warpdrive Documentation

Release 0.26.4

Graham Dumpleton

December 19, 2016

Contents

The `warpdrive` project provides scripts for implementing a build and deployment system for Python web applications. It targets local development, as well as deployment directly to a managed host, a Docker hosting service, a more comprehensive container application platform, or a platform as a service (PaaS) provider.

In short, `warpdrive` aims to make working on and deploying Python web applications easy. It does this by managing for you the tasks of building up everything required to deploy your application, including Docker images, and then also managing the startup of your Python web application.

Builtin support is provided for the most popular Python WSGI servers, with hooking mechanisms to allow you to also make use of custom Python web servers, including ASYNC based web servers or frameworks.

Want to get a quick feel for what `warpdrive` can do for you, check out Getting Started.

CHAPTER 1

Getting Started

To illustrate what `warpdrive` is all about a simple demonstration is in order. For that we are going to first create a new Django web application project and get it running.

1.1 Creating a new Django project

To create a new Django project a number of steps are required. The first of these is to create the Django project itself.

```
$ django-admin startproject mydjangosite
```

The `startproject` admin command creates the new project in a fresh subdirectory. For the next steps we need to be in that sub directory.

```
$ cd mydjangosite/
```

Because Django sites will usually be backed by a database for persistent storage, we next need to initialise the database. For the default project the database is an instance of SQLite stored directly in the local file system.

```
$ python manage.py migrate
Operations to perform:
  Apply all migrations: admin, contenttypes, auth, sessions
Running migrations:
  Rendering model states... DONE
  Applying contenttypes.0001_initial... OK
  Applying auth.0001_initial... OK
  Applying admin.0001_initial... OK
  Applying admin.0002_logentry_remove_auto_add... OK
  Applying contenttypes.0002_remove_content_type_name... OK
  Applying auth.0002_alter_permission_name_max_length... OK
  Applying auth.0003_alter_user_email_max_length... OK
  Applying auth.0004_alter_user_username_opts... OK
  Applying auth.0005_alter_user_last_login_null... OK
  Applying auth.0006_require_contenttypes_0002... OK
  Applying auth.0007_alter_validators_add_error_messages... OK
  Applying sessions.0001_initial... OK
```

Once the database has been initialised, we need to create a super user account so that we will be able to login using the Django admin interface.

```
$ python manage.py createsuperuser
Username (leave blank to use 'grumpy'):
Email address: grumpy@example.com
```

```
Password:
Password (again):
Superuser created successfully.
```

Finally, to start up the web site we can run the Django development server.

```
$ python manage.py runserver
Performing system checks...

System check identified no issues (0 silenced).
April 08, 2016 - 04:08:52
Django version 1.9.5, using settings 'mydjangosite.settings'
Starting development server at http://127.0.0.1:8000/
Quit the server with CONTROL-C.
```

Visting the root of the the site you will be presented with the Django 'It Worked!' page.

To get access to the Django admin interface, go to the URL `http://127.0.0.1:8000/admin`. Here you will be presented with the login page for the Django admin interface.

As we are using the builtin Django development server the styling for the login page should look correct as the development server automatically worries about static file assets such as style sheets.

As you all hopefully know, the Django development server should not though be used for a production system. Switching from the Django development server to a production grade server and real world deployment is where things can very quickly get a lot more complicated.

Not only do you have to worry about how to set up the particular Python WSGI server you choose to use, you also have to make configuration changes to your Django project to enable you to collect together any static file assets for a web server to host. For a WSGI server which doesn't provide a builtin way of hosting static files, you would also need to add in an appropriate WSGI middleware to your Django application to handle serving up of the static files.

To explain all these extra steps and how to set up a particular WSGI server is out of scope for this discussion, but then it isn't important anyway. This is because `warpdrive` will handle all that for you.

1.2 Packages and virtual environments

In the prior instructions, one detail which was not covered was the step of actually installing Django. It was assumed that you had already installed it.

When working on a complex project, Django will not be the only Python package you will need to install. Installing every package separately is a laborious process. The more typical way installation of required packages is handled is by listing them in a `requirements.txt` file. This is an input file that can be supplied to `pip`, with `pip` installing all the packages in one go.

We therefore create the `requirements.txt` file and into it add:

```
Django
```

As your project grows this is where you would add new packages. Each time you add new packages you would need to rerun `pip` against the file to update your Python installation.

Equally as important as being able to easily install packages is where you install them. As a general rule, it is a bad idea to install packages into you actual Python installation. Best practice is to always use a Python virtual environment.

Using a distinct Python virtual environment for your application ensures you are able to install the actual versions of packages you need for that application. You do not have a problem of conflicting requirements arising from multiple applications sharing the same Python installation.

On top of using `pip` you also need to be familiar with tools such as `virtualenv` or `pyvenv`.

1.3 Working in a local environment

As you can already see, there are lots of little steps you need to start remembering and would have to replicate when it comes to deploying your web application to a production environment.

This is where `warpdrive` can help out, as it knows how to run many of these tasks for you, or can at least capture the knowledge of what needs to be run, and provides you with a simple interface to run them all at the appropriate times.

Lets now start over and see how you would use `warpdrive` to work on the same project.

The first step is to get `warpdrive` installed. To install `warpdrive`, use `pip`.

```
pip install warpdrive
```

This can be into your main Python installation, or you can create a dedicated Python virtual environment which contains only `warpdrive`.

Next we are going to add some extra lines to your login shell profile. If using `bash` you should add this to the end of `~/.bash_profile`:

```
WARPDRIVE=$HOME/Python/warpdrive/bin/warpdrive
export WARPDRIVE

source `$WARPDRIVE rcfile`
```

The `WARPDRIVE` variable should be set to where the `warpdrive` command was installed. In the case of using a Python virtual environment, there is no need to activate the Python virtual environment you installed `warpdrive` into. What you are adding to the login shell profile will ensure that `waprdrive` always works without you needing to do that.

That completes the once off initial installation of `warpdrive`. Create a new shell so the updated login shell profile is picked up and you are good to go.

To start out with `warpdrive` we first need to set up the project for your application. While in the Django project directory, now run:

```
warpdrive project mydjangosite
```

This should update your command prompt to include `(warpdrive+mydjangosite)`. This is done so you know what environment you are working in.

What this command does is create a Python virtual environment for you, specifically dedicated to the named application. It will also set a number of environment variables which will allow you to run further operations on your project no matter what directory you are in.

At this point nothing has been installed that your project requires so you aren't yet ready to work on it. To get the project ready to work on and run, all you need to do though is run `warpdrive build`.

```
(warpdrive+mydjangosite) $ warpdrive build
 -----> Installing dependencies with pip (requirements.txt)
Collecting Django (from -r requirements.txt (line 1))
  Downloading Django-1.9.5-py2.py3-none-any.whl (6.6MB)
    100% || 6.6MB 968kB/s
Installing collected packages: Django
Successfully installed Django-1.9.5
Collecting mod-wsgi
  Downloading mod_wsgi-4.5.1.tar.gz (1.8MB)
```

```
    100% || 1.8MB 1.2MB/s
Installing collected packages: mod-wsgi
  Running setup.py install for mod-wsgi ... done
Successfully installed mod-wsgi-4.5.1
 -----> Collecting static files for Django
Copying '...'
...

56 static files copied to '.../warpdrive+mydjangosite/tmp/django/static'.
```

This will install all the Python packages listed in the `requirements.txt` file, as well as automatically run any special steps required by Django to get an application ready to use, such as running the Django admin command `collectstatic`.

It should be pointed out that although `warpdrive` is running special steps here related to Django, it isn't Django specific. It will automatically detect when certain web frameworks are being used and as necessary run any special steps they require. If your web framework isn't supported, or you have your own custom steps, then `warpdrive` can be told about them and trigger them as part of the build as well.

One example of where you will want to capture such commands is those special steps we ran to initialise the database and create the initial super user. You could run these explicitly again, but it is better to capture them in a script and add it to your application source code. You can then have `warpdrive` run them for you, ensuring that the correct environment has been set up so that it will work.

What we are going to therefore do is create what is called an action hook. These are executable programs which `warpdrive` will run when needed. For any special setup steps for the application or database we are going to add them to the file `.warpdrive/action_hooks/setup`.

```
#!/bin/bash

echo " -----> Running Django database migration"

python manage.py migrate

if [ x"$DJANGO_ADMIN_USERNAME" != x"" ];
then
    echo " -----> Creating predefined Django super user"
    (cat - | python manage.py shell) << !
from django.contrib.auth.models import User;
User.objects.create_superuser('$DJANGO_ADMIN_USERNAME',
                              '$DJANGO_ADMIN_EMAIL',
                              '$DJANGO_ADMIN_PASSWORD')
!
else
    if (tty > /dev/null 2>&1); then
        echo " -----> Running Django super user creation"
        python manage.py createsuperuser
    fi
fi
```

Having created the script and made it executable, we now run the command `warpdrive setup`.

```
(warpdrive+mydjangosite) $ warpdrive setup
 -----> Running Django database migration
Operations to perform:
  Apply all migrations: admin, contenttypes, auth, sessions
Running migrations:
  Rendering model states... DONE
  Applying contenttypes.0001_initial... OK
```

```
  Applying auth.0001_initial... OK
  Applying admin.0001_initial... OK
  Applying admin.0002_logentry_remove_auto_add... OK
  Applying contenttypes.0002_remove_content_type_name... OK
  Applying auth.0002_alter_permission_name_max_length... OK
  Applying auth.0003_alter_user_email_max_length... OK
  Applying auth.0004_alter_user_username_opts... OK
  Applying auth.0005_alter_user_last_login_null... OK
  Applying auth.0006_require_contenttypes_0002... OK
  Applying auth.0007_alter_validators_add_error_messages... OK
  Applying sessions.0001_initial... OK
 -----> Running Django super user creation
Username (leave blank to use 'grumpy'):
Email address: grumpy@example.com
Password:
Password (again):
Superuser created successfully.
```

We can now startup the Django web application using `warpdrive start`.

```
(warpdrive+mydjangosite) $ warpdrive start
 -----> Configuring for server type of 'auto'
 -----> Default server for 'auto' is 'mod_wsgi'
 -----> Running server script start-mod_wsgi
 -----> Executing server command 'mod_wsgi-express start-server
     --log-to-terminal --startup-log --port 8080 --application-type module
     --entry-point mydjangosite.wsgi --callable-object application
     --url-alias /static/ .../warpdrive+mydjangosite/tmp/django/static/'
Server URL         : http://localhost:8080/
Server Root        : /tmp/mod_wsgi-localhost:8080:502
Server Conf        : /tmp/mod_wsgi-localhost:8080:502/httpd.conf
Error Log File     : /dev/stderr (warn)
Startup Log File   : /dev/stderr
Request Capacity   : 5 (1 process * 5 threads)
Request Timeout    : 60 (seconds)
Queue Backlog      : 100 (connections)
Queue Timeout      : 45 (seconds)
Server Capacity    : 20 (event/worker), 20 (prefork)
Server Backlog     : 500 (connections)
Locale Setting     : en_AU.UTF-8
[Fri Apr 08 17:52:52.946943 2016] [mpm_prefork:notice] [pid 25978]
    AH00163: Apache/2.4.18 (Unix) mod_wsgi/4.5.1 Python/2.7.10
    configured -- resuming normal operations
[Fri Apr 08 17:52:52.947336 2016] [core:notice] [pid 25978]
    AH00094: Command line: 'httpd (mod_wsgi-express)
    -f /tmp/mod_wsgi-localhost:8080:502/httpd.conf
    -E /dev/stderr -D FOREGROUND'
```

We again have the Django web application running, but this time the Django development server is not being used. Instead `mod_wsgi-express` is being used. You will note though that you did not have to do anything to configure Apache or mod_wsgi. This is because `warpdrive` and `mod_wsgi-express` together have done that for you.

Because `mod_wsgi-express` is being used, the same setup can be used for a production deployment as well.

Even though it is a production grade WSGI server, it is still quite suitable for use in a development environment, and using the same WSGI server in both development and production would actually be preferred. This is because being the same WSGI server you are now more likely to uncover problems in development, rather than only uncovering them when you switch WSGI servers and move to production.

As to features like automatic source code reloading which the Django development server offerred, this can easily be enabled using an environment, variable.

```
(warpdrive+mydjangosite) $ MOD_WSGI_RELOAD_ON_CHANGES=1 warpdrive start
 -----> Configuring for server type of 'auto'
 -----> Default server for 'auto' is 'mod_wsgi'
 -----> Running server script start-mod_wsgi
 -----> Executing server command 'mod_wsgi-express start-server
     --log-to-terminal --startup-log --port 8080 --reload-on-changes
     --application-type module --entry-point mydjangosite.wsgi
     --callable-object application
     --url-alias /static/ .../warpdrive+mydjangosite/tmp/django/static/'
Server URL         : http://localhost:8080/
Server Root        : /tmp/mod_wsgi-localhost:8080:502
Server Conf        : /tmp/mod_wsgi-localhost:8080:502/httpd.conf
Error Log File     : /dev/stderr (warn)
Startup Log File   : /dev/stderr
Request Capacity   : 5 (1 process * 5 threads)
Request Timeout    : 60 (seconds)
Queue Backlog      : 100 (connections)
Queue Timeout      : 45 (seconds)
Server Capacity    : 20 (event/worker), 20 (prefork)
Server Backlog     : 500 (connections)
Locale Setting     : en_AU.UTF-8
[Fri Apr 08 21:11:18.275624 2016] [mpm_prefork:notice] [pid 26330]
    AH00163: Apache/2.4.18 (Unix) mod_wsgi/4.5.1 Python/2.7.10
    configured -- resuming normal operations
[Fri Apr 08 21:11:18.275898 2016] [core:notice] [pid 26330]
    AH00094: Command line: 'httpd (mod_wsgi-express)
    -f /tmp/mod_wsgi-localhost:8080:502/httpd.conf
    -E /dev/stderr -D FOREGROUND'
[Fri Apr 08 11:11:18.533299 2016] [wsgi:error] [pid 26332] monitor
    (pid=26332): Starting change monitor.
```

In addition to automatic source code reloading, `mod_wsgi-express` offers a range of other development focused features builtin which can be enabled. These include interactive post mortem debugging, request auditing, profiling and code coverage.

Although `mod_wsgi-express` offers the most flexibility as far as how it can be configured and was to a degree purpose built for this way of being used, you can if need be flag that you instead want to use other WSGI servers such as `gunicorn`, `uWSGI` and `Waitress`. In all cases when in automatic mode, as above, all the details of how to start up the WSGI server are handled for you. This includes using whatever means is appropriate for handling serving up of static files, without you needing to make changes in your application code to support that through special WSGI middleware.

For example, if it had been defined that `Waitress` should instead be used the result would be as follows.

```
(warpdrive+mydjangosite) $ warpdrive start
 -----> Configuring for server type of 'auto'
 -----> Default server for 'auto' is 'waitress'
 -----> Running server script start-waitress
 -----> Executing server command 'waitress-serve --port 8080 --threads=5
     whitenoise_wrapper:application'
serving on http://0.0.0.0:8080
```

Don't like how `warpdrive` automatically works out what options need to be supplied to the WSGI server, well you can disable that as well. In that case `warpdrive` will still start the WSGI server, but will only supply the absolute minimum options to have the WSGI server listen on the correct port and log output appropriately. You would then supply the specific options you want to use in a configuration file.

What now if you make changes to the source code for your application?

If they are just changing how the code works, but are not touching static files or database models, you can simply run `warpdrive start` again after making the changes.

If you had made changes to the list of Python packages which needed to be installed, or if you made changes to any static files, or added new static files, then you would first run `warpdrive build` again. This will ensure everything is all made up to date again in preparation for running `warpdrive start` again.

If you make changes to database models that would necessitate a database migration. As with setup of the application and database, we want to capture what is required for this as well so we don't have to remember every time. For any special steps related to database migration or otherwise migrating from one version of your application code to another, we are going to use the `.warpdrive/action_hooks/migrate` file.

```
#!/bin/bash

echo " -----> Running Django database migration"

python manage.py migrate
```

We can then run `warpdrive migrate` and not need to worry about the details of what needs to be run, or ensuring that the environment is setup correctly to run it as `warpdrive` will worry about it.

```
$ warpdrive migrate
 -----> Running Django database migration
Operations to perform:
  Apply all migrations: admin, contenttypes, auth, sessions
Running migrations:
  No migrations to apply.
```

Along with actions hooks for `setup` and `migrate`, there are also others for `build` and `deploy` steps. These allow you to specify extra steps to be run when `warpdrive build` and `warpdrive start` are being run.

The `warpdrive` command therefore provides a higher level command which can be used to hide all the steps and ensure that they are reliably performed every time they are required and in the required order.

If not using Django but some other Python web framework, or even a simple WSGI `hello world` application, then how you use `warpdrive` is exactly the same. You therefore have only one set of commands to remember if using different Python web frameworks at different times as the special steps will be captured by the action hooks.

1.4 Building an image for Docker

Once you are finished developing your web application you will next be thinking about how to deploy it. The flavour of the month for that right now is to use Docker to bundle up your Python web application and then host that Docker image in some way.

Creating Docker images on the face of it appears simple, but there are in fact quite a lot of pitfulls and things you can get wrong. It is very easy to stuff things up and you can be left with an insecure environment, something that doesn't perform very well or which isn't very configurable or easy to maintain.

Like how `warpdrive` can manage the tasks of building everything required for your web application and starting any WSGI server in the most appropriate way, it can also manage creating a Docker image for you. By using `warpdrive` to do this you do not need to worry at all about how to write a `Dockerfile`. Instead `warpdrive` does it, employing all the best practices which may exist and providing you with a secure environment.

To create a Docker image, all you need to do is run `warpdrive image`.

```
(warpdrive+mydjangosite) $ warpdrive image mydjangosite
I0408 21:49:39.696772 27066 install.go:236] Using "assemble" installed from "image:///usr/local/s2i/k
I0408 21:49:39.696877 27066 install.go:236] Using "run" installed from "image:///usr/local/s2i/bin/ru
I0408 21:49:39.696900 27066 install.go:236] Using "save-artifacts" installed from "image:///usr/local
---> Installing application source
---> Building application from source
-----> Installing dependencies with pip (requirements.txt)
Collecting Django (from -r requirements.txt (line 1))
Downloading Django-1.9.5-py2.py3-none-any.whl (6.6MB)
Installing collected packages: Django
Successfully installed Django-1.9.5
-----> Collecting static files for Django
Copying '...'

56 static files copied to '/opt/app-root/tmp/django/static'.
---> Fix permissions on application source
```

This obviously requires you to have Docker installed locally, plus you will also need a program installed called Source to Image (S2I).

The S2I program which does all the hard work, uses a special Docker base image which already incorporates `warpdrive` and all the required Python run time and other tools that may be needed. The contents of your application are combined with that to produce the final application image you can then use.

Running the Docker image is then as simple as running `docker run`.

```
(warpdrive+mydjangosite) $ docker run --rm -p 8080:8080 mydjangosite
---> Executing the start up script
 -----> Configuring for server type of 'auto'
 -----> Default server for 'auto' is 'mod_wsgi'
 -----> Running server script start-mod_wsgi
 -----> Executing server command 'mod_wsgi-express start-server
     --log-to-terminal --startup-log --port 8080 --application-type module
     --entry-point mydjangosite.wsgi --callable-object application
     --url-alias /static/ /tmp/django/static/'
[Fri Apr 08 11:57:10.122619 2016] [mpm_event:notice]
    [pid 28:tid 139770362414848] AH00489: Apache/2.4.18 (Unix)
    mod_wsgi/4.4.22 Python/2.7.11 configured -- resuming normal operations
[Fri Apr 08 11:57:10.122789 2016] [core:notice]
    [pid 28:tid 139770362414848] AH00094: Command line: 'httpd
    (mod_wsgi-express)   -f /tmp/mod_wsgi-localhost:8080:1001/httpd.conf
    -E /dev/stderr -D MOD_WSGI_MPM_ENABLE_EVENT_MODULE
    -D MOD_WSGI_MPM_EXISTS_EVENT_MODULE -D MOD_WSGI_MPM_EXISTS_WORKER_MODULE
    -D MOD_WSGI_MPM_EXISTS_PREFORK_MODULE -D FOREGROUND'
```

You can see that the same steps for running your Python web application are followed as when you were working in the local environment. This is because `warpdrive` is being used inside of the container as well. This ensures that how things were run on your local environment are as close as possible to the final deployment.

As before we still need to initialise the database and setup everything for the application. Just like before `warpdrive setup` is used, but this time you need to run it within an initial container. We will use here the running container we just started up.

```
$ docker exec -it trusting_yonath warpdrive setup
 -----> Running Django database migration
Operations to perform:
  Apply all migrations: contenttypes, admin, sessions, auth
Running migrations:
  Rendering model states... DONE
```

```
  Applying contenttypes.0001_initial... OK
  Applying auth.0001_initial... OK
  Applying admin.0001_initial... OK
  Applying admin.0002_logentry_remove_auto_add... OK
  Applying contenttypes.0002_remove_content_type_name... OK
  Applying auth.0002_alter_permission_name_max_length... OK
  Applying auth.0003_alter_user_email_max_length... OK
  Applying auth.0004_alter_user_username_opts... OK
  Applying auth.0005_alter_user_last_login_null... OK
  Applying auth.0006_require_contenttypes_0002... OK
  Applying auth.0007_alter_validators_add_error_messages... OK
  Applying sessions.0001_initial... OK
 -----> Running Django super user creation
Username (leave blank to use 'default'): grumpy
Email address: grumpy@example.com
Password:
Password (again):
Superuser created successfully.
```

Similarly, when necessary, `warpdrive migrate` can be run within a container to perform any database migration.

We were only using a local filesystem database in this case, so if you were using a database like PostgreSQL or MySQL where it was running in a separate container, you would need to link the containers when running.

1.5 Hosting on a Platform as a Service

There is no reason why `warpdrive` couldn't also be used with a Platform as a Service (PasS). The only restriction would be whether the PaaS environment allows you to hook into their own system for building and deploying your web application. They also need to provide a Python installation that has been installed correctly with a shared library for Python if wanting to use `mod_wsgi-express`.

One system which is already supported by `warpdrive` is OpenShift 3. This platform supports the S2I image tool which was used above to create the Docker image. It is therefore a simple matter of telling OpenShift to use the appropriate S2I builder and point it at the Git repository which contains your application source code. How to do that will be covered in more detail elsewhere.