SMTP server, you are less likely to have your emails marked as spam or filtered by company-specific policies, ensuring more reliable delivery, especially if your primary audience is internal employees.
- By utilizing an existing internal SMTP server, you can potentially reduce costs associated with third-party ESPs.

1.1.2 Limitations

As `django-email-relay` is based on `django-mailer`, it shares a lot of the same limitations, detailed here. Namely:

- Since file attachments are stored in a database, large attachments can potentially cause space and query issues.
- From the Django applications sending emails, it is not possible to know whether an email has been sent or not, only whether it has been successfully queued for sending.
- Emails are not sent immediately but instead saved in a database queue to be used by the relay service. This means that emails will not be sent unless the relay service is started and running.
- Due to the distributed nature of the package and the fact that there are database models, and thus potentially migrations to apply, care should be taken when upgrading to ensure that all Django projects using `django-email-relay` are upgraded at roughly the same time. See the *Updating* section of the documentation for more information.

1.2 Installation

You should install and setup the relay service provided by `django-email-relay` first before installing and configuring the Django app on your distributed Django projects. In the setup for the relay service, the database will be created and migrations will be run, which will need to be done before the distributed Django apps can use the relay service to send emails.

1.2.1 Relay Service

The relay service provided by `django-email-relay` is responsible for reading a central database queue and sending emails from that queue through an SMTP server. As such, it should be run on infrastructure that has access to the SMTP server you would like to use. There are currently two ways to run the service:

1. A Docker image
2. A `runrelay` management command to be run from within a Django project

If you are using the Docker image, only PostgreSQL is supported. However, when using the management command directly, you can use whatever database you are using with the Django project it is being run from, provided your other externally hosted Django projects that you would like to relay emails for also have access to the same database. If you would like the Docker image to support other databases, please open an issue and it will be considered.

Installation of the relay service differs depending on whether you are using the provided Docker image or the management command.

Docker

A prebuilt Docker image is provided via the GitHub Container Registry, located here:

```
ghcr.io/westerveltco/django-email-relay
```

It can be run any way you would normally run a Docker container, for instance, through the CLI:

```
docker run -d \
  -e "DATABASE_URL=postgres://email_relay_user:email_relay_password@localhost:5432/email_
↪relay_db" \
  -e "EMAIL_HOST=smtp.example.com" \
```

(continues on next page)

(continued from previous page)

```
  -e "EMAIL_PORT=25" \
  --restart unless-stopped \
  ghcr.io/westerveltco/django-email-relay:latest
```

It is recommended to pin to a specific version, though if you prefer, you can ride the lightning by always pulling the `latest` image.

The `migrate` step is baked into the image, so there is no need to run it yourself.

See the documentation *here* for general information about configuring `django-email-relay`, *here* for information about configuring the relay service, and *here* for information specifically related to configuring the relay service as a Docker container.

Django

If you have a Django project already deployed that has access to the preferred SMTP server, you can skip using the Docker image and install the package and use the included `runrelay` management method instead.

1. Install the package from PyPI:

```
pip install django-email-relay
```

2. Add `email_relay` to your `INSTALLED_APPS` setting:

```
INSTALLED_APPS = [
    # ...
    "email_relay",
    # ...
]
```

3. Run the `migrate` management command to create the email relay database:

```
python manage.py migrate
```

4. Run the `runrelay` management command to start the relay service. This can be done in many different ways, for instance, via a task runner, such as Celery or Django-Q2, or using supervisord or systemd service unit file.

```
python manage.py runrelay
```

See the documentation *here* for general information about configuring `django-email-relay`, *here* for information about configuring the relay service, and *here* for information specifically related to configuring the relay service as a Django app.

1.2.2 Django App

For each distributed Django project that you would like to use the preferred SMTP server, you will need to install the `django-email-relay` package and do some basic configuration.

1. Install the package from PyPI:

```
pip install django-email-relay
```

2. Add `email_relay` to your `INSTALLED_APPS` setting:

```
INSTALLED_APPS = [
    # ...
    "email_relay",
    # ...
]
```

3. Add the `RelayDatabaseEmailBackend` to your `EMAIL_BACKEND` setting:

```
EMAIL_BACKEND = "email_relay.backend.RelayDatabaseEmailBackend"
```

4. Add the email relay database to your `DATABASES` setting. A default database alias is provided at `email_relay.conf.EMAIL_RELAY_DATABASE_ALIAS` which you can import and use:

```
from email_relay.conf import EMAIL_RELAY_DATABASE_ALIAS

DATABASES = {
    # ...
    EMAIL_RELAY_DATABASE_ALIAS: {
        "ENGINE": "django.db.backends.postgresql",
        "NAME": "email_relay_db",
        "USER": "email_relay_user",
        "PASSWORD": "email_relay_password",
        "HOST": "localhost",
        "PORT": "5432",
    },
    # ...
}
```

If you would like to use a different database alias, you will also need to set the `DATABASE_ALIAS` setting within your `DJANGO_EMAIL_RELAY` settings:

```
DATABASES = {
    # ...
    "some_alias": {
        "ENGINE": "django.db.backends.postgresql",
        "NAME": "email_relay_db",
        "USER": "email_relay_user",
        "PASSWORD": "email_relay_password",
        "HOST": "localhost",
        "PORT": "5432",
    },
    # ...
}

DJANGO_EMAIL_RELAY = {
    # ...
    "DATABASE_ALIAS": "some_alias",
    # ...
}
```

4. Add the `EmailDatabaseRouter` to your `DATABASE_ROUTERS` setting:

```
DATABASE_ROUTERS = [
    # ...
```

(continues on next page)

(continued from previous page)

```
    "email_relay.db.EmailDatabaseRouter",
    # ...
]
```

See the documentation *here* for general information about configuring `django-email-relay`.

1.2.3 Requirements

- Python 3.8, 3.9, 3.10, 3.11, or 3.12
- Django 3.2, 4.2, or 5.0
- PostgreSQL (for provided Docker image)

1.3 Usage

Once your Django project is configured to use `email_relay.backend.RelayDatabaseEmailBackend` as its `EMAIL_BACKEND`, sending email is as simple as using Django's built-in ways of sending email, such as the `send_mail` method:

```
from django.core.mail import send_mail

send_mail(
    "Subject here",
    "Here is the message.",
    "from@example.com",
    ["to@example.com"],
    fail_silently=False,
)
```

Any emails sent this way, or one of the other ways Django provides, will be stored in the database queue and sent by the relay service.

See the Django documentation on sending email for more information.

1.3.1 Relay Service Health Check

As mentioned in *limitations*, if the relay service is not running, or otherwise not operational, emails will not be sent out. To help with this, `django-email-relay` provides a way to send a health check ping to a URL of your choosing after each loop of sending emails is complete. This can be used to integrate with a service like Healthchecks.io or UptimeRobot.

To get started, you will need to install the package with the `hc` extra. If you are using the included Docker image, this is done automatically. If you are using the management command directly from a Django project, you will need to adjust your installation command:

```
pip install django-email-relay[hc]
```

At a minimum, you will need to configure which URL to ping after a loop of sending emails is complete. This can be done by setting the *`RELAY_HEALTHCHECK_URL`* setting in your `DJANGO_EMAIL_RELAY` settings:

```
DJANGO_EMAIL_RELAY = {
    "RELAY_HEALTHCHECK_URL": "https://example.com/healthcheck",
}
```

It should be set to the URL provided by your health check service. If available, you should set the schedule of the health check within the service to what you have configured for the relay service's *EMPTY_QUEUE_SLEEP* setting, which is `30` seconds by default.

There are also a few other settings that can be configured, such as the HTTP method to use, the expected HTTP status code, and the timeout. See the *configuration* section for more information.

1.4 Configuration

Configuration of `django-email-relay` is done through the `DJANGO_EMAIL_RELAY` dictionary in your Django settings.

Depending on whether you are configuring the relay service or the Django app, different settings are available. Some settings are available for both, while others are only available for one or the other. If you configure a setting that does not apply, for instance, if you configure something related to the relay service from one of the distributed Django apps, it will have no effect. See the individual sections for each setting below for more information.

All settings are optional. Here is an example configuration with the default values shown:

```
DJANGO_EMAIL_RELAY = {
    "DATABASE_ALIAS": email_relay.conf.EMAIL_RELAY_DATABASE_ALIAS,  # "email_relay_db"
    "EMAIL_MAX_BATCH": None,
    "EMAIL_MAX_DEFERRED": None,
    "EMAIL_MAX_RETRIES": None,
    "EMPTY_QUEUE_SLEEP": 30,
    "EMAIL_THROTTLE": 0,
    "MESSAGES_BATCH_SIZE": None,
    "MESSAGES_RETENTION_SECONDS": None,
    "RELAY_HEALTHCHECK_METHOD": "GET",
    "RELAY_HEALTHCHECK_STATUS_CODE": 200,
    "RELAY_HEALTHCHECK_TIMEOUT": 5.0,
    "RELAY_HEALTHCHECK_URL": None,
}
```

1.4.1 Configuring the Relay Service

At a minimum, you should configure the relay service's connection to the database and how it will connect to your SMTP server, which, depending on your SMTP server, can include any or all the following Django settings:

- `EMAIL_HOST`
- `EMAIL_PORT`
- `EMAIL_HOST_USER`
- `EMAIL_HOST_PASSWORD`

Additionally, the service can be configured using any setting available to Django by default, for example, if you want to set a default from email (`DEFAULT_FROM_EMAIL`) or a common subject prefix (`EMAIL_SUBJECT_PREFIX`).

Configuration of the relay service differs depending on whether you are using the provided Docker image or the management command within a Django project.

Docker

When running the relay service using Docker, config values are set via environment variables. The names of the environment variables are the same as the Django settings, e.g., to set `DEBUG` to `True`, you would set `-e "DEBUG=True"` when running the container.

For settings that are dictionaries, a `__` is used to separate the keys, e.g., to set `DATABASES["default"]["CONN_MAX_AGE"]` to `600` or 10 minutes, you would set `-e "DATABASES__default__CONN_MAX_AGE=600"`.

For the database connection, `dj-database-url` is used to parse the `DATABASE_URL` environment variable, e.g., `-e "DATABASE_URL=postgres://email_relay_user:email_relay_password@localhost:5432/email_relay_db"`.

Django

When running the relay service from a Django project, config values are read from the Django settings for that project.

1.4.2 DATABASE_ALIAS

Component	Configurable
Relay Service	Yes
Django App	Yes

The database alias to use for the email relay database. This must match the database alias used in your `DATABASES` setting. A default is provided at `email_relay.conf.EMAIL_RELAY_DATABASE_ALIAS`. You should only need to set this if you are using a different database alias.

1.4.3 EMAIL_MAX_BATCH

Component	Configurable
Relay Service	Yes
Django App	No

The maximum number of emails to send in a single batch. The default is `None`, which means there is no limit.

1.4.4 EMAIL_MAX_DEFERRED

Component	Configurable
Relay Service	Yes
Django App	No

The maximum number of emails that can be deferred before the relay service stops sending emails. The default is `None`, which means there is no limit.

1.4.5 EMAIL_MAX_RETRIES

Component	Configurable
Relay Service	Yes
Django App	No

The maximum number of times an email can be deferred before being marked as failed. The default is `None`, which means there is no limit.

1.4.6 EMPTY_QUEUE_SLEEP

Component	Configurable
Relay Service	Yes
Django App	No

The time in seconds to wait before checking the queue for new emails to send. The default is `30` seconds.

1.4.7 EMAIL_THROTTLE

Component	Configurable
Relay Service	Yes
Django App	No

The time in seconds to sleep between sending emails to avoid potential rate limits or overloading your SMTP server. The default is `0` seconds.

1.4.8 MESSAGES_BATCH_SIZE

Component	Configurable
Relay Service	No
Django App	Yes

The batch size to use when bulk creating `Messages` in the database. The default is `None`, which means Django's default batch size will be used.

1.4.9 MESSAGES_RETENTION_SECONDS

Component	Configurable
Relay Service	Yes
Django App	No

The time in seconds to keep `Messages` in the database before deleting them. `None` means the messages will be kept indefinitely, `0` means no messages will be kept, and any other integer value will be the number of seconds to keep messages. The default is `None`.

1.4.10 RELAY_HEALTHCHECK_METHOD

Component	Configurable
Relay Service	Yes
Django App	No

The HTTP method to use for the healthcheck endpoint. *RELAY_HEALTHCHECK_URL* must also be set for this to have any effect. The default is `"GET"`.

1.4.11 RELAY_HEALTHCHECK_STATUS_CODE

Component	Configurable
Relay Service	Yes
Django App	No

The expected HTTP status code for the healthcheck endpoint. *RELAY_HEALTHCHECK_URL* must also be set for this to have any effect. The default is `200`.

1.4.12 RELAY_HEALTHCHECK_TIMEOUT

Component	Configurable
Relay Service	Yes
Django App	No

The timeout in seconds for the healthcheck endpoint. *RELAY_HEALTHCHECK_URL* must also be set for this to have any effect. The default is `5.0` seconds.

1.4.13 RELAY_HEALTHCHECK_URL

Component	Configurable
Relay Service	Yes
Django App	No

The URL to ping after a loop of sending emails is complete. This can be used to integrate with a service like Healthchecks.io or UptimeRobot. The default is `None`, which means no healthcheck will be performed.

1.5 Updating

As `django-email-relay` involves database models and the potential for migrations, care should be taken when updating to ensure that all Django projects using `django-email-relay` are upgraded at roughly the same time. See the *deprecation policy* for more information regarding backward incompatible changes.

When updating to a new version, it is recommended to follow the following steps:

1. Update the relay service to the new version. As part of the update process, the relay service should run any migrations that are needed. If using the provided Docker container, this is done automatically as Django's `migrate` command is baked into the image. When running the relay service from a Django project, you will need to run the `migrate` command yourself, either as part of your deployment strategy or manually.
2. Update all distributed projects to the new version.

1.5.1 Deprecation Policy

Road to v1.0.0

Before `django-email-relay` reaches version 1.0.0, the deprecation policy is a little more relaxed. See the changelog for more information regarding backward incompatible changes.

Any changes that involve models and/or migrations, or anything else that is potentially backward incompatible, will be split across two or more releases:

1. A release that adds the changes in a backward compatible way, with a deprecation warning. This release will be tagged with a minor version bump, e.g., `0.1.0` to `0.2.0`.
2. A release that removes the backward compatible changes and removes the deprecation warning. This release will be tagged with a major version bump, e.g., `0.2.0` to `1.0.0`.

This is unlikely to happen often, but it is important to keep in mind when updating.

A major release does not necessarily mean that there are breaking changes or ones involving models and migrations. You should always check the *changelog* and a version's release notes for more information.

1.6 Contributing

All contributions are welcome! Besides code contributions, this includes things like documentation improvements, bug reports, and feature requests.

You should first check if there is a GitHub issue already open or related to what you would like to contribute. If there is, please comment on that issue to let others know you are working on it. If there is not, please open a new issue to discuss your contribution.

Not all contributions need to start with an issue, such as typo fixes in documentation or version bumps to Python or Django that require no internal code changes, but generally, it is a good idea to open an issue first.

We adhere to Django's Code of Conduct in all interactions and expect all contributors to do the same. Please read the Code of Conduct before contributing.

1.6.1 Setup

The following setup steps assume you are using a Unix-like operating system, such as Linux or macOS, and that you have a *supported* version of Python installed. If you are using Windows, you will need to adjust the commands accordingly. If you do not have Python installed, you can visit python.org for instructions on how to install it for your operating system.

1. Fork the repository and clone it locally.
2. Create a virtual environment and activate it. You can use whatever tool you prefer for this. Below is an example using the Python standard library's `venv` module:

```
python -m venv venv
source venv/bin/activate
```

3. Install `django-email-relay` and the `dev` dependencies in editable mode:

```
python -m pip install --editable '.[dev]'
# or using [just](#just)
just bootstrap
```

1.6.2 Testing

We use `pytest` for testing and `nox` to run the tests in multiple environments.

To run the test suite against the default versions of Python (lower bound of supported versions) and Django (lower bound of LTS versions), run:

```
python -m nox --session "test"
# or using [just](#just)
just test
```

To run the test suite against all supported versions of Python and Django, run:

```
python -m nox --session "tests"
# or using [just](#just)
just testall
```

1.6.3 just

just is a command runner that is used to run common commands, similar to make or invoke. A Justfile is provided at the base of the repository, which contains commands for common development tasks, such as running the test suite or linting.

To see a list of all available commands, ensure just is installed and run the following command at the base of the repository:

```
just
```

Releasing a New Version

When it comes time to cut a new release, follow these steps:

1. Create a new git branch off of main for the release.

   Prefer the convention release-<version>, where <version> is the next incremental version number (e.g. release-v0.3.0 for version 0.3.0).

   ```
   git checkout -b release-v<version>
   ```

   However, the branch name is not *super* important, as long as it is not main.

2. Update the version number across the project using the bumpver tool. See *this section* for more details about choosing the correct version number.

   The pyproject.toml in the base of the repository contains a [tool.bumpver] section that configures the bumpver tool to update the version number wherever it needs to be updated and to create a commit with the appropriate commit message.

   bumpver is included as a development dependency, so you should already have it installed if you have installed the development dependencies for this project. If you do not have the development dependencies installed, you can install them with either of the following commands:

   ```
   python -m pip install --editable '.[dev]'
   # or using [just](CONTRIBUTING.md#just)
   just bootstrap
   ```

   Then, run bumpver to update the version number, with the appropriate command line arguments. See the bumpver documentation for more details.

   Note: For any of the following commands, you can add the command line flag --dry to preview the changes without actually making the changes.

   Here are the most common commands you will need to run:

   ```
   bumpver update --patch  # for a patch release
   bumpver update --minor  # for a minor release
   bumpver update --major  # for a major release
   ```

   To release a tagged version, such as a beta or release candidate, you can run:

```
bumpver update --tag=beta
# or
bumpver update --tag=rc
```

Running these commands on a tagged version will increment the tag appropriately, but will not increment the version number.

To go from a tagged release to a full release, you can run:

```
bumpver update --tag=final
```

3. Ensure the *CHANGELOG* is up to date. If updates are needed, add them now in the release branch.
4. Create a pull request from the release branch to `main`.
5. Once CI has passed and all the checks are green , merge the pull request.
6. Draft a new release on GitHub.

   Use the version number with a leading `v` as the tag name (e.g. `v0.3.0`).

   Allow GitHub to generate the release title and release notes, using the 'Generate release notes' button above the text box. If this is a final release coming from a tagged release (or multiple tagged releases), make sure to copy the release notes from the previous tagged release(s) to the new release notes (after the release notes already generated for this final release).

   If this is a tagged release, make sure to check the 'Set as a pre-release' checkbox.
7. Once you are satisfied with the release, publish the release. As part of the publication process, GitHub Actions will automatically publish the new version of the package to PyPI.

Choosing the Next Version Number

We try our best to adhere to Semantic Versioning, but we do not promise to follow it perfectly (and let's be honest, this is the case with a lot of projects using SemVer).

In general, use your best judgement when choosing the next version number. If you are unsure, you can always ask for a second opinion from another contributor.

1.7 Justfile

This project uses Just as a command runner.

The following commands are available:

- *bootstrap*
- *copier-copy*
- *copier-update*
- *copier-update-all*
- *coverage*
- *docs-build*
- *docs-install*
- *docs-serve*

- *fmt*
- *lint*
- *makemigrations*
- *manage*
- *migrate*
- *pup*
- *test*
- *testall*
- *types*

1.7.1 Commands

```
$ just --list
```

```
Available recipes:
    bootstrap
    copier-copy TEMPLATE_PATH DESTINATION_PATH="." # apply a copier template to project
    copier-update ANSWERS_FILE *ARGS # update the project using a copier answers file
    copier-update-all *ARGS # loop through all answers files and update the project␣
↪using copier
    coverage
    docs-build LOCATION="docs/_build/html"
    docs-install
    docs-serve
    fmt                     # format justfile
    lint                    # run pre-commit on all files
    makemigrations *APPS
    mm *APPS                # alias for `makemigrations`
    manage *COMMAND
    migrate *ARGS
    pup
    test *ARGS
    testall *ARGS
    types
```

bootstrap

```
$ just bootstrap
```

```
bootstrap:
    @just pup
    python -m uv pip install --editable '.[dev]'
```

copier-copy

```
$ just copier-copy
```

```
# apply a copier template to project
copier-copy TEMPLATE_PATH DESTINATION_PATH=".":
    copier copy {{ TEMPLATE_PATH }} {{ DESTINATION_PATH }}
```

copier-update

```
$ just copier-update
```

```
# update the project using a copier answers file
copier-update ANSWERS_FILE *ARGS:
    copier update --trust --answers-file {{ ANSWERS_FILE }} {{ ARGS }}
```

copier-update-all

```
$ just copier-update-all
```

```
# loop through all answers files and update the project using copier
@copier-update-all *ARGS:
    for file in `ls .copier/`; do just copier-update .copier/$file "{{ ARGS }}"; done
```

coverage

```
$ just coverage
```

```
coverage:
    python -m nox --session "coverage"
```

docs-build

```
$ just docs-build
```

```
@docs-build LOCATION="docs/_build/html":
    just _cog
    sphinx-build docs {{ LOCATION }}
```

docs-install

```
$ just docs-install
```

```
@docs-install:
    @just pup
    python -m uv pip install 'django-email-relay[docs] @ .'
```

docs-serve

```
$ just docs-serve
```

```
@docs-serve:
    #!/usr/bin/env sh
    just _cog
    if [ -f "/.dockerenv" ]; then
        sphinx-autobuild docs docs/_build/html --host "0.0.0.0"
    else
        sphinx-autobuild docs docs/_build/html --host "localhost"
    fi
```

fmt

```
$ just fmt
```

```
# format justfile
fmt:
    just --fmt --unstable
```

lint

```
$ just lint
```

```
# run pre-commit on all files
lint:
    python -m nox --session "lint"
```

makemigrations

```
$ just makemigrations
```

```
makemigrations *APPS:
    @just manage makemigrations {{ APPS }}
```

manage

```
$ just manage
```

```
manage *COMMAND:
    #!/usr/bin/env python
    import sys

    try:
        from django.conf import settings
        from django.core.management import execute_from_command_line
    except ImportError as exc:
        raise ImportError(
            "Couldn't import Django. Are you sure it's installed and "
            "available on your PYTHONPATH environment variable? Did you "
            "forget to activate a virtual environment?"
        ) from exc

    settings.configure(INSTALLED_APPS=["email_relay"])
    execute_from_command_line(sys.argv + "{{ COMMAND }}".split(" "))
```

migrate

```
$ just migrate
```

```
migrate *ARGS:
    @just manage migrate {{ ARGS }}
```

pup

```
$ just pup
```

```
pup:
    python -m pip install --upgrade pip uv
```

test

```
$ just test
```

```
test *ARGS:
    python -m nox --session "test" -- "{{ ARGS }}"
```

testall

```
$ just testall
```

```
testall *ARGS:
    python -m nox --session "tests" -- "{{ ARGS }}"
```

types

```
$ just types
```

```
types:
    python -m nox --session "mypy"
```