Divio Cloud developer handbook

Documentation

Release 1.0

Daniele Procida

Aug 07, 2020
## Contents

1 Contents .......................................................... 1
   1.1 Tutorials .................................................. 1
   1.2 How-to guides .......................................... 1
   1.3 Reference ................................................ 1
   1.4 Background .............................................. 1

2 Our development/deployment cycle in seven minutes .......... 3

3 About the Divio cloud deployment platform ...................... 5

4 Detailed table of contents ...................................... 7
   4.1 Tutorial .................................................. 7
   4.2 How-to guides .......................................... 38
   4.3 Technical reference .................................... 107
   4.4 Background information ................................ 143
CHAPTER 1

Contents

1.1 Tutorials
Get started with a hands-on introduction to Divio for developers.

1.2 How-to guides
Step-by-step guides for the developer covering key operations and procedures

1.3 Reference
Technical reference - tools, components and commands

1.4 Background
Explanation and discussion of key topics
CHAPTER 2

Our development/deployment cycle in seven minutes
About the Divio cloud deployment platform

Divio is a platform for containerised web projects. Divio’s cloud platform aims to offer developers:

**More reliable deployment** - it’s built in Python and Django, and uses Docker to give application developers a local development environment that is consistent between the Cloud live and test servers - in other words, a system where if it works on your machine, you can expect it to work in production.

**Easier deployment and maintenance** - the Dockerised Cloud platform makes it possible for developers to get their projects online, and to take charge of deployment, maintenance and scaling, without needing the operations or system administrator skills this usually demands to do well.

**Better portability** - the containerisation technology used in Divio projects guarantees portability and means freedom from lock-in to a single provider. A Divio project can be easily deployed on another platform that supports Docker.
4.1 Tutorial

4.1.1 About the tutorial

**Prerequisites**

The tutorial assumes you are comfortable with the command line, and understand the basics of using SSH keys, Git and so on.

**Not a developer?** If you would like a quick overview of the Divio platform features and interface, see our short video Web project management with Divio.

The tutorial will introduce you to the Divio toolchain, and the complete cycle of project creation, development, deployment and management as a Developer, from setting up a project locally to deploying your own application on the cloud.

4.1.2 Choose your pathway

We currently offer three pathways though the tutorial, for Django, Wagtail and Laravel (currently in beta).

The same principles will be used for any language you use for your applications on Divio. We recommend starting with one of these tutorials even if you plan to work with a different language later, as the tutorials have been designed to help you become familiar with key when working with Divio.

4.1.3 Tutorial chapters
Set up the local development environment

This is the part of the tutorial with the greatest potential for stumbling-blocks, because it involves the installation of software. However the components are very reliable and the vast majority of users encounter no problems at all.

If you do run into any difficulties, please don’t hesitate to contact Divio support, who will be glad to help you out.

Why is the local development environment so important?

Docker makes it possible to run and work on a project locally in the same environment as it runs on in the cloud. This side-steps some of the most troublesome problems faced by development teams, in which something works well in development, but then runs into trouble as a result of different environment conditions in production, or when another team member tries to set it up on their own machine.

Using Docker means that not only does every member of the development team work in the same environment - which includes versions of installed packages, environment variables, database and other services - but they’re all in the same environment that the application will have in production.

The local development environment includes the tools and software that allow you to work on your project, testing it as you go on your own computer. This uses Docker, just like our cloud deployment architecture. The local environment is also integrated with the cloud infrastructure - it’s like having a hotline to the cloud.

In this section we will:

- **install Docker**
- **install the Divio CLI**
- **add your public key to the Divio Control Panel**

Older versions of Macintosh OS X and Windows

Older versions of Macintosh OS X and Windows do not support the native Docker application, but require Docker to be run using VirtualBox. This is considerably more complex to set up than Docker running natively. We do not support for this combination. Check the system requirements for Macintosh and Windows.

Before you start

You will need to have the following installed or configured, and know at least the basics of using them, before proceeding:

- Git (see GitHub’s set up Git guide)
- SSH, so that you can provide your public key to a server (GitHub’s guides to setting up SSH)
- Pip, the Python Package Installer

Install Docker and Docker Compose

- Macintosh users: Docker for Mac
- Windows users: Docker for Windows
- Linux users: Docker CE and Docker Compose
(Windows users should consult the checklist below).

Launch Docker. You can check that it’s running correctly with:

```console
docker run --rm busybox true
```

### Additional checklist for Docker installation on Windows

- In Docker’s settings, make sure that it is set to use *Linux containers*.
- Ensure that your Windows user is in the *docker-users* group.
- When you launch Docker, make sure that you do so as a Windows administrator.

You’re now ready to set up the project you created in the previously step in your local environment for development work.

### Install the Divio CLI package

The *Divio CLI application* is installable using Pip. Note that it requires Python 3.6 or higher. Depending on your system, you may need to use `pip3` in the examples below.

```console
pip install divio-cli
```

If you already have it installed, check that they are up-to-date:

```console
pip install --upgrade divio-cli
```

You can do this in a virtual environment if you prefer not to install it globally. Otherwise, it’s a lightweight component and can easily be removed later if you decide you don’t require it.

### Log in

Make sure you are logged in to your account on the Divio Control Panel. If you don’t already have one, now is the time to create it (a Divio account is free to set up and use indefinitely).

The Divio CLI needs to be authenticated with the Control Panel in order to interact with it, using the command:

```console
divio login
```

This will open your browser at `https://control.divio.com/account/desktop-app/access-token/`, where you can copy an access token to paste into the prompt.

### Add your public SSH key to the Control Panel

The Control Panel needs your public key, so that you can interact with our Git server and so on. Visit SSH Keys in the Control Panel. Add your public key. If you’re not sure how to manage SSH keys, see the excellent GitHub articles on how to connect with SSH.

Test that your key is set up correctly: you should receive an interactive access response:

```console
ssh -T git@git.divio.com
```
On to the next step

Now you’re ready to go on to the next step. At this point the tutorial branches, and you can continue working with Python/Django or PHP/Laravel.

- continue with Django
- continue with Wagtail
- continue with PHP/Laravel

Next, select your path through the tutorial.

Python/Django chapters

Create a new Django project

In this section we will create and deploy a new project in the Control Panel using Django, the most popular Python web application framework. The principles covered by the tutorial will apply to any other development stack.

Note: This tutorial is intended to introduce the basics of working with Divio, using Django as an example. It is not a tutorial for Django. However, you don’t need to know Django or Python, or have them installed on your system.

Set up a project in the Cloud

If you have not already done so, you will need to set up the local development environment.

Create the project

In the Divio Control Panel, create a new project.

Any web application in any language can run on Divio, as long as there is nothing that prevents its being Dockerised. However for convenience we also provide a number of ready-to-go project types for applications built in Java, PHP, Node and other languages (and the list is growing).

Select the following options for your project (other options should be left on their default settings):

- Python: Python 3.x
- Project type: Django

Django 2.2

At the time of writing, version 2.2 is Django’s Long-Term Support release, and is guaranteed support until at least April 2022. This is the version currently selected by default in Divio projects.

Divio projects use Git for code management. We provide a Git server that your projects will use by default; you can also choose to use another Git service if you prefer. For this tutorial, use our Git server.

Hit Create. (You may be asked to select a subscription for the project; select the free Developer plan if so.)

It takes a few moments to create the project. During this process, the Control Panel defines the Docker image for your application by adding commits to its repository, assembling its Dockerfile and other files.
Project environments

Your project has two independent server environments, Test and Live. The Test and Live environments have their own services, and unique environment variables to configure access to them. They can be deployed independently, and can also be configured to track different Git branches.

Deploy the project

Once the project has been fully created, use the Deploy button to deploy the Test server.

The deployment process first builds the Docker image from the Dockerfile, and then launches a Docker container from the image. The container environment will include automatically-configured environment variables for connections to services such as the database, media storage and so on.

Typically, this takes a minute or so.

Open the Test environment website

Once you have successfully deployed the Test environment, the Control Panel will indicate this in the interface:

Last deployment successful at . . .

and the project URL in the Test environment pane will display as a link, for example https://tutorial-project-stage.us.aldryn.io. Select the link to access the project.

Since this is your own project, you can use our single-sign-on to log in by selecting Log in with Divio. You’ll see the familiar Django admin for a new project.
The new project doesn’t do anything very useful or interesting yet - but it’s up and running and ready to start working on.

**About deployment**

Any time new changes to the project code are committed to its repository, the Control Panel will indicate this with a message showing the number of undeployed commits for each of its server environments.

New code and configuration changes applied via the Control Panel (to subscriptions, cron jobs, environment variable, domains or other settings) will not take effect on either server environment until it is deployed once again.

If for whatever reason a deployment fails, there will be no down-time - the containers that are currently running will continue running, and the failing changes will not be deployed.

**Explore the Dashboard**

The Divio project Dashboard provides you with access to useful controls and information for the project. They are fairly self-explanatory and at this stage you don’t need to interact with any of them, but it’s worth familiarising yourself with what’s available.

**Set up your project locally**

In this section we will set up locally the cloud project you created earlier.

Obtain the project’s slug (its unique ID) from the Dashboard:

```
tutorial project
  add
  test
```

Alternatively you can use the `divio` command to list your cloud project, which will show their slugs:

```
divio project list
```
Build the project locally

Run the `divio project setup` command (for example if your project slug is `tutorial-project`):

```
divio project setup tutorial-project
```

The Divio CLI will execute a number of steps - this make take a few minutes, depending on how much needs to be downloaded and processed. The Divio CLI tool will build your project locally (see *The project deployment process* for a more detailed description of what’s happening here):

```
Creating workspace
cloning project repository
[...]
downloading remote docker images
[...]
building local docker images
[...]
creating new database container
[...]
syncing and migrating database
[...]
Your workspace is setup and ready to start.
```

cd into the newly-created project directory, where you will find your project code.

Start the local project

Start the project by running `docker-compose up` in the terminal:

```
docker-compose up
Starting tutorial-project_db_1
Performing system checks...
System check identified no issues (0 silenced).
May 19, 2020 - 03:29:06
Django version 2.2.12, using settings 'settings'
Starting development server at http://0.0.0.0:80/
Quit the server with CONTROL-C.
```

Open the project in your web browser by visiting `http://localhost:8000`.

(You may notice above that Django claims to be running on port 80, not port 8000. It is - but that’s only inside the container. The `docker-compose.yml` configuration file is responsible for this port-mapping.)

**Note:** If you didn’t previously log in to the cloud site before setting up the project locally, you’ll need to add a user to the database before you can log in. The *Divio SSO system* allows you to do this from the Django login page with the **Add user** option.

Or, you could run:

```
docker-compose run web manage.py createsuperuser
```

See below for more on the use of `docker-compose`.

If you open a new terminal window and run:
docker ps

it will show you the Docker processes that are running - you will see something like (note that the details will differ):

<table>
<thead>
<tr>
<th>CONTAINER ID</th>
<th>IMAGE</th>
<th>COMMAND</th>
<th>CREATED</th>
<th>STATUS</th>
</tr>
</thead>
<tbody>
<tr>
<td>d6007edba3f32</td>
<td>tutorialproject_web</td>
<td>&quot;/tini -g -- python...&quot;</td>
<td>17 minutes ago</td>
<td>Up 8s</td>
</tr>
<tr>
<td></td>
<td></td>
<td>-seconds 0.0.0.0:8000-&gt;80/tcp tutorialproject_web_</td>
<td></td>
<td></td>
</tr>
<tr>
<td>27ff3e661027</td>
<td>postgres:9.6</td>
<td>&quot;docker-entrypoint...&quot;</td>
<td>17 minutes ago</td>
<td>Up 8s</td>
</tr>
<tr>
<td></td>
<td></td>
<td>-seconds 5432/tcp tutorialproject_db_</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The first container is an instance of the image that you built, just like the one in the cloud deployment. The second is the database, running in its own Docker container.

Once you have successfully logged into the local site, stop the project, using CONTROL-C.

Useful commands

So far, we have used the divio, docker-compose and docker commands. It's good to have a basic familiarity with them and what they do. As you proceed through this tutorial, you may encounter the occasional issue. These commands will help you when this happens.

Note: See our local commands cheat sheet for many useful commands.

Using divio

The divio command is used mainly to manage your local project’s resources and to interact with our Control Panel. You have already used divio project setup and divio project list; you can also use it to do things like push and pull database and media content. Try:

divio project dashboard

See the Divio CLI reference for more.

Using docker

The docker command is mostly used to manage Docker processes, and Docker itself. Mostly, you’ll never need to use it, but it can be useful when you need to understand what Docker is doing on your machine, or for certain operations. You have already used docker ps. Try:

docker info

Using docker-compose

The docker-compose command is used mainly to control and interact with your local project. You will mostly use it to start the local project and open a shell in the local web container. You have already used docker-compose build and docker-compose up.

Just for example, try:
docker-compose run web python manage.py shell

which will open a Django shell in the web container.

You now know how to set up a project in the local environment, and launch it. The next step is to do some development work in the project, test it, and deploy it to the cloud.

Make changes and deploy them

Next, we’re going to install a new package, Django Axes, into the project (Django Axes keeps track of log-in attempts). Then we’ll test it and deploy it to the cloud.

Install a package

To be used in a containerised system, packages must be built into the image, otherwise the next time a container is launched, the package will not be there. The image is built by the Dockerfile, and in our Dockerfile for Django projects, this includes an instruction to process the project’s requirements.in file with Pip. This is where the package needs to be added. Open requirements.in and at the end of it add a new line:

django-axes==3.0.3

It’s important to pin dependencies to a particular version this way; it helps ensure that we don’t run into unwanted surprises if the package is updated, and the new version introduces an incompatibility.

Now you can build the project again by running:

docker-compose build

Configure the Django settings

Django Axes requires that it be listed in the project’s INSTALLED_APPS in settings.py. Add axes to the list:

```python
INSTALLED_APPS.extend([
    "axes",
])
```

A brief explanation of Aldryn Addons

This project uses the optional Aldryn Addons system, which makes it possible for projects to configure themselves. For example, you can find all the configuration that Aldryn Django does for Django settings in addons/aldryn-django/aldryn_config.py. (Aldryn Django is simply a convenience wrapper for Django - the Django used by your project is a wholly standard Django installation obtained from PyPI.)

You don’t have to use Aldryn Addons on Divio; if you prefer to manage settings manually, that will work just as well. However it makes development much faster, as it takes care of all the settings that would otherwise need to be managed correctly for the different cloud environments as well as the local environment.
One advantage of Aldryn Django is that it declutters the settings.py file, removing deployment-related values that are better handled via environment variables, and also provides a guarantee that settings for database, media and so on will always be correct. Aldryn Django’s aldryn_config.py sets them appropriately for each environment, including the local development environment, and also appropriately at each stage of the build/deployment process.

In settings.py, you’ll find the lines:

```python
import aldryn_addons.settings
aldryn_addons.settings.load(locals())
```

These lines load all those settings into the settings module. This includes populating INSTALLED_APPS. A good way to see what settings are applied is via Django’s diffsettings command:

```bash
docker-compose run web python manage.py diffsettings
```

### Run migrations

Django Axes introduces new database tables, so we need to run migrations:

```bash
docker-compose run web python manage.py migrate
```

(As you have probably noticed, we can run all the usual Django management commands, but because we need to run them inside the containerised environment, we precede each one with docker-compose run web.)

### Check the project

If you launch the project again with docker-compose up you’ll find Django Axes in the admin:

![Access attempts](image)

Test it by attempting to log in to the Django admin with an incorrect password.

### Deploy to the cloud

If you are satisfied with your work, you can deploy it to the cloud.

We made changes to two files (requirements.in, settings.py). So:

```bash
git add .
git commit -m "Added Django Axes"
git push
```

On the project Dashboard, you will see that your new commit is listed as 1 Undeployed commit. You can deploy this using the Control Panel, or by running:
When it has finished deploying, you should check the Test server to see that all is as expected. Once you’re satisfied that it works correctly, you can deploy the Live server too:

```bash
divio project deploy live
```

### Using `divio project push/pull`

Your local database has new content, but your cloud database hasn’t been touched by the work you did locally. One very useful function of the Divio CLI is ability to push and pull your database and media storage to and from the cloud environments. For example, try:

```bash
divio project push db
```

This will push the local database to the cloud Test environment. Once the process has completed, you can refresh the cloud Test site; you’ll see that it now has the same content in its database as the local site.

Similarly, you can push/pull media files, and also specify which cloud environment. See the local commands cheat-sheet. A common use-case is to pull live content into the development environment, so that you can test new development with real data.

---

*The next section* looks at some more complex configuration and application integration.

### Configure a more complex application

In the previous section of the tutorial, we added an application and deployed it. However, the installation process was extremely simple and required very minimal configuration.

In practice, adding a Django application to a project will generally require more complex configuration.

We’ll explore this by adding Django Debug Toolbar to the project.

#### Add `django-debug-toolbar` to `requirements.in`

The Django Debug Toolbar installation notes suggest to install it using `pip install django-debug-toolbar`. The latest stable version at the time of writing is 2.2, so add:

```bash
django-debug-toolbar==2.2
```

to `requirements.in`.

As before, run `docker-compose build web` to rebuild the project with the new requirement.

#### Configure `settings.py`

Django Debug Toolbar requires various settings to be configured.
Configure INSTALLED_APPS

Debug Toolbar requires django.contrib.staticfiles and debug_toolbar to be present in INSTALLED_APPS. Is django.contrib.staticfiles already there? There’s an easy way to check: run docker-compose run web python manage.py diffsettings

The Django diffsettings management command shows the differences between your settings and Django’s defaults. In this case it should reassure us that django.contrib.staticfiles is already there as it’s included in Divio Django projects by default, so we just need to add debug_toolbar to INSTALLED_APPS:

```python
INSTALLED_APPS.extend(['debug_toolbar',])
```

Configure middleware settings

The installation documents note that we must set up the middleware, and that it should come as soon as possible in the list “after any other middleware that encodes the response’s content, such as GZipMiddleware.”

A suitable place would be right after django.middleware.gzip.GZipMiddleware, and we can use a little Python list manipulation to insert it there. In addition, it makes sense only to activate the Debug Toolbar middleware when we’re running with Django’s Debug mode, and we’ll check for this (with if DEBUG):

```python
if DEBUG:
    MIDDLEWARE.insert(MIDDLEWARE.index("django.middleware.gzip.GZipMiddleware") + 1, "debug_toolbar.middleware.DebugToolbarMiddleware")
```

This will find the GZipMiddleware in the list, and insert the DebugToolbarMiddleware immediately after it.

Triggering the toolbar

The toolbar should only be triggered if certain conditions are met. By default, it’s only if DEBUG = True and the server IP address is listed in INTERNAL_IPS.

With Docker, we don’t have a way to know what internal IP address a project will have, so we can’t rely on that. However, relying on DEBUG will be enough, so we define a function that will serve as a SHOW_TOOLBAR_CALLBACK callback to replace the default.

At the end of the settings.py file, add:

```python
def show_toolbar(request):
    return DEBUG

DEBUG_TOOLBAR_CONFIG = {"SHOW_TOOLBAR_CALLBACK": show_toolbar}
```

Configure urls.py

We need to include the debug_toolbar.urls in the project’s URL configuration. Our approach here is similar: we only want it active in DEBUG mode, so add this to the end of your project’s urls.py:
from django.conf import settings

if settings.DEBUG:
    from django.urls import include, path
    import debug_toolbar
    urlpatterns = [
        path('__debug__/', include(debug_toolbar.urls)),
    ] + urlpatterns

See the results

And that’s it (Debug Toolbar has no database tables, so you don’t need to run migrations).
Visit the admin to see the Debug Toolbar in action.

Where to go next?

This completes the basic cycle of project creation, development and deployment; you should now be familiar with the fundamental concepts and tools involved.

Other sections of the documentation expand upon them. The how-to guides in particular cover many common operations. And if there’s something you’re looking for but can’t find, please contact Divio support.

Wagtail chapters

Create a new Wagtail Django project

In this section we will create and deploy a new project in the Control Panel using Wagtail, a very popular Django-based content management system framework. The principles covered by the tutorial will apply to any other development stack.

You could equally well create a plain Django project, and install Wagtail in that. However, the Wagtail project type saves some time by setting up a ready-to-go project, with Wagtail automatically installed and configured.
Note: This tutorial assumes some basic familiarity with Wagtail. It is intended to introduce the basics of working with Divio, using Django/Wagtail as an example. It is not a tutorial for learning Wagtail (or Django). For a simpler Django-based introduction, use our basic Django tutorial.

Set up a project in the Cloud

If you have not already done so, you will need to set up the local development environment.

Create the project

In the Divio Control Panel, create a new project.

Any web application in any language can run on Divio, as long as there is nothing that prevents its being Dockerised. However for convenience we also provide a number of ready-to-go project types for applications built in Java, PHP, Node and other languages (and the list is growing).

Select the following options for your project (other options should be left on their default settings):

- **Python:** Python 3.x
- **Project type:** Wagtail

Wagtail 2.9.2

At the time of writing, the latest default version Aldryn Wagtail addon uses Wagtail 2.9.2 - others are also available.

Divio projects use Git for code management. We provide a Git server that your projects will use by default; you can also choose to use another Git service if you prefer. For this tutorial, use our Git server.

Hit Create. (You may be asked to select a subscription for the project; select the free Developer plan if so.)

It takes a few moments to create the project. During this process, the Control Panel defines the Docker image for your application by adding commits to its repository, assembling its Dockerfile and other files.
Project environments

Your project has two independent server environments, Test and Live. The Test and Live environments have their own services, and unique environment variables to configure access to them. They can be deployed independently, and can also be configured to track different Git branches.

Deploy the project

Once the project has been fully created, use the Deploy button to deploy the Test server.

The deployment process first builds the Docker image from the Dockerfile, and then launches a Docker container from the image. The container environment will include automatically-configured environment variables for connections to services such as the database, media storage and so on.

Typically, this takes a minute or so.

Open the Test environment website

Once you have successfully deployed the Test environment, the Control Panel will indicate this in the interface:

Last deployment successful at . . .

and the project URL in the Test environment pane will display as a link, for example https://tutorial-project-stage.us.aldryn.io. Select the link to access the project.

Since this is your own project, you can use our single-sign-on to log in by selecting Log in with Divio. You’ll see the “Welcome to Wagtail” page.

More interesting are the admin pages:

- the Wagtail admin, at /admin, shown below
- the Django admin, at /django-admin
The new project doesn’t do anything very useful or interesting yet - but it’s up and running and ready to start working on.

**About deployment**

Any time new changes to the project code are committed to its repository, the Control Panel will indicate this with a message showing the number of undeployed commits for each of its server environments.

New code and configuration changes applied via the Control Panel (to subscriptions, cron jobs, environment variable, domains or other settings) will not take effect on either server environment until it is deployed once again.

If for whatever reason a deployment fails, there will be no down-time - the containers that are currently running will continue running, and the failing changes will not be deployed.

**Explore the Dashboard**

The Divio project Dashboard provides you with access to useful controls and information for the project. They are fairly self-explanatory and at this stage you don’t need to interact with any of them, but it’s worth familiarising yourself with what’s available.

**Set up your project locally**

In this section we will set up locally the cloud project you created earlier.

Obtain the project’s slug (its unique ID) from the Dashboard:
Alternatively you can use the `divio` command to list your cloud project, which will show their slugs:

```
divio project list
```

### Build the project locally

Run the `divio project setup` command (for example if your project slug is `tutorial-project`):

```
divio project setup tutorial-project
```

The Divio CLI will execute a number of steps - this may take a few minutes, depending on how much needs to be downloaded and processed. The Divio CLI tool will build your project locally (see *The project deployment process* for a more detailed description of what’s happening here):

- Creating workspace
- Cloning project repository
- Downloading remote docker images
- Building local docker images
- Creating new database container
- Syncing and migrating database
- Your workspace is setup and ready to start.

```
cd into the newly-created project directory, where you will find your project code.
```

### Start the local project

Start the project by running `docker-compose up` in the terminal:

```
docker-compose up
Starting tutorial-project_db_1
Performing system checks...
System check identified no issues (0 silenced).
May 19, 2020 - 03:29:06
Django version 2.2.12, using settings 'settings'
Starting development server at http://0.0.0.0:80/
Quit the server with CONTROL-C.
```

4.1. Tutorial
Open the project in your web browser by visiting http://localhost:8000.

(You may notice above that Django claims to be running on port 80, not port 8000. It is - but that’s only inside the container. The docker-compose.yml configuration file is responsible for this port-mapping.)

**Note:** If you didn’t previously log in to the cloud site before setting up the project locally, you’ll need to add a user to the database before you can log in. The Divio SSO system allows you to do this from the Django login page with the **Add user** option.

Or, you could run:

```
docker-compose run web manage.py createsuperuser
```

See below for more on the use of `docker-compose`.

If you open a new terminal window and run:

```
docker ps
```

it will show you the Docker processes that are running - you will see something like (note that the details will differ):

<table>
<thead>
<tr>
<th>CONTAINER ID</th>
<th>IMAGE</th>
<th>COMMAND</th>
<th>CREATED</th>
<th>STATUS</th>
</tr>
</thead>
<tbody>
<tr>
<td>d6007edba322</td>
<td>tutorialproject_web</td>
<td>&quot;/tini -g -- python...&quot;</td>
<td>17 minutes ago</td>
<td>Up 8s</td>
</tr>
<tr>
<td>27ff3e661027</td>
<td>postgres:9.6</td>
<td>&quot;docker-entrypoint...&quot;</td>
<td>17 minutes ago</td>
<td>Up 8s</td>
</tr>
</tbody>
</table>

The first container is an instance of the image that you built, just like the one in the cloud deployment. The second is the database, running in its own Docker container.

Once you have successfully logged into the local site, stop the project, using CONTROL–C.

**Useful commands**

So far, we have used the `divio`, `docker-compose` and `docker` commands. It’s good to have a basic familiarity with them and what they do. As you proceed through this tutorial, you may encounter the occasional issue. These commands will help you when this happens.

**Note:** See our local commands cheat sheet for many useful commands.

**Using divio**

The `divio` command is used mainly to manage your local project’s resources and to interact with our Control Panel. You have already used `divio project setup` and `divio project list`; you can also use it to do things like push and pull database and media content. Try:

```
divio project dashboard
```

See the **Divio CLI reference** for more.
Using **docker**

The `docker` command is mostly used to manage Docker processes, and Docker itself. Mostly, you’ll never need to use it, but it can be useful when you need to understand what Docker is doing on your machine, or for certain operations. You have already used `docker ps`. Try:

```
docker info
```

Using **docker-compose**

The `docker-compose` command is used mainly to control and interact with your local project. You will mostly use it to start the local project and open a shell in the local web container. You have already used `docker-compose build` and `docker-compose up`.

Just for example, try:

```
docker-compose run web python manage.py shell
```

which will open a Django shell in the web container.

You now know how to set up a project in the local environment, and launch it. The next step is to do some development work in the project, test it, and deploy it to the cloud.

**Make changes and deploy them**

Now you have a working Django Wagtail installation. It includes the `Page` model, from `wagtail.core.models`, that typically you would extend to include your own fields.

This process is described in the Wagtail tutorial, and adds a `home` application to the Django project that adds a new page type. We’ll run through the steps here.

**Add a new `home` application**

You’ll notice that we use `docker-compose run web` a lot here, to execute familiar Django commands inside the project’s Docker environment.

**Create the application**

```
docker-compose run web python manage.py startapp home
```

Edit its `models.py` to add a new HomePage model with a `body` field:

```
from django.db import models

from wagtail.core.models import Page
from wagtail.core.fields import RichTextField
from wagtail.admin.edit_handlers import FieldPanel

(continues on next page)
Configure the Django settings

Home needs to be listed in the project's INSTALLED_APPS in settings.py. Add home to the list:

```python
# all Django settings can be altered here
INSTALLED_APPS.extend([
    "home",
])
```

A brief explanation of Aldryn Addons

This project uses the optional Aldryn Addons system, which makes it possible for projects to configure themselves. For example, you can find all the configuration that Aldryn Django does for Django settings in addons/aldryn-django/aldryn_config.py. (Aldryn Django is simply a convenience wrapper for Django - the Django used by your project is a wholly standard Django installation obtained from PyPI.)

You don't have to use Aldryn Addons on Divio; if you prefer to manage settings manually, that will work just as well. However it makes development much faster, as it takes care of all the settings that would otherwise need to be managed correctly for the different cloud environments as well as the local environment.

One advantage of Aldryn Django is that it declutters the settings.py file, removing deployment-related values that are better handled via environment variables, and also provides a guarantee that settings for database, media and so on will always be correct. Aldryn Django's aldryn_config.py sets them appropriately for each environment, including the local development environment, and also appropriately at each stage of the build/deployment process.

In settings.py, you'll find the lines:

```python
import aldryn_addons.settings
aldryn_addons.settings.load(locals())
```

These lines load all those settings into the settings module. This includes populating INSTALLED_APPS. A good way to see what settings are applied is via Django's diffsettings command:

```bash
docker-compose run web python manage.py diffsettings
```

Create migrations and migrate the database

```bash
docker-compose run web python manage.py makemigrations home
docker-compose run web python manage.py migrate home
```
Add templates to the project

Project-level base.html template

In templates, add a base.html:

```html
<!DOCTYPE html>
<head>
  <title>{{ self.title }}</title>
</head>
<body>
  <h1>{% block page_title %}{% endblock %}</h1>
  {% block content %}{% endblock %}
</body>
</html>
```

Application-level templates

In home/templates/home/home_page.html:

```html
{% extends "base.html" %}
{% load wagtailcore_tags %}
{% block page_title %}{{ page.title }}{% endblock %}
{% block content %}{{ page.body|richtext }}{% endblock %}
```

Add a page in the Wagtail admin

In the usual Wagtail way, add a new page under Home, and ensure that in Settings > Sites, the default Site is attached to it.

Deploy to the cloud

If you are satisfied with your work, you can deploy it to the cloud.

We made changes settings.py, added the home application and some templates. So:

```
git add .
git commit -m "Added Home application"
git push
```

On the project Dashboard, you will see that your new commit is listed as 1 Undeployed commit. You can deploy this using the Control Panel, or by running:

```
divio project deploy
```
When it has finished deploying, you should check the Test server to see that all is as expected. Once you’re satisfied that it works correctly, you can deploy the Live server too:

```
divio project deploy live
```

### Using `divio project push/pull`

Your local database has new content, but your cloud database hasn’t been touched by the work you did locally. One very useful function of the Divio CLI is ability to push and pull your database and media storage to and from the cloud environments. For example, try:

```
divio project push db
```

This will push the local database to the cloud Test environment. Once the process has completed, you can refresh the cloud Test site; you’ll see that it now has the same content in its database as the local site.

Similarly, you can push/pull media files, and also specify which cloud environment. See the [local commands cheat-sheet](#). A common use-case is to pull live content into the development environment, so that you can test new development with real data.

### Install a package from pip

Next, we’re going to install a new package, Django Axes, into the project (Django Axes keeps track of log-in attempts). Then we’ll test it and deploy it to the cloud.

To be used in a containerised system, packages must be built onto the image, otherwise the next time a container is launched, the package will not be there. The image is built by the Dockerfile, and in our Dockerfile for Django projects, this includes an instruction to process the project’s `requirements.in` file with Pip. This is where the package needs to be added. Open `requirements.in` and at the end of it add a new line:

```
django-axes==3.0.3
```

It’s important to pin dependencies to a particular version this way; it helps ensure that we don’t run into unwanted surprises if the package is updated, and the new version introduces an incompatibility.

Now you can build the project again by running:

```
docker-compose build
```

### Configure the Django settings

As before, add the application (axes) to the settings:

```python
INSTALLED_APPS.extend([
    [...]
    "axes",
    ])
```

# all Django settings can be altered here

## Chapter 4. Detailed table of contents
Run migrations

docker-compose run web python manage.py migrate axes

Check the project

If you launch the project again with `docker-compose up` you’ll find Django Axes in the admin at `/django-admin`:

Test it by attempting to log in to the Django admin with an incorrect password.

Deploy again

Once more, you need to:

- commit the changes
- push them
- deploy them on the cloud

More complex configuration

See *Configure a more complex application* from the basic Django tutorial pathway. This includes some further configuration examples that it is good to know about.

Where to go next?

This completes the basic cycle of project creation, development and deployment; you should now be familiar with the fundamental concepts and tools involved.

Other sections of the documentation expand upon them. The *how-to guides* in particular cover many common operations. And if there’s something you’re looking for but can’t find, please contact Divio support.

PHP/Laravel chapters

Create a new PHP/Laravel project

Our Flavours implementation is in beta
In this section we will create and deploy a new project in the Control Panel using Laravel, a popular PHP web application framework. The principles covered by the tutorial will apply to any other development stack.

**Note:** This tutorial assumes some basic familiarity with PHP. It is intended to introduce the basics of working with Divio, using Laravel as an example. **It is not a tutorial for Laravel.**

### Set up a project in the Cloud

If you have not already done so, you will need to set up the local development environment.

### Create the project

In the Divio Control Panel, create a new project.

Any web application in any language can run on Divio, as long as there is nothing that prevents its being Dockerised. However for convenience we also provide a number of ready-to-go project types for applications built in Java, PHP, Node and other languages (and the list is growing).

Select the following options for your project (other options should be left on their default settings):

- **Python:** PHP
- **Project type:** Laravel

Divio projects use Git for code management. We provide a Git server that your projects will use by default; you can also choose to use another Git service if you prefer. For this tutorial, use our Git server.

Hit Create. (You may be asked to select a subscription for the project; select the free Developer plan if so.)

It takes a few moments to create the project. During this process, the Control Panel defines the Docker image for your application by adding commits to its repository, assembling its Dockerfile and other files.
Project environments

Your project has two independent server environments, Test and Live. The Test and Live environments have their own services, and unique environment variables to configure access to them. They can be deployed independently, and can also be configured to track different Git branches.

Deploy the project

Once the project has been fully created, use the Deploy button to deploy the Test server.

The deployment process first builds the Docker image from the Dockerfile, and then launches a Docker container from the image. The container environment will include automatically-configured environment variables for connections to services such as the database, media storage and so on.

Typically, this takes a minute or so.

Open the Test environment website

Once you have successfully deployed the Test environment, the Control Panel will indicate this in the interface:

Last deployment successful at . . .

and the project URL in the Test environment pane will display as a link, for example https://tutorial-project-stage.us.aldryn.io. Select the link to access the project.

You’ll see the familiar Laravel starting page.
The new project doesn’t do anything very useful or interesting yet - but it’s up and running and ready to start working on.

**About deployment**

Any time new changes to the project code are committed to its repository, the Control Panel will indicate this with a message showing the number of undeployed commits for each of its server environments.

New code and configuration changes applied via the Control Panel (to subscriptions, cron jobs, environment variable, domains or other settings) will not take effect on either server environment until it is deployed once again.

If for whatever reason a deployment fails, there will be no down-time - the containers that are currently running will continue running, and the failing changes will not be deployed.

**Explore the Dashboard**

The Divio project Dashboard provides you with access to useful controls and information for the project. They are fairly self-explanatory and at this stage you don’t need to interact with any of them, but it’s worth familiarising yourself with what’s available.

**Set up your project locally**

In this section we will set up locally the cloud project you created earlier.

Obtain the project’s slug (its unique ID) from the Dashboard:
Alternatively you can use the `divio` command to list your cloud project, which will show their slugs:

```bash
divio project list
```

**Build the project locally**

Run the `divio project setup` command (for example if your project slug is `tutorial-project`):

```bash
divio project setup tutorial-project
```

The Divio CLI will execute a number of steps - this make take a few minutes, depending on how much needs to be downloaded and processed. The Divio CLI tool will build your project locally (see *The project deployment process* for a more detailed description of what’s happening here):

```mermaid
sequenceDiagram
    participant CLT
    participant Workspace
    participant ProjectRepository
    participant Docker
    participant Database
    participant Migrations
    CLT --> Workspace: Creating workspace
    CLT --> ProjectRepository: cloning project repository
    CLT --> Docker: downloading remote docker images
    CLT --> Docker: building local docker images
    CLT --> Database: creating new database container
    CLT --> Database: syncing and migrating database
    CLT --> Migrations: Your workspace is setup and ready to start.
```

`cd` into the newly-created project directory, where you will find your project code.

**Run the set-up script**

This step is required as part of the beta implementation of the PHP/Laravel project type, and will be refined in later releases.

You’ll find a script in `divio/setup.php` that sets up components in the project, and performs database migrations. Run it with:

```bash
docker-compose run web php /app/divio/setup.php
```

This takes a few minutes. Once complete, you can run your project.
Start the local project

Start the project by running `docker-compose up` in the terminal:

```
docker-compose up
tutorial-project_database_default_1 is up-to-date
Starting tutorial-project_web_1 ... done
Attaching to tutorial-project_database_default_1, tutorial-project_web_1
database_default_1 | 2020-07-14 16:38:39+00:00 [Note] [Entrypoint]: Entrypoint_
˓→script for MySQL Server 5.6.49-1debian9 started.
[...]
database_default_1 | 2020-07-14 16:44:42 1 [Note] Event Scheduler: Loaded 0 events
database_default_1 | Version: '5.6.49' socket: '/var/run/mysqld/mysqld.sock' port: '3306' MySQL Community Server (GPL)
web_1 | Laravel development server started: http://0.0.0.0:80
```

Open the project in your web browser by visiting **http://localhost:8000**.

(You may notice above that Laravel claims to be running on port 80, not port 8000. It is - but that’s only *inside* the container. The `docker-compose.yml` configuration file is responsible for *this* port-mapping.)

If you open a new terminal window and run:

```
docker ps
```

it will show you the Docker processes that are running - you will see something like (note that the details will differ):

```
docker ps
CONTAINER ID IMAGE COMMAND CREATED STATUS
˓→ PORTS NAME
```

The first container is an instance of the image that you built, just like the one in the cloud deployment. The second is the database, running in its own Docker container.

Once you have successfully logged into the local site, stop the project, using **CONTROL-C**.

Useful commands

So far, we have used the `divio`, `docker-compose` and `docker` commands. It’s good to have a basic familiarity with them and what they do. As you proceed through this tutorial, you may encounter the occasional issue. These commands will help you when this happens.

**Note:** See our local commands cheat sheet for many useful commands.

Using divio

The `divio` command is used mainly to manage your local project’s resources and to interact with our Control Panel. You have already used `divio project setup` and `divio project list`; you can also use it to do things like push and pull database and media content. Try:
See the Divio CLI reference for more.

Using docker

The docker command is mostly used to manage Docker processes, and Docker itself. Mostly, you’ll never need to use it, but it can be useful when you need to understand what Docker is doing on your machine, or for certain operations. You have already used docker ps. Try:

docker info

Using docker-compose

The docker-compose command is used mainly to control and interact with your local project. You will mostly use it to start the local project and open a shell in the local web container. You have already used docker-compose build and docker-compose up.

Just for example, try:

docker-compose run web composer install

This will run the composer install inside the container (in fact this is one of the command in the setup.php script you ran earlier).

You now know how to set up a project in the local environment, and launch it. The next step is to do some development work in the project, test it, and deploy it to the cloud.

Make changes and deploy them

Make a change

We’ll make a simple change to the project code. Find the file resources/views/welcome.blade.php, which is responsible for the Laravel welcome page, and adjust the line controlling the background colour, for example:

```html
<!-- Styles -->
<style>
    html, body {
        background-color: black;
        color: #636b6f;
    }
</style>
```

and reload the page to check that it has taken effect.

Deploy to the Cloud

To deploy your changes to the Test server, push your changes, and run a deployment command:

git add composer.json app.flavour .flavour

```sh
git commit -m "Added laravel-respsecache"
git push origin master
```
On the project Dashboard, you will see that your new commit is listed as *1 Undeployed commit*. You can deploy this using the Control Panel, or by running:

```
divio project deploy
```

When it has finished deploying, you should check the Test server to see that all is as expected. Once you’re satisfied that it works correctly, you can deploy the Live server too:

```
divio project deploy live
```

This is about a simple change as its possible to make and deploy, but it helps illustrate the workflow and the development/deployment cycle. In the next section we’ll work through some more sophisticated steps.

### Install an application using Flavours

There are various ways to add packages to PHP projects. You are probably used to using Composer, and adding dependencies to the `composer.json` file. You can try that now with a dependency you’re already familiar with.

First, add the dependency. Then, test that it is installed as expected by running:

```
docker-compose build
```

If you commit and push your changes once again as you did in the previous section, the project will be rebuilt with the new dependency when the cloud environment is redeployed.

### Introducing Flavours

However, this project is managed using Flavours, which gives us an additional way of working. Flavours is a platform-independent specification for building containerised web projects; the Flavours addon manager for PHP Laravel - `flavour/fam-php-laravel` - knows how to add a package to a Laravel project.

We will add the `laravel-responsecache` package. It’s an open-source addon, released by the Belgian agency Spatie, and can improve performance of Laravel sites by caching responses.

### Install the Flavours CLI

The Flavours CLI is an open-source package, published at GitHub.

Run:

```
npm install -g @flavour/cli
```

### Run the `flavour add` command

Run the `flavour add` command, as follows:

```
flavour add composer/spatie/laravel-responsecache:6.1.1
   Installing composer/spatie/laravel-responsecache:6.1.1
     ✓ Getting metadata
     ✓ Checking validity
     ✓ Adding requirement
     ✓ Installed composer/spatie/laravel-responsecache:6.1.1
```
laravel-responsecache is now installed in the project.

**About the command**

The command breaks down thus:

The Flavours CLI looks up the open Flavours registry at https://addons.flavours.dev, and finds the particular version there, and pulls down the YAML data it contains.

The CLI uses the information provided about the addon to identify the appropriate addon manager, which processes the YAML and performs the steps required by the *add* action.

**Check what the command has done to the project**

In this case, the addon manager will apply some changes to the project. You can see what they are by running `git diff`:

In its *app.flavour*, which includes Flavours description of the project, you’ll find in the *addons* section:

```yaml
'composer/spatie/laravel-responsecache:6.1.1':
  manager: 'flavour/fam-php-laravel:0.1.1'
  hash: 9c5f4b2311089d4c5b0def4a0ded5bd927dd8936d7db18da4cb84283e3413d1
```

*app.flavour* is in essence what makes a project Flavours-aware.

And on the project’s *composer.json* require section, the addon is listed as a component of the project, so that when the project is built, the addon will be installed:

```json
"spatie/laravel-responsecache": "6.1.1"
```

Finally, if it wasn’t there already, you will find a *flavour* directory, which contains information about the addon and some configuration for it.

Rebuild the project:

```
docker-compose build web
```

When you start the project again with `docker-compose up` it will now be running with laravel-responsecache installed and activated.

**Deploy to the Cloud**

**Push your code**

To deploy your changes to the Test server, push your changes, and run a deployment command:

```
git add composer.json app.flavour .flavour
git commit -m "Added laravel-responsecache"
git push origin master
divo project deploy test
```

Divio’s hosting service is Flavours-aware; your changes, once pushed and deployed on the Control Panel, will automatically use the *composer.json* file to rebuild the project with the new package installed.
Push the database

Using `divio project push/pull`

Your cloud database hasn’t yet been migrated, unlike the local database. One very useful function of the Divio CLI is the ability to push and pull your database and media storage to and from the cloud environments. Push the database with:

```
divo project push db
```

The local database will be pushed to the cloud Test environment; you’ll see it the records there after a few moments. Similarly, you can push/pull media files, and also specify which cloud environment. See the `local commands cheat-sheet`. A common use-case is to pull live content into the development environment, so that you can test new development with real data.

Explore configuration

As a Flavours-aware host, the Divio Control Panel recognises the newly installed package. In the project’s `Addons` view in the Dashboard, you will see it listed along with its version number and configuration options:

![Addons view in the Dashboard](image)

From its options menu, select `Configure`. You will be presented with a pane of default options (some of which you can edit). These defaults were contained in the addon’s YAML. If you hit `Save`, your options will be applied as environment variables (you can see them in the `Env Variables` view).

When next deployed, those variables will be applied.

Where to go next?

This completes the basic cycle of project creation, development and deployment; you should now be familiar with the fundamental concepts and tools involved.

Other sections of the documentation expand upon them. The `how-to guides` in particular cover many common operations. And if there’s something you’re looking for but can’t find, please contact Divio support.

4.2 How-to guides

4.2.1 Working in the local development environment
How to get started with the Divio CLI

Although we provide the Divio app, a GUI application for working with your projects locally, for many developers the preferred tool for working with Divio projects is the Divio CLI.

See also:

If you are completely new to Divio, please see our tutorial, which guides you through installation and use of the Divio CLI in more detail.

Pre-requisites

In order to use the Divio CLI, you will need to install various packages if you do not already have them installed, including:

- Docker
- Git
- Pip

Install the CLI

The Divio CLI is a Python application. Note that it requires Python 3.6 or higher. Install it with:

```bash
pip install divio-cli
```

Log in using the CLI

Run:

```bash
divio login
```

This will open https://control.divio.com/account/desktop-app/access-token/ in your browser, from where you can copy a token to paste into the terminal.

Add your public key to the Control Panel

Upload your public key at https://control.divio.com/account/ssh-keys/.

Usage

The CLI allows you to interact with projects locally and on the Cloud; for example, to set up a Cloud project locally:

```bash
divio project setup <project slug>
```

See the reference guide.
How to run a local project in live configuration

The Local, Test and Live server environments are as identical as possible, to help guarantee that your applications will run in just the same way across all of them.

However, there are a few differences. See Default project conditions for the default configuration in each environment. Occasionally, you may wish to run the local server in a configuration closer to the live set-up. A few steps are needed to achieve this. You may not need to take all of these steps - it depends which aspects of the environment matter in your particular case.

**Build the project**

Build the project in the normal way (docker-compose build web) if there have been any changes to it.

**Isolate the container’s file system from your own**

The default volumes configuration in the docker-compose.yml file overwrites directories inside the container with directories from your own file system:

<table>
<thead>
<tr>
<th>volumes:</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>- &quot;.:/app:rw&quot;</code> # overwrites <code>/app</code> with the entire project directory</td>
</tr>
<tr>
<td><code>- &quot;.:/data:/data:rw&quot;</code> # overwrites <code>/data</code> with the entire data directory</td>
</tr>
</tbody>
</table>

This is useful for development purposes, because it allows you to make changes on your filesystem such as code changes and have them immediately reflected inside the container. Additionally, it allows media files (which are not included in the image) that you pull to your local environment to be made available inside the container at /data.

However, it does overwrite any files in either of these locations that belong to the image, inside the container, each time a container is launched using docker-compose.

This does not occur in cloud deployments, so you will often want to prevent this when running a local project in live configuration.

For example, our Aldryn Django Dockerfile contains a collectstatic command that collects static files at /static_collected in the image. Most of the time, it doesn’t matter that these are overwritten in the local environment container, because by default it runs Django in DEBUG mode and serves the files from the installed packages. To run in the same conditions as on the cloud, these files need to be served from the /static_collected directory that was created by the image, so that needs to be available.

Comment out the line:

| - ".:/app:rw" |

in order to protect the image’s files from being overwritten in the container.

(Note that once you do this, you can no longer amend files inside the container by making changes on your files system.)

**Ensure the application runs in a production configuration**

Your application may run in a different configuration in different environments.

The STAGE environment variable in the file .env-local provides the application with information about the environment. Use:
There may be specific environment variables that also need to be checked. For example, Aldryn Django projects run in DEBUG mode locally, in which case you’d also need:

```
DEBUG=False
```

### Run the `migrate` command

Migration commands, also known as release commands, are run during deployment *after* the image build has completed successfully. They are not run automatically locally, but you can execute them with:

```
docker-compose run --rm web start migrate
```

(In an Aldryn Django project, you can see these commands listed in the `MIGRATION_COMMANDS` setting, populated by applications using the addons framework).

### Use the cloud media storage rather than local file storage

Your local project will use local file storage rather than the cloud storage. Cloud media files are pulled to the local environment when you run `divio project setup` (or later, `divio project pull media`). Usually this is most appropriate for development, and also faster and more convenient than using the remote cloud storage. However, sometimes you might want to use the cloud storage when the application is running locally.

Aldryn Django will use the `DEFAULT_STORAGE_DSN` environment variable to configure storage. This is provided in all cloud environments (each environment gets its own value). If the variable is not present, Aldryn Django will revert to using `FileSystemStorage` (Django’s default), which is what happens locally.

In order to use the cloud storage instead, find the value of `DEFAULT_STORAGE_DSN` using the `divio project env-vars` command, and add the variable to the `.env-local` file. The next time you start the container, it will use the cloud storage.

### Use the cloud database rather than a local instance

Your project uses our database cluster on the cloud. Locally, it sets up the same database in its own container.

The databases for our public regions are not accessible except from containers running on our own infrastructure, for security reasons. Access can be made possible for databases on private clusters only.

### Use the production web server

The `docker-compose.yml` launches your website, but doesn’t necessarily do it the way it would be launched on the cloud. For example, in Aldryn Django projects, it uses the Django `runserver` command, whereas the cloud environments use uWSGI.

To the production web server (using uWSGI, and serving static files) rather than the Django runserver, change:

```
command: python manage.py runserver 0.0.0.0:80
```

to:

```
```
command: start web

With other project types, you will need to amend the command suitably.
The local server will now be running in a configuration much closer to that of the live project.

4.2.2 Building your Docker application

How to manage a project’s base image

Divio provides a number of Docker base images for projects. These base images provide an underlying Linux operating system layer, and other layers in the stack, upon which you can build your custom application.

For example, our Python base images will include a particular version of Python, and any system-level components required to run it.

New base images will be released with updates (for example, for newer Python releases) and other improvements or changes. Generally, it’s recommended to use the newest version of the base image in a particular Release Channel.

A project’s Release Channel is indicated in its Dashboard, in Settings > Base Project. Note however that because the base image can also be specified in the repository via the Dockerfile, the manual setting will override what’s indicated in the Dashboard.

This document is concerned with using the Dockerfile to manage the base image.

Choose a base image

You don’t need to use a Divio-provided base image. However, it’s recommended.

Our base images are listed on Docker Hub.

Your base image should include the runtime environment(s) you need for your application, such as an appropriate version of Python.

Specify the base image in the Dockerfile

For example, to use our 0.4-py3.7-slim-stretch base image:

```
# <DOCKER_FROM>
FROM divio/base:0.4-py3.7-slim-stretch
# </DOCKER_FROM>
```

See our Dockerfile reference guide for more information on the # <DOCKER_FROM> section.

Test the build locally with docker-compose build before pushing the change to the cloud.

Caveats

Different base images may provide different packages and libraries. Our newer images tend to be slimmer, in order to make builds faster and to use fewer resources, but may lack some software that your application previously relied on.

For example, you may encounter a build error after adopting one of these images, such as:

```
ENOGIT git is not installed or not in the PATH
```
In other words, Git was available in the old image but not the new one, and is required to build the project. In such a case, the missing libraries should be **installed into the image manually**.

### How to install system packages in a project

See also:

- **How to install Python dependencies in a project**

All Divio projects are based on a customised Docker image, which uses a version of Ubuntu Linux. If your project requires a particular system package, you can include them in the Docker image, by listing the commands required to install them - typically, using `RUN apt-get` in the **Dockerfile**.

The commands in the **Dockerfile** are executed in order, so an appropriate place to put such commands is early on, after:

```bash
FROM aldryn/base-project:py3-3.23
```

and before any Python-related commands that might depend on the package.

You should include an `apt-get update` in the installation commands, and run `apt-get` with the `-y` ("Say yes") option, for example:

```bash
RUN apt-get update
RUN apt-get install -y wkhtmltopdf
```

To rebuild the docker image, installing the new packages:

```
docker-compose build web
```

The build output will show the new `RUN` instructions being executed as part of your build.

To make a quick check that the command installs what you require, without having to rebuild the entire project, jump into a new container running `bash` with:

```
docker-compose run --rm web bash
```

This container will disappear (`--rm`) when you exit.

### 4.2.3 Adding new functionality to a project

**How to configure Sass CSS compilation**

**Sass** is a popular CSS extension language, favoured by many frontend developers.

This document explains how to implement Sass compilation in a Divio project. Although this guide specifically deals with Sass, many of the principles it involves can be applied to other systems.

In this example we will install, set up and run:

- Node, the server-side JavaScript application framework
- `npm`, the Node Package Manager
- `gulp`, to build the Sass CSS
**Note:** Note that this document assumes you are working with a project that does *not* already have Node components set up and activated.

### What we want to do

If we were doing this by hand we might run:

```bash
# Activate Node Version Manager using nvm.sh (installed by default). The directory $NVM_DIR is set as an environment variable by the base project:
source $NVM_DIR/nvm.sh
nvm install 6.10.1
nvm alias default 6.10.1  # set a default Node version to be used in any new shell
nvm use default          # use the default Node version now
npm install -g npm@5.8.0  # ensure the correct version of NPM is installed
npm install -g gulp@3.9.1

# Install all required packages for building Sass (locally):
export NODE_PATH=$NVM_DIR/versions/node/v6.10.1/lib/node_modules  # set NODE_PATH as an environment variable
export PATH=$NVM_DIR/versions/node/v6.10.1/bin:$PATH         # Add the node directory to PATH
npm install gulp@3.9.1 autoprefixer@6.7.7 gulp-clean-css@3.0.4 gulp-postcss@6.4.0
    gulp-sass@3.1.0 gulp-sourcemaps@2.4.1 gulp-util@1.6.4
gulp watch    # Start watching the files specified in our gulfile.js to build the CSS
```

However, we can use Docker to automate this for us, and also use to build some more abstraction into the process, making it easier to maintain.

### Building this into the Dockerfile

See the *Divio Dockerfile reference* for more information on how our Dockerfile works.

### Set up the Node environment

Above, we specified some version numbers for the Node environment, and we can export them here as environment variables.

In your project’s Dockerfile, after the `# <DOCKER_FROM>[...]# </DOCKER_FROM>` section:

```
ENV NODE_VERSION=6.10.1 NPM_VERSION=5.8.0
```

Using environment variables like this for version numbers allows them to be specified just once, and re-used wherever required.

Other commands can be collected into an installation script file. This will use the NODE_VERSION and NPM_VERSION variables we set above:

```
#!/bin/bash

# Exit immediately in case of error
set -e

source $NVM_DIR/nvm.sh
nvm install $NODE_VERSION
```

(continues on next page)
nvm alias default $NODE_VERSION
nvm use default

npm install -g npm@"$NPM_VERSION"
npm install -g gulp@3.9.1

The file can be added to the project repository at `scripts/install.sh`.

Using a separate bash script for the installation commands allows us to maintain a cleaner `Dockerfile`, and manage the installation of frontend components separately from other concerns.

Back in the `Dockerfile`, we need to copy scripts directory to the container, and then execute the file:

```
ADD scripts scripts
RUN bash scripts/install.sh
```

and add the Node components to the appropriate paths:

```
ENV NODE_PATH=$NVM_DIR/versions/node/v$NODE_VERSION/lib/node_modules 
    PATH=$NVM_DIR/versions/node/v$NODE_VERSION/bin:$PATH
```

### Install other Node packages

Various other packages need to be installed locally: `gulp`, `autoprefixer`, `gulp-clean-css`, `gulp-postcss`, `gulp-sass`, `gulp-sourcemaps`, `gutil`.

These should be added to a `package.json` in the root of the project:

```
{
    "name": "package",
    "private": true,
    "dependencies": {
        "autoprefixer": "^6.7.7",
        "gulp": "^3.9.1",
        "gulp-clean-css": "^3.0.4",
        "gulp-postcss": "^6.4.0",
        "gulp-sass": "^3.3.0",
        "gulp-sourcemaps": "^2.4.1",
        "gutil": "^1.6.4"
    },
    "devDependencies": {}
}
```

In order to process these, you can add:

```
# <NPM>
# package.json is put into / so that mounting /app for local
# development does not require re-running npm install
ENV PATH=/node_modules/bin:$PATH
COPY package.json /
RUN (cd / && npm install --production && rm -rf /tmp/*)
# </NPM>
```

**Note:** It is strongly recommended to place these lines inside the `# <NPM> [...] # </NPM>` comments that exist

### 4.2. How-to guides
by default in every Divio Dockerfile. This is because the Divio Control Panel will *automatically* fill this section (if it exists) with appropriate commands when it discovers `package.json` in the project.

## Run compilation of CSS at deployment time

The final part of the task is to execute `gulp build` to compile the CSS.

Towards the end of the Dockerfile, inside the `# <GULP>[...]</GULP>` section, add:

```bash
# <GULP>
ENV GULP_MODE=production
RUN gulp build
# </GULP>
```

**Note:** The `# <GULP>[...]</GULP>` section exists in the Dockerfile by default. On deployment, the Divio Control Panel will *automatically* fill this section (if it exists) with appropriate commands when it discovers `gulpfile.js` in the project.

You will need an appropriate `gulpfile.js` at the root of the project too. It is beyond the scope of this document to describe how to create a `gulpfile`. For reference however, you may use the file provided in our own django CMS Boilerplate Sass. This file looks for Sass files in `private/sass` and compiles them to `/static/css`.

### Building the updated project

Run `docker-compose build web` (locally) to test the changes, or deploy them to the Test server.

In either case, the project will be started up as before, this time with compiled CSS files.

You can start the project locally with `divio project up` as usual. Running `docker-compose run --rm web gulp build` will start a watcher that executes compilation instantly whenever a Sass file in `private/sass` is changed.

### Further frontend development

This is just an example of a particular case. It’s possible to set up very extensive and sophisticated components and processes for your project’s frontend. Our django CMS Boilerplate Webpack is an example.

Though it’s beyond the scope of this documentation to describe how to do this in detail for every case, the basic principles are the same as in this example. If it’s possible to set up, it’s possible to automate the set-up of your project’s frontend components using Docker with consistent and reliable results.

### Using Boilerplates for quicker project creation

If you typically use the same particular frontend set-up for many sites, you should consider packaging it up as a *Boilerplate* that can be used at project creation time.

### How to configure Application Performance Monitoring

This example uses Elastic APM in a Django project. However, the principles are the same for other projects and services.
Set up the APM instance

Log in via the Elastic control panel. Create a new Elastic deployment. It makes sense to select the same cloud platform and region you’re using on Divio - select AWS US East unless you know otherwise, and the default I/O Optimized configuration.

Note the username and password provided for future reference. The APM instance created will also have a secret token; make a note of this too. Finally, you will need to copy the APM Server URL. (It takes a few moments to create the instance and for all this information to become available.)

Other services will allow you to create an instance and will provide the credentials for using it in much the same way.

Install and configure the Divio Telemetry addon

Divio Telemetry is available as an Aldryn addon via the Control Panel. It uses the official elastic-apm package provided by Elastic, and adds easy configuration via a single environment variable, DEFAULT_APM_DSN.

Note: At the time of writing, you will need to pin the elastic-apm package in your requirements.in:

```
elastic-apm<5.8.0
```

as the result of a known issue.

DEFAULT_APM_DSN is a URL, and can be assembled from the Elastic credentials you collected earlier. It’s in the form:

```
https://<Divio project slug>:<secret token>@<APM Server URL>:443
```

Your own URL should look something like:

```
https://apm-elastic-test:lAGtsyPGvg2T4o25Yr@d26758984b0c7792826e42918c785738.apm.us-east-1.aws.cloud.es.io:443
```

It should not include the protocol fragment (https://) of the APM Server URL.

Apply this as an environment variable, either using the Control Panel, or in the .env-local file. It makes sense to use a different APM instance for each environment.

Using a different provider, you will need to install the appropriate software in your project and provide it with the necessary credentials to access the third-party service.

Check the results

When the project has been rebuilt/deployed as required, you can visit the APM Server URL. Log in with the username and password. Your new instance will be displayed, along with the data it has collected.

How to configure media serving on a custom domain

By default, media files in Divio projects are served directly from our S3 cloud storage service, and the URL of each object will refer to that storage endpoint - for example, objects may be served from an S3 bucket which might have a domain like: example-test-68564d3f78d04cd2935f-8f20b19.aldryn-media.io.

Some users may prefer or require that their media are served from their own domain, say media.example.com.
This is not an option available by default, but it can be configured by setting up a reverse proxy.

**Set up the reverse proxy project**

Create a new Divio project, using the options:

- **Python**: No platform
- **Project type**: Nginx

This project will contain the reverse proxy, that will refer all requests made to the custom domain to the actual domain that the application serves its media from. The only function of this reverse proxy project is to refer requests to the actual application project.

**Set up a domain for media files**

Using the Control Panel, add the domain to the reverse proxy project.

**Configure Nginx as a reverse proxy**

We expect users who adopt this approach to be familiar with the management of reverse proxies and comfortable with their configuration. It’s beyond the scope of our support to provide details of how this should be done, however:

- set up the Nginx configuration to proxy requests from (say) `media.example.com` to (say) `example-test-68564d3f78d04cd2935f-8f20b19.aldryn-media.io`
- you *may* also need to configure header rewriting; some S3 hosts are stricter than others and may refuse to accept requests with the wrong heads

Deploy the project and check that requests for media objects on the custom domain are correctly referred to the actual storage, and that they are served as expected.

**Configure media storage URLs in the application project**

Your application has an automatically-configured `DEFAULT_STORAGE_DSN` environment variable. This contains the domain used by your code (e.g. Django Storages, for Django templates and views) when it needs to refer to the URL of a media object.

You will need to adapt this value, substituting the custom domain you wish to use, and apply it manually in the Environment Variables section of the project’s dashboard. Then the application project will need to be redeployed.

**Note:** As far as your application is concerned, it will still interact with the storage bucket, using the storage backend, at the original address. It is only when it needs to use a URL to refer to objects for access over HTTP that it will use the custom domain.

**Caveats**

When using our automatically-configured `DEFAULT_STORAGE_DSN`, you don’t need to be concerned about keeping this up-to-date - it’s managed for you.
When using your own custom domain, you will need to manage the configuration. For example, if your project is redeployed to a different region, the URL for its storage could change, and both the Nginx configuration and the `DEFAULT_STORAGE_DSN` will need to be changed too.

### 4.2.4 Platform-specific guides

**Python**

**How to install Python dependencies in a project**

See also:

- *Adding applications to a project* in our tutorial
- *How to install system packages*

To install dependencies in a project, you must first *List your dependencies*, then *Process the list*. Both steps are described below.

#### List your dependencies

Your Divio project has a `requirements.in` file, processed by the `pip-compile` command from `pip-tools` when the project is built.

Place your dependencies in the file, making sure that they are *outside* the:

```
# <INSTALLED_ADDONS>...
# </INSTALLED_ADDONS>
```

tags, since that part of the file is maintained automatically and is overwritten automatically with the requirements from the Addons system.

**Wherever possible, pin your dependencies**

An unpinned dependency is a hostage to fortune, and is highly likely to break something without warning when a new release is made.

Unpinned dependencies are the *number one cause of deployment failures*. Nothing in the codebase may have changed, but a fresh deployment can unexpectedly pick up a newly-released version of a package.

When *installing from a version control repository*, it is strongly recommended to pin the package by specifying a tag or commit, rather than branch.

Sometimes your dependencies may themselves have unpinned dependencies. In this case, it can be worth explicitly pinning those too - you can easily *pin all dependencies in a project* automatically.

#### Listing packages from PyPI

Use the package name, pinned with an optional (but *very strongly recommended*) version number, for example:

```markdown==2.6.8```
Listing packages from version control systems or as archives

You can use the URL of a tarballled or zipped archive of the package, typically provided by a version control system.

Important: More recent versions of pip tools as used in the Divio base projects require you to use URLs that provide the egg fragment (as shown in the examples below), and will raise an error if they encounter URLs lacking it. Older versions would allow you to omit the fragment.

Examples from GitHub

Master branch, as tarball:

```plaintext
https://github.com/account/repository/archive/master.tar.gz#egg=package-name
```

or as a zipped archive:

```plaintext
https://github.com/account/repository/archive/master.zip#egg=package-name
```

Specify a different branch:

```plaintext
https://github.com/account/repository/archive/develop.zip#egg=package-name
```

However, we very strongly recommend specifying either a tag:

```plaintext
https://github.com/account/repository/archive/1.6.0.zip#egg=package-name
```

or a commit:

```plaintext
https://github.com/account/repository/archive/2d8197e2ec4d01d714dc68810997aeeef65e81bcl.zip#egg=package-name
```

Note: Our pip set-up does not support VCS protocols - you cannot use for example URLs starting `git+` or `hg+`, such as `git+git@github.com:divio/django-cms.git`.

However, as long as the version control system host offers full package downloads, you can use the tarball or zip archive URL for that to install from the VCS, as in the examples above.

Process the list

The requirements file is processed when the project is build. This is taken care of in Cloud deployments by the `Dockerfile`, and locally by running a `build` command:

```plaintext
docker-compose build web
```

Make sure that you don’t also have a `requirements.txt` of pinned dependencies, otherwise you will simply be re-installing the old list.
How to pin all of your project’s Python dependencies

Pinning dependencies is good practice

If a dependency is *unpinned* (that is, a particular version is not specified in the project’s requirements) pip will install the latest version it finds, even if a different version was previously installed. This can cause your project to fail with an deployment error or worse, a runtime error, the next time it is built - *even if you didn’t change anything in it yourself*.

We strongly recommend that when you add a dependency to a project via its requirements.in that you pin it to a particular version, by specifying its version number.

For example, if you use rsa and know that version 1.3.4 works, specify it:

```
rsa==1.3.4
```

That way if a newer, incompatible version of the package is released, your project will still install the correct version the next time you build or redeploy it.

What about dependencies of dependencies?

However, dependencies can in turn themselves have dependencies. Even if you pin your requirements, their dependencies may be unpinned.

For example, your project may specify some-package==1.2.3, but if some-package lists rsa in its requirements, then the next time the project is built, it will attempt to do so using the latest version of rsa - which might not be compatible.

The solution is to pin all the dependencies in your project.

Compile requirements.txt

First, you need to have a working local set-up.

Now you can run:

```
docker-compose run --rm web pip-reqs compile
```

You won’t see any output, but you will now find a requirements.txt file in the project, containing a list of all the packages in the environment, along with their versions.

When your project is built using the new requirements.txt instead of requirements.in, you’ll have a guarantee that no unexpected changes will be permitted to find their way in to the project.

Amend the Dockerfile

In order to have your project built using requirements.txt instead of requirements.in, you need to remove the pip-reqs compile instruction from your project’s Dockerfile.

You can do this in two different ways:

- **Locally**: edit the Dockerfile to remove the pip-reqs compile instruction.
- **Let the Control Panel do it**: push your requirements.txt file to the project’s repository. At the next deployment, the Control Panel will recognise the file and amend the Dockerfile itself.
If you later remove the requirements.txt file, the Control Panel will recognise this and will restore the pip-reqs compile instruction to the Dockerfile when the project is next deployed. Alternatively you can restore it locally yourself.

**Important:** If the relevant sections in the Dockerfile are surrounded by the Divio-specific comment tags:

```
# <PYTHON>
...
# </PYTHON>
```

remove these tags - otherwise the Control Panel will simply overwrite your changes.

See the Dockerfile reference for more information about how the Control Panel populates the Dockerfile.

**Re-compile when required**

Any time a change is made to requirements.in, or to any addons in the project, you will need to re-compile requirements.txt. If the change is made to an addon via the Divio Control Panel, you will need to:

- pull the latest changes to your local repository
- run `docker-compose run --rm web pip-reqs compile` to compile the pinned requirements
- rebuild it locally to check that it works as expected
- push the updated requirements.txt back to the cloud.

**Django**

**How to add a new Django application to a project**

**Note:** This article assumes you are already familiar with the steps involved. For a full walk-through, see the Make changes and deploy them section of the developer tutorial.

The recommended way of installing Django applications is to use a Divio addon - an application that has already been packaged for easy installation in our projects.

If an addon has not yet been created for the application you require, you have two options:

- Add the application to the project manually (described in this article).
- Create an addon (described in How to package a Django application as an addon).

**Make the package available to the project**

You can do this in one of two ways:

- Copy the application to the root of the Python directory, so it’s on the Python path.
- Add it to requirements.in. See How to install Python dependencies in a project for details on how to do this.
Configure the project

Configure settings

Add the names of any required applications to the `INSTALLED_APPS.extend()` method in `settings.py`. Other key settings (such as `MIDDLEWARE_CLASSES`) will already be defined in settings, so don’t simply declare them (e.g. `MIDDLEWARE_CLASSES = [...].` If you do this, you will overwrite existing settings. Instead, use for example `MIDDLEWARE_CLASSES.extend([...]).`

Ordering of settings lists

The ordering of applications, middleware and other settings lists can matter, in which case you may need to make sure you add the item at the start, end or particular position in the list.

If for example your `DebugToolbarMiddleware` should be directly after the `GZipMiddleware`, you could do:

```
MIDDLEWARE_CLASSES.insert(
   MIDDLEWARE_CLASSES.index("django.middleware.gzip.GZipMiddleware") + 1,
   "debug_toolbar.middleware.DebugToolbarMiddleware"
)
```

Configure URLs

Edit the `urls.py` of the project in the usual way, to include the `urls.py` of your application, for example:

```
urlpatterns = [
   url(r'^polls/', include('polls.urls', namespace='polls')),
] + aldryn_addons.urls.patterns() + i18n_patterns(
   *aldryn_addons.urls.i18n_patterns() # MUST be the last entry!
)
```

Alternatively, add the URL configuration to be included via one of the `addon URL settings`, in your project’s `settings.py`.

Migrate the database

If the application has migrations, you should test them locally. Run:

```
docker-compose run web python manage.py migrate
```

Deploy the project

Push your changes

```
git add <changed or added files>
git commit -m "<message describing what you did>"
git push origin develop
```
Deploy the Test server

```
divio project deploy test
```

How to configure Django settings

In Django projects, settings are handled via the settings module (usually, the settings.py file).

Settings can be added to this file in the usual way, but in Divio projects, some settings need to be inspected and manipulated programmatically, to allow the addons system to handle configuration automatically. See How settings are handled in Django addons for more on how this works.

This can entail a little extra work when you need to change settings yourself, but the huge convenience it offers is more than worth the effort.

The correct way to manage settings such as INSTALLED_APPS is to manipulate the existing value, after having loaded the settings from the addons with aldryn_addons.settings.load(locals()). For example, in the default settings.py you will find:

```python
import aldryn_addons.settings
aldryn_addons.settings.load(locals())

INSTALLED_APPS.extend([  # add your project specific apps here
    ...
])
```

This allows you to add items to INSTALLED_APPS without overwriting existing items, by manipulating the list.

You will need to do the same for other configured settings, which will include:

- MIDDLEWARE (or the older MIDDLEWARE_CLASSES)
- TEMPLATES (or the older TEMPLATE_CONTEXT_PROCESSORS, TEMPLATE_DEBUG and other template settings)
- application-specific settings, for example that belong to django CMS or Wagtail. See each application’s aldryn_config.py file for the settings it will configure.

Inserting an item at a particular position

Sometimes it’s not enough just to add an application or class to a list. It may need to be added before another item. Say you need to add your application security just before cms. In this case you can target cms in the list like this:

```python
INSTALLED_APPS.insert(  
    INSTALLED_APPS.index("cms") + 0,  
    "security"
)
```

(+ 0 will insert the new item "security" immediately before "cms" in the list).

Of course you can use Python to manipulate the collections in any way you require.
Manipulating more complex settings

Note that in the case of more complex settings, like TEMPLATES, which is no longer a simple list, you can’t just extend them directly with new items, you’ll need to dive into them to target the right list in the right dictionary, for example:

```python
TEMPLATES[0]["OPTIONS"]['context_processors'].append('my_application.some_context_processor')
```

Listing applied settings

The Django `diffsettings` management command will show the differences between your settings and Django’s defaults, for example with:

```bash
docker-compose run web python manage.py diffsettings
```

In some projects (with addons that manipulate settings late in the start-up process), you may get an error: `RuntimeError: dictionary changed size during iteration.`

In this case you can run a script to print out your settings:

```python
from django.conf import settings
settings.configure()
django_settings = {}
for attr in dir(settings):
    value = getattr(settings, attr)
    django_settings[attr] = value
for key, value in django_settings.items():
    if not key.startswith('_'):
        print(f'{key} = {value}
```

How to migrate an existing Django project to Divio

Initial project setup

Create a new project in the Divio Control Panel. You’ll need to make sure that the project options are appropriate, including the Python version and project type.

There are a number of available project types, including Django, Django-plus-django CMS and Django-plus-Wagtail, that are already set up with the relevant addon packages.

Note: In general, if the software included in your project already exists on Divio as an Addon, it’s recommended to use the packaged addon version. This will help ensure not only that it is suitably configured for Divio, but that it will also co-exist well with other components.

Select the Boilerplate you want to use. Several are available, with different built-in frontend components to work with. If you choose a complex Boilerplate and later decide that you don’t need its functionality, it’s easy to remove from a
project. However, select the Blank Boilerplate if you are sure you’d rather to set up and manage your site’s frontend starting from scratch.

See Boilerplates for more on the subject.

Hit Create project.

**Important:** Do not start a deployment yet - we’ll cover that later.

---

**Check addon versions**

For each of the key components in your project for which a Divio addon exists, check that it is set to the correct version in your project, via the project’s Manage addons. This could include:

- Django
- django CMS (as well as key applications such as Django Filer, Aldryn News & Blog and so on)
- Wagtail

Note that the version you seek may exist in the Beta or Alpha release channels of the addon.

---

**Set up the project locally**

Once any addons have been appropriately configured, you’ll need to set the project up locally. (See the local setup section in the tutorial if this is new to you.)

Using the Divio CLI set up a local copy of the project:

```
divio project setup <your-project-slug>
```

---

**Migrate your existing code base**

**Add requirements**

Addons will install their dependencies, so there is no need to add those explicitly as requirements. Compare the output of:

```
docker-compose run --rm web pip list
```

with your existing project’s requirements, or the output of `pip list` in its environment, to see what requirements will need to be added manually. The missing dependencies will need to be added via the `requirements.in` file. See How to install Python dependencies in a project for more on adding Python packages to the project.

Your project may also have some other requirements; see How to install system packages in a project.

---

**Add application code**

If your project contains custom applications that are part of the project itself (i.e. they live in directories inside the project, and are not reusable applications or libraries installed via Pip), copy them into the project directory.
Add templates and static files

Your project’s templates similarly need to be copied to the new project’s templates directory, and static files to static.

Configure settings

The settings for your project and its applications need to be added to settings.py.

**Important:** Do not simply copy all your settings into the file. This will not work as expected.

Add them in the appropriate way, which will depend on how they are configured.

**INSTALLED_APPS**

It can be a tedious and error-prone process to get all the INSTALLED_APPS correct, without either missing or duplicating any. It will help to get a complete list, sorted alphabetically, and to run a diff on the list from each project.

Add the following to the end of the settings.py of both your source project and the new Divio project:

```python
for app in sorted(INSTALLED_APPS):
    print(app)
```

For the original project, run:

```bash
python manage.py shell
```

and for the Divio project run:

```bash
docker-compose run --rm web python manage.py shell
```

In each case, copy the list of applications into a file and save the file. Now run a diff on the two files:

```bash
diff original-installed-apps new-installed-apps
```

In the output you will see lines starting with:

- > - an application present in the Divio project, but not in the original
- < - an application listed in the original, but not in the Divio project

In the first case, no action is required. In the second case, you may see entries such as:

`< some_application`

and you will know that this application has not yet been added to your Divio project’s INSTALLED_APPS.

(Once done, don’t forget to remove the lines you added.)

4.2. How-to guides
Importing content

Database

Divio projects use Postgres databases by default, with other options available. It’s beyond the scope of this document to cover all possible eventualities of database importing.

**Note:** In the examples below `<container_name>` will usually be something like `<project_slug>_db_1` - but you can confirm this by running `docker ps`:

```bash
docker ps
CONTAINER ID IMAGE COMMAND CREATED STATUS          
→ PORTS NAMES
71fe7e930f60 postgres:9.4 "docker-entrypoint..." About an hour ago Up About an
→ hour 5432/tcp import_project_db_1
[...]
```

The **NAMES** column will list the container name.

**Example of Postgres-to-Postgres migration**

If you’re already using Postgres, you’re likely to find that steps along these lines will work:

Drop the database of the newly-created project:

```bash
docker exec <container_name> dropdb -U postgres db --if-exists
```

Create a new, empty database:

```bash
docker exec <container_name> createdb -U postgres db
```

Add the **hstore** extension:

```bash
docker exec <container_name> psql -U postgres --dbname=db -c "CREATE EXTENSION IF NOT
→ EXISTS
hsstore"
```

Finally, assuming that you have already dumped your existing database to a local file, import it:

```bash
docker exec -i <container_name> psql -U postgres --dbname db < /path/to/dump
```

**Migrating from one database to another**

If you need to convert your existing database, you can use a conversion script such as `https://github.com/lanyrd/mysql-postgresql-converter`.

Alternatively, you can export the data to a JSON file (via Django’s `manage.py dumpdata` command) and then load it back into the new database with `manage.py loaddata`.

You may find these resources useful:

- `https://github.com/lanyrd/mysql-postgresql-converter`
- `https://wiki.postgresql.org/wiki/Converting_from_other_Databases_to_PostgreSQL`
Once you have loaded your data, check that its migrations are in order, using the `python manage.py migrate`.

**Media files**

Media files should be copied to your project’s `data/media` directory.

**Test the local site**

You’re now in a position to test the local site, which should be done thoroughly. Start it up with:

```
divio project up
```

**Upload your changes back to the Divio**

Your project is a Git repository (certain files and directories are excluded), and should be pushed to Git server in the usual way (`git add/git commit/git push`).

Media files are not included in the Git repository (static files are however) and must be pushed:

```
divio project push media
```

And the database also needs to be pushed:

```
divio project push db
```

The project can now be deployed on the Test server:

```
divio project deploy
```

**Upload your project to an independent version control repository**

Optionally, you can maintain your project’s code in an independent version control repository. You can add another Git remote or even a Mercurial or other remote, and push it there.

**How to create a multi-site Django project using Mirrors**

Start with the original, the project from which the mirrors will be created. The mirrors will share the codebase, database and media storage of the original, but will run as wholly independent Docker instances.

**Create the mirrors**

Ensure that both Test and Live servers have been successfully deployed.

Create one or more mirrors.
Apply environment variables to each mirror

Database and media storage

The following environment variables from the Test and Live environments of the original will need to be applied to
the corresponding environments on each of the mirrors:

- DATABASE_URL
- DEFAULT_STORAGE_DSN

You can do this locally, using:

divio project env-vars -s test --all  # use live to collect the values from the live server

or you can SSH into the cloud servers and run the env command to list them.

Using the Control Panel, add the variables to the Test and Live environments of each mirror.

SITE_ID

The SITE_ID of the original will be 1 by default. For each mirror, add a SITE_ID environment variable, increment-
ing it each time.

If two sites share the same SITE_ID, or if you save an object in the Sites admin before deploying a mirror with its
SITE_ID in place, you may have unexpected results.

Deploy your mirrors

Mirrors can be deployed from their own Dashboards, or from the Mirrors view of the original (see our support article
Working with mirrors).

In the Django Sites admin you will see each mirror now listed.

If you’re using a Django application that makes use of the Sites framework, such as django CMS, you will see that it
now has access to multiple independent sites. In django CMS for example this means that pages can be one site of the
project or another.

Managing database migrations

Typically when working with mirrors, code changes will be applied to the original and then rolled out to each mirror
over a period of time - at any rate, not all mirrors will immediately be deployed along with the original. Because the
deployment of any one of the sites will run any outstanding migrations in the database they all share, this means that
you could be in a situation where the database and codebase are out of synchronisation for some of the sites.

Backwards-compatible migrations

One solution to this is to adopt a two-step strategy for migrations that could be affected.

For example, suppose that a code change removes a database field. In that case, when the original is deployed, it will
immediately change the database schema, and any code in the mirrors that expects to find the field will fail.

Instead, you would need to:
1. without changing the model field, remove code in views, model methods and so on that would attempt to use the field
2. roll out the change to all mirrors
3. remove the field
4. migrate the database

**How to configure an external logging service**

Your Test and Live servers have their own runtime logs, available from the project’s dashboard in the Control Panel. These logs are provided as a convenience. However they are limited to only the last 1000 lines of output and are not intended to be a comprehensive logging system for production purposes.

For that we recommend subscribing to a dedicated logging service, of which there are several, and configuring your project to route different kinds of logs (access, errors and so on) to different destinations, so you can use them more effectively.

**Example using LogDNA**

This document will show you how to set up logging using the popular LogDNA service. Using other services the principle will be the same, with only some minor differences.

If you don’t already have a LogDNA account, visit https://logdna.com and register for a free account. LogDNA will provide you with an **ingestion key**.

Set your project up locally. We’ll assume that you are using a standard Divio project using Aldryn Django.

**Install the logdna Python library**

You will need to add logdna to its requirements (strongly recommended: pin it to a particular version) and rebuild the project (docker-compose build).

This package provides a new logging handler (logdna.LogDNAHandler) that will forward log messages to LogDNA.

**Amend the LOGGING configuration**

In Aldryn Django’s aldryn.config.py, you will find the default logging configuration, that defines a LOGGING dictionary with several keys.

First, we’ll add the new logging **handler** to this.

```python
LOGGING["handlers"]['logdna'] = {
    'class': 'logdna.LogDNAHandler',
    'key': '<insert your ingestion key here>',
    'options': {
        'hostname': 'your-website-name',
        'index_meta': True
    }
}
```
What we have done here is added the new handler, logdna (the name doesn’t actually matter) as a key to the dictionary.

Next, we need to configure the existing loggers, that actually produce the logs, to use the handler. In this example, we will append the logdna handler to the configuration of:

- the unnamed root logger ""
- the django logger
- the django.request logger

```
LOGGING['loggers']['']['handlers'].append('logdna')
LOGGING['loggers']['django']['handlers'].append('logdna')
LOGGING['loggers']['django.request']['handlers'].append('logdna')
```

More information about configuring Django logging can be found in Django’s logging documentation.

**Other logging options**

The above is just a very basic example of using external logging. We recommend becoming familiar with Django’s logging framework and configuring it to send the most useful logs for your purposes.

**Sentry**

Sentry is another popular service; Aldryn Django is Sentry-aware and requires only the provision of a SENTRY_DSN environment variable to configure integration with Sentry.

**How to configure Celery**

**Note:** This article assumes that you are already familiar with the basics of using Celery with Django. If not, please see Celery’s documentation.

**Add Celery to your project**

In your project’s subscription, add the number of Celery workers you require. You can start with just one and add more later if required.

**Important:** If your Test and Live servers have not yet been deployed, please deploy each of them. This is required before Celery can be provisioned on the project.

Celery will then be provisioned on your project’s Test and Live servers by our infrastructure team. This includes the installation of our Aldryn Celery addon, and configuration of new environment variables your project will need.

Once provisioned and deployed, your cloud project will run with new Docker instances for the Celery workers. The containers running the Celery workers are built using the same image as the web container.

Note that a project’s Test server, or projects on the free Developer plan, will pause after 15 minutes’ inactivity in order to save resources. This will also pause the Celery workers.
About Aldryn Celery

Aldryn Celery is a wrapper application that installs and configures Celery in your project, exposing multiple Celery settings as environment variables for fine-tuning its configuration.

You don’t need to use Aldryn Celery to use Celery and Django Celery on Divio - you can of course install and configure Celery components manually if you prefer, perhaps if you wish to use a version that we haven’t provided support for in Aldryn Celery. You will in that case need to:

- install the Celery components you need in your project’s requirements file
- apply the settings we provide as environment variables.

Configure Celery for the local server

For development purposes you will need to set up Celery in the local environment too, in such a way that it reflects the provision made on our Cloud.

<table>
<thead>
<tr>
<th>function</th>
<th>handled by</th>
<th>on the cloud</th>
<th>local container</th>
</tr>
</thead>
<tbody>
<tr>
<td>AMPQ message broker service responsible for the creation of task queues</td>
<td>RabbitMQ</td>
<td>CloudAMPQ</td>
<td>rabbitmq</td>
</tr>
<tr>
<td>task execution</td>
<td>Celery workers</td>
<td>Celery containers</td>
<td>celeryworker</td>
</tr>
<tr>
<td>scheduling</td>
<td>Celery beat</td>
<td>Celery beat container</td>
<td>celerybeat</td>
</tr>
<tr>
<td>monitoring</td>
<td>Celery snapshots</td>
<td>Celery camera container</td>
<td>celerycam</td>
</tr>
</tbody>
</table>

Locally, the four new containers will be set up as new local services using the `docker-compose.yml` file.

Note that in the cloud environment, the Celery-related containers are launched automatically. They, and the AMPQ message queue, are not accessible. All monitoring and interaction must be handled via the main application running in the web containers. The `docker-compose file is not used on the cloud`.

Your project will already have at least two services, web and db, listed in `docker-compose.yml`. Each of the new services will be need to be added in a similar way.

RabbitMQ

Set up the RabbitMQ messaging service, by adding the following lines:

```yaml
services:
  web:
    [...]
  db:
    [...]
  rabbitmq:
    image: rabbitmq:3.5-management
    hostname: rabbitmq
    ports:
      - "15672:15672"
    expose:
```

(continues on next page)
This uses the official Docker RabbitMQ image (the `rabbitmq:3.5-management` image in turn installs `rabbitmq:3.5`). It also gives the container a hostname (`rabbitmq`), maps and exposes the management interface port (15672) and sets a `RABBITMQ_ERLANG_COOKIE` environment variable (the actual `secret_cookie_value` here doesn’t matter too much - you’re only using this locally).

**Celery worker**

Next add a Celery worker service in the same way. This service needs to run a Django environment almost identical to that used by the web service, as it will use the same codebase, need access to the same database and so on. Its definition will therefore be very similar, with key changes noted here:

```yaml
celeryworker:
  build: "."
  links:
    - "db:postgres"  # the actual value will depend on your project's database
    - "rabbitmq:rabbitmq"
  volumes:
    - ":/app:rw"
    - ":/data:rw"
  command: aldryn-celery worker
  env_file: .env-local
```

Rather than copying the example above, use the actual web service in your `docker-compose` file as its basis, in case it contains other values that need to be present. Note that the `ports` option is not used.

The `command` option starts the worker process, and `links` provides a reference to the `rabbitmq` service.

**Celery beat**

Celery beat needs to be set up in much the same way:

```yaml
celerybeat:
  build: "."
  links:
    - "db:postgres"  # the actual value will depend on your project's database
    - "rabbitmq:rabbitmq"
  volumes:
    - ":/app:rw"
    - ":/data:rw"
  command: aldryn-celery beat
  env_file: .env-local
```

**Celery cam**

And Celery cam:
celerycam:
  build: "."
  links:
    - "db:postgres" # the actual value will depend on your project's database
    - "rabbitmq:rabbitmq"
  volumes:
    - ":/app:rw"
    - ":/data:/data:rw"
  command: aldryn-celery cam
  env_file: .env-local

The web service

Finally, to the links option in web, you also need to add the link to rabbitmq:

web:
  [...]  
  links:
    [...] 
    - "rabbitmq:rabbitmq"

Set up local environment variables

In .env-local add:

```
RABBITMQ_ERLANG_COOKIE=secret_cookie_value
BROKER_URL="amqp://guest:guest@rabbitmq:5672/"
```

(Don’t confuse the port 5672 of the RabbitMQ server with the port 15672 of its management interface.)

Run the local project

Build the newly-configured project:

```
docker-compose build
```

Now docker-compose up or divio project up will start the services that Celery requires.

Note that although the Django runserver in your web container will restart automatically to load new code whenever you make changes, that will not apply to the other services.

These will need to be restarted manually, for example by stopping and restarting the local project or by running docker-compose restart. (Usually, only the celeryworker container needs to be restarted, so you can do docker-compose restart celeryworker.)

If you make any local changes to a project’s configuration that need to be accessible to the Celery workers, run docker-compose build to rebuild them.

Testing

It’s not within the scope of this documentation to explain how to get started with or use Celery, but as a quick check that you have configured your local environment correctly, you can create a small Celery task in your project, in a new
tasks_app application.

In the root of your project, add the application:

```python
tasks_app/
    __init__.py
    tasks.py
```

And in the tasks.py file:

```python
from celery.task import task
from aldryn_celery.celery import app

@app.task()
def add(x, y):
    return x + y
```

Note that we are using Aldryn Celery’s ready configured code here for convenience - otherwise, you would follow the steps as described in the First steps with Django from the Celery documentation.

And finally, add "tasks_app" to INSTALLED_APPS in settings.py.

Restart the celeryworker container, and start a new Django shell with:

```
docker-compose run --rm web python manage.py shell
```

Then in the shell:

```python
>>> from tasks_app.tasks import add
>>> result = add.delay(2, 3)
```

result is a Celery AsyncResult instance, so you can get the return value:

```python
>>> result.get(timeout=1)
5
```

If that works successfully, you have created a task, and been able to use RabbitMQ to send it to a waiting Celery worker.

See the Celery documentation for more information.

Environment variables

When Celery is enabled for your project, two new environment variables will be configured:

- `BROKER_URL`
- `RABBITMQ_ERLANG_COOKIE`

The Test and Live servers will have different values for both.

Other environment variables used by Aldryn Celery can be found in its `aldryn_config.py`.

If you change environment variables locally, the containers will need to be stopped and restarted in order to pick up the changes.

The number of Celery workers per Docker instance can be configured with the `CELERYD_CONCURRENCY` environment variable. The default is 2. This can be increased, but in that case, you will need to monitor your own RAM consumption via the Control Panel.
How to manage access authentication

In Django projects, access via password can be managed by the *Aldryn SSO addon*.

Require login

By default, the Test site is password protected while the Live site is not. This is controlled by the `ALDRYN_SSO_ALWAYS_REQUIRE_LOGIN` environment variable (False for Test, True for Live).

To override the behaviour, you can set the value explicitly in the *Environment variables* view in the Control Panel.

Basic access authentication

`.htaccess` is a familiar way of adding password protection to a web server at directory level.

Your Test server is always protected by our *SSO*, but you may occasionally require other forms of site-wide password protection.

It can be useful in the development process, for example, when you need to restrict access, or for a site that provides API endpoints that should require the client to authenticate.

A similar site-wide password requirement can be added to a Django site, using environment variables. Set them as follows:

```text
ALDRYN_SSO_ALWAYS_REQUIRE_LOGIN=basicauth
ALDRYN_SSO_BASICAUTH_USER=<username>
ALDRYN_SSO_BASICAUTH_PASSWORD=<password>
```

Those values can be set independently for test/live servers in the Environment Variables settings for each project.

See *Basic access authentication* for more.

How to manage redirects in Django projects

Protocol redirects

Divio projects are HTTPS-ready by default, and we provide free SSL certificates on all projects.

To force redirect from HTTP to HTTPS in Django, set the `SECURE_SSL_REDIRECT` environment variable to True.

Domain name redirects

You can set up your site’s domains using the *Domains* section of the Control Panel. This includes the ability to set a primary and secondary domains. The secondary domains can each be set to redirect to the primary domain if required.

The domains that are to be redirected to the primary domain can also be managed manually, via `DOMAIN_REDIRECTS`.

Language redirects

Django provides redirection to the the default language URL when none is specified. (In addition, django CMS offers complex fallback options for unavailable languages.)
For example, /about will redirect to /en/about if English is the default language.

`aldryn-django` can be configured to not use the prefix for the default language, by checking the `Remove URL language prefix for default language` checkbox on the settings for the Aldryn Django addon in the Control Panel.

**Permanent redirects**

`ALDRYN_SITES_REDIRECT_PERMANENT` will force permanent redirects for protocol and domain directs. This is to be used with caution.

**How to log in to a local Django project**

By default, Divio projects include the `Aldryn SSO` addon.

This allows you to log in to any of your projects automatically (whether locally, on the test environment or the live server) with your credentials provided to the Control Panel.

When you reach a log-in page locally, Aldryn SSO will offer you various options.

1. Create new Django super user.
2. Login with an existing user (if you’ve previously logged in on the Test server, your user will have been created automatically in the database).
3. Sign in with a Django user created for example with `manage.py createsuperuser`. 
Go-live checklist

Check your subscription plan

- In the Control Panel, check the project’s Subscription, and that it includes the technical and support resources it will require once the site is live.

Dependencies

- Consider pinning all dependencies. This will help ensure that future deployments do not introduce unexpected software updates.

DEBUG mode

- Ensure that your Live server is configured to run with DEBUG = False (this is the default, but may have been changed during development).
Domains

- If you are using existing domains, prepare them for the switch. Ensure that they have low (less than 60 seconds) TTLs. High TTLs can cause problems when the domains are pointed at the new site, including delays in the automatic provisioning of SSL certificates.

- Check that the live domain for the server is set up for the site in the Control Panel (support article: how to use your own domain with Divio).

- Check that any domains that should redirect to the primary domain are also set in the Domains setting in the Control Panel.

- If required, enable redirects to HTTPS by setting the `SECURE_SSL_REDIRECT` environment variable to `True`.

Environment variables

- Check that any other environment variables required on Live have been set (support article: How to use environment variables with your projects).

Serving configuration

- Check the Aldryn Django addon configuration. We recommend the `Hash static filenames` option, which lets you take advantage of caching.

Other settings

- Check your project’s `settings.py` for any settings that may have been temporarily configured during development.

Deployment

- Run a deployment of the Live server. If you have been using the Test server to build content prior to launch, use the `Copy data from Test and deploy` option.

After deployment

- Run a crawler on the live site to check for broken links, such as the W3C Link Checker or the open-source LinkChecker application.

- Check your site as a logged-in user, an anonymous user and in your browser’s private/incognito mode to verify expected behaviour.

- Check response times with a tool such Pingdom.

- If necessary, allocate more resources to the project via its Subscription and consult the How to fine-tune your server’s performance guide.
Node.js

How to force HTTPS with Express.js

In order to ensure secure communication with users of your Express.js applications, you can make all traffic to use HTTPS by forcing a re-direct from HTTP.

An Express-based application on is always accessed through a reverse proxy. When a user makes a request to your application, the reverse proxy interacts with your application on behalf of the client. For more on this topic, see the Express.js documentation.

We need to inform Express that it is behind a proxy. Edit your application’s server.js to add:

```javascript
app.enable('trust proxy')
```

This setting will populate the request object with extra information; we can then query the request to determine whether it was made over HTTP or HTTPS. Add the following prior to your other routing (you will need to modify this to suit your application):

```javascript
app.use(function(request, response, next) {
    if (process.env.NODE_ENV != 'development' && !request.secure) {
        return response.redirect("https://" + request.headers.host + request.url) }
    next()

})
```  
This will redirect all non-HTTPS requests.

Working locally

When working locally, you won’t want this behaviour enabled. The best way to control it is via an environment variable, that you read into server.js appropriately.

4.2.5 The development pipeline

How to configure external Git hosting

All Divio projects can use the Git private server we provide. This article describes how you can instead use the Git hosting provider of your choice.

**Important:** Once you have set up an external Git provider on a project, you will no longer be able to revert to Divio’s own Git server. Please ensure that this is what you want to do before using this feature.

The steps in this process are:

1. Prepare the external Git repository
2. Add the Git repository URL to the Control Panel
3. Add your project’s public key to the Git host
4. Test access
5. **Configure a webhook for the Git repository**

**Prepare the external Git repository**

Go to your Git hosting service. The next step depends on whether you are creating a new Divio project, or migrating an existing Divio project:

**Creating a new Divio project**

In order for our Control Panel to be able to check out the Git repository, it must be able to check out the `master` branch, with no conflicts.

1. Create a new repository at the Git provider.
2. Ensure the new repository has a `master` branch. The branch must not contain anything other than `.git`, `LICENSE`, `README`, `README.md` or `README.rst`.

If these conditions are not met, the Control Panel will not accept the repository URL.

**Migrating an existing Divio project**

1. Add the Git repository to the local version of your project as a remote: `git remote add external <repository URL>`.
2. Ensure that all the branches you wish to keep are present and up to date with the Divio server: `git pull <branch>`
3. Push the branches you require to the new remote: `git push external <branch>`

**Add the Git repository URL to the Control Panel**

You will need to supply the URL (SSH URLs are recommended, but you can also use HTTPS URLs) of your new repository to the Control Panel. The next step depends on whether this is a new or existing Divio project:

**Creating a new Divio project**

- Select *Repository > Custom* in the project creation page.

**Migrating an existing Divio project**

1. Select *Repository* from your project’s menu in the Dashboard.
2. Select *Migrate to external repository*.

**Add your project’s public key to the Git host**

The Divio Control Panel will provide you with a public key to add to the Git host, allowing our infrastructure to access the repository (*see below for HTTPS*).

Copy the key, and add it to the Git repository:
GitHub

1. In the repository, go to Settings > Deploy keys > Add deploy key.
2. Paste the key.
3. Select Allow write access.

GitLab

1. In the repository, go to Settings > Repository > Deploy keys > Create a new deploy key.
2. Paste the key.
3. Select Write access allowed.

BitBucket

1. Optionally, create a Bitbucket account specifically for Divio projects - otherwise the key will grant access to all your Bitbucket projects.
2. Go to Bitbucket settings > SSH keys > Add key.
3. Paste the key.

Test access

When you hit Continue in the Control Panel, it tests its access by performing a git pull action. If successful, the project Dashboard will show the repository URL.

Configure a webhook for the Git repository

In order for the Control Panel to receive a signal when the repository is updated, you need to set up a webhook. This step is optional but strongly recommended for convenience.

In the Repository view, select the appropriate webhook type (GitHub, GitLab and BitBucket each have their own type of webhook. Other providers will generally use a webhook that is similar to one of these).

The Control Panel will give you a URL to use for the webhook, and a secret key.

At the Git host, add a new webhook:

1. In the repository, go to Settings > Webhooks > Add webhook.
2. Add the Webhook URL to the Payload URL field.
3. Leave the Content type as application-x/www-form-urlencoded.
4. Add the Webhook Shared Secret to the Secret field.
5. Set Push events as the trigger for the webhook.
GitLab

1. In the repository, go to Settings > Integrations.
2. Add the Webhook URL to the URL field.
3. Add the Webhook Shared Secret to the Secret token field.
4. Leave the Push events trigger set.

BitBucket

1. In the repository, go to Settings > Webhooks.
2. Give the webhook a title.
3. Add the Webhook URL to the URL field.
4. For Triggers, ensure that Repository push is set.

The Webhook Shared Secret is not used.

Using the external Git remote

Your external Git remote has now been set up.

The Control Panel can save commits to it (using the key you provided) and the repository can send a signal to the Control Panel to pull in new commits when they land (using the webhook).

Options and special cases

Read-only access to the repository

In most cases, you will want to allow write access to the repository. This is the standard configuration and gives you full access to the benefits of the Divio addon system, in which the Control Panel records interface actions and addon configuration as Git commits.

It’s also possible to maintain stricter control over the repository, allowing only read access. In this case, project configuration that would normally be maintained via the Control Panel must be undertaken manually.

Write access is required to set up the external Git configuration, but may be disabled subsequently.

HTTPS authentication

By default, the Control Panel will assume that you will be using SSH authentication to the Git provider, which is preferred.

However, HTTPS can be useful in environments where SSH is not permitted, and is available if you explicitly provide an HTTPS URL.

You can give the Divio Project access to the Git repository over HTTPS by providing the Git hosting username and a personal access token.
This is disabled for those providers that would allow us to connect using your password. In accordance with our security policies, Divio will not request or store your passwords for other services.

Some Git providers enforce the use of personal access tokens for HTTPs, rather than allowing passwords to be used. However, GitHub, GitLab and BitBucket all permit HTTPS authentication using account passwords, and for this reason we do not permit HTTPS as an authentication method for these platforms.

Errors and what they mean

The remote repository requires a master branch

The Control Panel expected to find a branch (by default named master) at the remote.

If the Divio project uses the Custom tracking branches feature, then whatever branch is used for the Test environment should be present at the remote.

Check the repository for the expected branch.

Authentication error

The most likely problem is that one or more of:

- the URL of the Git repository entered into the Control Panel
- (for SSH) the Control Panel public key that you added to the deploy keys of the Git repository, and the deploy keys must have correct read/write access
- (for HTTPS) the Git repository username/personal access token that you added to the Control Panel are not correct. Check these values.

The master branch must exist and only include a single readme file in order to create a new project

The master branch at the remote repository contained other files.

Check that no other files are in the branch.

You have no webhook set up

Although the Control Panel has been able to connect to the repository and authenticate, a webhook has not yet been set up.

This is not necessarily an error, but it does mean that your Divio project will not automatically receive signals from the remote when new commits are made to it, and so you will need to use the manual Update button to pull new changes to your project.

Using webhooks is recommended.

How to use Git to with a Divio project

Your Divio project is a Git repository, offering several advantages to the developer - fine-grained revision control, excellent collaboration options, easy export and replication.

We provide Git hosting by default. Our server is git.divio.com. You can also use an external Git provider.
By default, we use the branch `develop`, but you can specify the Git branches to be used by each environment. You will need to set up your public key on our Control Panel if you use our Git server.

**Important limitations**

Certain conditions can cause deployment errors when the Control Panel tries to read the Git repository. These will typically appear in the deployment log with an exception from `pygit2`, such as:

```python
Traceback (most recent call last):
  [...]
File "/usr/local/lib/python3.6/site-packages/pygit2/repository.py", line 131, in __getitem__
    raise KeyError(key)
KeyError: abaddeed2d00ad47a9bb82db969707a21dead81ed
```

This can be caused by:
- an empty directory committed to the repository (remove it or add a file to it)
- a Git submodule included in the repository (remove it)
- files containing mixed line endings, if the `.gitattributes` configuration includes an instruction to resolve them (remove `* text=auto` if it appears in `.gitattributes`).

**Git with the Divio app or Divio shell**

If you use the Divio app or the *Divio Shell*, SSH keys to our Git server will be set up for you.

The *Divio app*, our desktop application for project management, also uses Git behind the scenes for its code *Upload* and *Download* operations.

**Basic Git operations**

It’s beyond the scope of this documentation to provide a guide to Git, but to get you started, the basic operations you will need are described here.

**Push your changes to the Cloud**

If you have made some local changes and want to push (i.e. upload) them to the Cloud, the basic steps you need are:
- `git status` to see the changed files
- `git add <file1> <file2>` (etc) to stage the changes (alternatively, you can do `git add .` to stage everything)
- `git status` to make sure everything has been staged
- `git commit -m "<your commit message>"` to commit the changes (provide a meaningful message for your own benefit)
- `git push origin develop` to push your local changes to the origin (i.e. our server)
Pull changes from the Cloud

- `git pull` will pull fetch and merge any changes that have been made on the Cloud

How to configure a CI/CD Deployment service

Continuous Integration/Continuous Delivery is a powerful addition to your development workflow. CI/CD automates deployment of commits that have passed tests and been merged into the deployment branch.

The basic principles

When a build completes successfully on the CI service, it needs to send a signal to the Control Panel to start the deployment. Typically this would be with the command:

```
divo project deploy --remote-id <website id>
```

where the `<website id>` is as shown in its Divio Control Panel dashboard URL.

The Divio CLI will need to be installed in the environment that executes this command, and to be authenticated with the Control Panel.

An example using Travis CI

In this example we will use Travis CI as an example, though Circle CI, Jenkins and other services will work just as well, and the principles are the same.

Connect your project to the CI service

For Travis, this means activating the project in your Travis account settings (https://travis-ci.org/account/repositories).

Your project needs a `.travis.yml` file, which depending on the project might look something like:

```
language: python

python:
  - 3.5

sudo: false

install:
  - pip install isort flake8

script:
  - flake8 --max-line-length=200
```

Triggering the deployment on success

When a build is successful, we need Travis to log in as a Divio user and execute some steps in the `after_success` section of the file:

- install the Divio CLI
• log in
• run a `divio project deploy` command, using your project’s website id

Authentication

To log in, Travis needs to provide your Divio authentication token.

Never include security keys in plain text

The token must **never** be included in plain text in the repository. Instead, you need to use an encrypted version. We recommend creating a Divio account specifically for using with Travis, so that you can more easily manage and revoke access.

You can encrypt your token using Travis's encrypt tool. Once you have obtained the encrypted token, you can add the following to the `travis.yml` file:

```yaml
after_success:
  - pip install divio-cli
  - divio login <encrypted token>
  - divio project deploy --remote-id <website id>
```

You could also run:

```
travis encrypt TOKEN="secretvalue" --add
```

which will add something like:

```yaml
env:
  global:
    secure: EsKcqn4H0EBqZhEts [...] X6klJCNI=
```

to the file, and make $TOKEN available as an environment variable. So you would then use:

```
- divio login $TOKEN
```

in the file.

Using the deployment API directly

The Divio CLI is the most elegant way of interacting with the Control Panel for deployment, but if desired, you can also use the API directly:

```
```

Caveats

• Multiple successive pull requests could lead to a race condition, depending on the order in which they arrive, in which a successful CI build triggers a deployment that then prevents the next build from deploying.

• If the CI job finishes very fast, it could trigger a deployment on the server before the Control Panel has had time to pull the changes to be deployed.
How to use our API

We provide a REST API using OpenAPI.

**Note:** Our Control Panel API is currently provided on an as-is basis. The API is scheduled for a full public release, but until then must be regarded as unstable and subject to change.

By design, the API will work only with projects that belong to an organisation. Projects under an user’s *Personal* space will not be fully accessible via the API.

Create a Divio account for use with the API

**Warning:** The API allows destructive operations to be carried out. Although you can, we strongly recommend that you *do not use your own account/access token with the API*, and especially not with code that is in development or not fully tested.

We recommend that you create a dedicated account for use with the API, and minimise the access it is granted. For example - where possible - rather than granting the account access to multiple organisations, give it access to just one, and do not give it admin access unless this is necessary.

Connecting to the API

The API’s end-point is https://api.divio.com/apps/v3/.

You will need a suitable client to connect to the API. This could be the `curl` command or another application.

Authentication

Each request made to the API must contain a valid Divio Control Panel access token. You can obtain an account’s access token from https://control.divio.com/account/desktop-app/access-token/.

**Warning:** The access token grants extensive access to your projects on our platform. Never share your access token with any other user. If you need to use it in an application or script, it is your responsibility to ensure that it is transmitted and stored safely.

The token must be included in a header `authorization`, in the form:

```
Token <include your token>
```

Basic requests

A example using `curl`:

```
curl https://api.divio.com/apps/v3/ -H 'authorization: Token yFagta25sbsus8d9JK9DrJCSKInqSWAoxU7NgN7IamtheCscry6jFfk3kingofthedivannTyYa10iqqD7EY5vPR6yN47
```

---

4.2. How-to guides 79
Here, the -H flag is used to pass the authorization header.

The principle using a dedicated REST API client, which will offer you more convenient ways to browse API, is much the same, whether it provides a graphical or command-line interface. It will behave in much the same way: you need to specify the URL, and provide the expected header.

The result of the command above will be a JSON response, something like:

```json
{
    "applications": "https://api.divio.com/apps/v3/applications/",
    "environments": "https://api.divio.com/apps/v3/environments/",
    "patches": "https://api.divio.com/apps/v3/patches/",
    "serviceinstances": "https://api.divio.com/apps/v3/serviceinstances/",
    "regions": "https://api.divio.com/apps/v3/regions/
}
```

### 4.2.6 Managing a project’s resources

#### How to interact with your project’s database

The database for your Divio project runs:

- in a Docker container for your *local* projects: *Interact with the local database*
- on a dedicated cluster for your *cloud-deployed* sites: *Interact with the Cloud database*

In either case, you will mostly only need to interact with the database using the tools provided by your project’s runtime stack (e.g. Django). However, if you need to interact with it directly, the option exists.

**Interact with the local database**

This is the recommended and most useful way to interact with the project’s database.

**From the project’s local Django web container**

**Using dbshell**

Run:

```
docker-compose run --rm web python ./manage.py dbshell
```

**Connecting to a Postgres database manually**

You can also make the connection manually from within the web container, for example:

```
docker-compose run --rm web psql -h postgres -U postgres db
```

As well as `psql` you can run commands such as `pg_dump` and `pg_restore`. This is useful for a number of *common operations*, below.
Using `docker exec`

Another way of interacting with the database is via the database container itself, using `docker exec`. This requires that the database container already be up and running.

For example, if your database container is called `example_db_1`:

```
docker exec -i example_db_1 psql -U postgres
```

**From your host environment**

If you have a preferred database management tool that runs on your own computer, you can also connect to the database from outside the application.

**Expose the database’s port**

In order to connect to the database from a tool running directly on your own machine, you will need to expose its port (5432 by default for Postgres).

Add a ports section to the `db` service in `docker-compose.yml` and map the port to your host. For Postgres, for example:

```
db:
  image: postgres:9.4
  ports:
  - 5432:5432
```

This means that external traffic reaching the container on port 5432 will be routed to port 5432 internally.

The ports are `<host port>:<container port>` - you can choose another host port if you are already using that port on your host.

Now restart the `db` container with: `docker-compose up -d db`

**Connect to a Postgres database**

You will need to use the following details:

- port: 5432
- username: postgres
- password: not required
- database: db

Access the database using your Postgres tool of choice. Note that you must specify the host address, `127.0.0.1`.

For example, if you’re using the `psql` command line tool, you can connect to the project database with:

```
psql -h 127.0.0.1 -U postgres db
```
Interact with the Cloud database

Note: It’s often more appropriate to pull down the Cloud database to a local project to interact with it there:

```
divio project pull db live  # or test
```

See the `divio project command reference` for more on using these commands.

From the project’s Cloud application container

Note: SSH access to an application container on the Cloud is available on Managed Cloud projects only.

Log into your Cloud project’s container (Test or Live) over SSH.

**Using dbshell in a Django project**

Run:

```
./manage.py dbshell
```

This will drop you into a command-line client, connected to your database.

**Connecting to a database manually**

You can also make the connection manually. Run `env` to list your environment variables. Amongst them you’ll find `DATABASE_URL`, which will be in the form:

```
schema://<user name>:<password>@<address>:<port>/<name>
```

You can use these credentials in the appropriate client, e.g. `psql`.

**From your own computer**

Access to cloud databases other than from the associated application containers is not possible - it is restricted, for security reasons, to containers running on our own infrastructure.

**Change the local database engine version**

Sometimes, you will need to change the database engine, or its version number, that your local project uses - for example if the cloud database is updated or changed. If the two database engines are not the same, you may run into problems.

The local database engine is specified by the `image` option in the `db` service in your project’s `docker-compose.yml` file, for example:

```
  db:
    image: postgres:9.6-alpine
```
Should you need to change this, that line should be updated - for example if the Cloud database is now running Postgres 11:

```
db:
  image: postgres:11-alpine
```

Docker will use the new version the next time the local project is launched.

If you are not sure what image to use for the local database, Divio support will be able to advise you.

**Important:** In the Divio architecture, the `docker-compose.yaml` file is **not** used for Cloud deployments, but **only** for the local server. The changes you make here will not affect the Cloud database.

---

**Usage examples for common basic operations**

It’s beyond the scope of this article to give general guidance on using the database, but these examples will help give you an idea of some typical operations that you might undertake while using Divio.

All the examples assume that you are interacting with the local database, running in its `db` container, and will use Postgres.

In each case, we launch the command from within the `web` container with `docker-compose run --rm web` and we specify:

- **host name:** `-h postgres`
- **user name:** `-U postgres`

### Dump the database

Dump the database `db` to a file named `database.dump`:

```
docker-compose run --rm web pg_dump -h postgres -U postgres db > database.dump
```

### Drop the database

Drop (delete) the database named `db`:

```
docker-compose run --rm web dropdb -h postgres -U postgres db
```

### Create the database

Create a database named `db`:

```
docker-compose run --rm web createdb -h postgres -U postgres db
```

### Apply the hstore extension

Apply the `hstore` extension (required on a newly-created local database) to the database named `db`:

```
```
docker-compose run --rm web psql -h postgres -U postgres db -c "CREATE EXTENSION hstore"

**Restore the database**

Restore a database named `db` from a file named `database.dump`:

```
docker-compose run --rm web pg_restore -h postgres -U postgres -d db database.dump --no-owner
```

**Reset the database**

To reset the database (with empty tables, but the schema in place) you would run the commands above to drop and create the database, create the hstore extension, followed by a migration:

```
docker-compose run --rm web python manage.py migrate
```

**Restore from a downloaded Cloud backup**

Untar the downloaded `backup.tar` file. It contains a `database.dump` file. Copy the file to your local project directory, then run the commands above to drop and create the database, create the hstore extension, and then restore from a file.

**How to interact with your project’s cloud media storage**

See also:

*Working with your project’s media storage in Python applications.*

Your cloud project’s media file storage is held on an S3 service - typically Amazon Web Services’s S3 service, or another S3 provider. Currently, most projects use Amazon’s own S3 service, or Exoscale for projects in our Swiss region.

Locally, your projects store their media in the `/data/media` directory, and you can interact with those directly. Then, you can use the Divio tools to push and pull media to the Cloud if required.

Occasionally you may need direct access to the S3 storage bucket for your project. You can manage this using a client of your choice that supports S3 and the particular storage provider.

**Interact with your project’s cloud S3 storage**

**Warning:** Note that S3 file operations tend to be destructive and do not necessarily have the same behaviours you may be used to from other models, such as FTP. It’s important that you know what you are doing and understand the consequences of any actions or commands.

**Obtain your storage access details**

Use the Divio CLI to obtain the `DEFAULT_STORAGE_DSN` for the environment, for example:
divio project env-vars -s test --all --get "DEFAULT_STORAGE_DSN"

See how to read environment variables.

This value contains the details you will need to use with a file transfer client for access to the storage bucket. The two examples below show which sections of the DSN correspond to the different parameters, for the hosts s3.amazonaws.com and sos.exo.io:

The key identifies you as a user.

The secret may contain some symbols encoded as hexadecimal values, and you will need to change them back before using them:

- \%2B must be changed to +
- \%2F must be changed to /

For any other values beginning with \% use a conversion table.

The bucket name identifies the resource you wish to work with.

The region is contained in the endpoint, the S3 host name. Sometimes it may be implicit, as in the case of Amazon’s default us-east-1:

<table>
<thead>
<tr>
<th>Provider</th>
<th>Endpoint</th>
<th>Region</th>
<th>Location</th>
</tr>
</thead>
<tbody>
<tr>
<td>Amazon</td>
<td>s3.amazonaws.com</td>
<td>us-east-1</td>
<td>US East (N. Virginia)</td>
</tr>
<tr>
<td></td>
<td>s3-eu-central-1.amazonaws.com</td>
<td>eu-central-1</td>
<td>EU (Frankfurt)</td>
</tr>
<tr>
<td></td>
<td>s3-eu-west-2.amazonaws.com</td>
<td>eu-west-2</td>
<td>EU (London)</td>
</tr>
<tr>
<td>Exoscale</td>
<td>sos-ch-dk-2.exo.io</td>
<td>ch-dk-2</td>
<td>Switzerland</td>
</tr>
</tbody>
</table>

See Amazon’s S3 regions table for more information about regions and their names.

The endpoint is the address that the client will need to connect to.

Save the parameters

Copy and paste each of these parameters into a text file, so you have them ready for use. Now that you have obtained the connection parameters, you can use them to connect with the client of your choice.

Choose a client

How-to guides are provided below for connecting to our storage using:

- AWS CLI, Amazon’s official S3 client
- s3cmd, an alternative command-line utility
- Transmit, a popular storage client for Macintosh
- CyberDuck, a popular storage client for Macintosh and Windows

Connect using AWS CLI

AWS CLI documentation is Amazon’s official S3 client. It’s a free, Python-based application.
Install and configure AWS CLI

Run:

```
pip install awscli
aws configure
```

You will be prompted for some of the *storage access parameters* values, extracted from the DSN, that you copied earlier.

- *AWS Access Key ID* - key
- *AWS Secret Access Key* - secret key
- *Default region name* - storage region
- *Default output format* - leave blank

Interact with your storage

Run `aws s3` followed by options, commands and parameters. For example, to list the contents of a bucket:

```
aws s3 ls example-test-68564d3f78d04cd2935f-8f20b19.aldryn-media.io
  PRE filer_public/
  PRE filer_public_thumbnails/
```

Or, to copy (`cp`) a file from your own computer to S3:

```
aws s3 cp example.png s3://example-test-68564d3f78d04cd2935f-8f20b19.aldryn-media.io/
  → example.png
upload: ./example.png to s3://example-test-68564d3f78d04cd2935f-8f20b19.aldryn-media.
  → io/example.png
```

Using AWS CLI with other providers

For non-AWS providers, such as Exoscale, you will need to add the `--url-endpoint` option to the command, as the AWS CLI assumes an endpoint on `.amazonaws.com/`. For the Exoscale example above, you would use:

```
aws s3 --endpoint-url=https://sos-ch-dk-2.exo.io [...]
```

Note that the scheme (typically `https://`) must be included.

Additional usage information

Run `aws s3 help` for more information on commands, or refer to the *AWS CLI Command Reference*. The AWS CLI can maintain multiple profiles and offers other features but it’s beyond the scope of this documentation to explain that here.

The `aws configure` command stores the configuration in `~/.aws`.

Connect using `s3cmd`

`s3cmd` is a free Python-based command line tool and client for uploading, retrieving and managing data in Amazon S3 and other cloud storage service providers that use the S3 protocol.
Install and configure s3cmd

Run:

```
pip install s3cmd
s3cmd --configure
```

You will be prompted for some of the storage access parameters values, extracted from the DSN, that you copied earlier:

- **Access Key** - enter the key from the DSN
- **Secret Key** - enter the secret key from the DSN
- **Default Region** - enter the storage region
- **S3 Endpoint** - enter the endpoint from the DSN

All other settings can be left untouched.

When you have entered the values, s3cmd will offer to test a connection with them (note that when using AWS, this will fail - ignore this).

Interact with your storage

Run `s3cmd` followed by options, commands and parameters. For example, to list the contents of a bucket:

```
s3cmd ls s3://example-test-68564d3f78d04cd2935f-8f20b19.aldryn-media.io
```

Note that the scheme (`s3://`) is required in front of the bucket name.

Additional usage information

Run `s3cmd` for more information on commands, or refer to Usage.

Using `s3cmd` you can take advantage of `--recursive` properties for iterating over the entire bucket contents; however it’s beyond the scope of this documentation to explain this or other features here.

`s3cmd --configure` creates a configuration file at `~/.s3cfg`.

Connect using Transmit

Install the Transmit file transfer application for Macintosh.

Create a new connection. You will need to enter some of the storage access parameters values, extracted from the DSN, that you copied earlier:

<table>
<thead>
<tr>
<th>Setting</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Protocol</td>
<td>Amazon S3</td>
</tr>
<tr>
<td>Address</td>
<td>endpoint</td>
</tr>
<tr>
<td>Access Key ID</td>
<td>key</td>
</tr>
<tr>
<td>Password</td>
<td>secret key</td>
</tr>
<tr>
<td>Remote Path</td>
<td>bucket name</td>
</tr>
</tbody>
</table>
Cyberduck

Install Cyberduck.

Create a new bookmark (note that you cannot simply use the Open Connection dialog, because this will not allow you to provide the required bucket name in order to proceed). You will be prompted for some of the storage access parameters values, extracted from the DSN, that you copied earlier:

<table>
<thead>
<tr>
<th>Setting</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Protocol</td>
<td>Amazon S3</td>
</tr>
<tr>
<td>Server</td>
<td>endpoint</td>
</tr>
<tr>
<td>Access Key ID</td>
<td>key</td>
</tr>
<tr>
<td>Path (in More Options)</td>
<td>bucket name</td>
</tr>
</tbody>
</table>

On attempting to connect, you will be prompted for the Secret Access Key; use the secret key.

For Exoscale (Divio Swiss region) deployments, you can also download and install the Exoscale profile for Cyberduck, which includes some prepared configuration.

Use the Divio CLI for local access to Cloud storage

The project’s media files can be found in the /data/media directory, and can be managed and manipulated in the normal way on your own computer.

Be aware that if you edit project files locally, your operating system may save some hidden files. When you push your media to the cloud, these hidden files will be pushed too. This will however not usually present a problem.

Pushing and pulling media files

The Divio CLI includes pull and push commands that target the test or live server as required.

**Warning:** Note that all push and pull operations completely replace all files at the destination, and do not perform any merges of assets. Locally, the /data/media directory will be deleted and replaced; on the cloud, the entire bucket will be replaced.

Limitations

You may encounter some file transfer size limitations when pushing and pulling media using the Divio CLI. Interacting directly with the S3 storage bucket is a way around this.

It can also be much faster, and allows selective changes to files in the system.

Storage ACLs (Access Control Lists)

When uploading files to your storage, note that you may need to specify explicitly the ACLs - in effect, the file permissions - on the files. If you don’t set the correct ACLs, you may find that attempts to retrieve them (for example in a web browser) give an “access denied” error.

On AWS S3, the public-read ACL needs to be set (by default it’s private). This is the ACL required for general use.
For example, you can use `--acl public-read` as a flag for operations such as `cp`, or `aws s3api put-object-acl` and `aws s3api get-object-acl` to set set and get ACLs on existing objects.

### How to manage your application’s environment variables

Your project relies on environment variables to configure application settings and access to storage, database and other services.

Environment variables can be set independently for each environment, whether on the cloud or locally.

Environment variables can be set:

- automatically by our infrastructure (particularly for key services, such as the database)
- by the user, either to add new variables or overwrite ones set by the system

See also:
- *Environment variables reference*

### Reading environment variables

#### From cloud environments

#### Using the Divio CLI

To read custom variables from the default cloud environment:

```
divio project env-vars
```

From another environment, using the `-s` option:

```
divio project env-vars -s live
```

Use the `--all` flag to include automatically applied variables, for example:

```
divio project env-vars -s live --all
```

<table>
<thead>
<tr>
<th>Key</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>ALT_DATABASE_DSN</td>
<td>mysql://example:<a href="mailto:Ro1T-d4Ldeaddeadqu6GtIt@appctl-black-mysql-00.cluster-cs4nfpsgul9fcn.us-east-1.rds.amazonaws.com">Ro1T-d4Ldeaddeadqu6GtIt@appctl-black-mysql-00.cluster-cs4nfpsgul9fcn.us-east-1.rds.amazonaws.com</a>:3306/example-live-b00bde685-deade65</td>
</tr>
<tr>
<td>CACHE_URL</td>
<td>db://django_dbcache</td>
</tr>
<tr>
<td>DEBUG</td>
<td>False</td>
</tr>
<tr>
<td>DEFAULT_DATABASE_DSN</td>
<td>postgres://example-live-beadle173ef383-3638:<a href="mailto:0Ut5vYvWjI3UQzczAGJ4aJ1XGWBtscsq_MobDltHiUMi12VHFbxyW_yKYA15-aw0F@appctl-black-sites-02.cs4nx9fcn.us-east-1.rds.amazonaws.com">0Ut5vYvWjI3UQzczAGJ4aJ1XGWBtscsq_MobDltHiUMi12VHFbxyW_yKYA15-aw0F@appctl-black-sites-02.cs4nx9fcn.us-east-1.rds.amazonaws.com</a>:5432/example-live-beadle173ef383638-ee6263</td>
</tr>
<tr>
<td>DEFAULT_STORAGE_DSN</td>
<td>s3://A5JHDYDTYZ3PLOF:t62LdAvROXgFR14</td>
</tr>
<tr>
<td>DOMAIN</td>
<td>example.us.aldryn.io</td>
</tr>
</tbody>
</table>
Using the Control Panel

Use the Env Variables view of a project to view (and add) custom variables.

In a terminal session to a cloud container

The env command will list all variables.

Setting environment variables

Setting a custom variable with the same name as one specified by the system will overwrite it.

Important: In all cases, changes to environment variables will not apply until the environment is relaunched (redeployed on the cloud, or restarted locally).

In cloud environments

Using the Divio CLI

Use divio project env-vars --set, for example to target the default environment:

```bash
divio project env-vars --set example_url https://www.example.com
```

or to specify an environment with the -s option:

```bash
divio project env-vars -s live --set example_url https://www.example.com
```

See the Divio CLI divio project env-vars reference for further information.

Using the Control Panel

Use the Env Variables view of a project to view and add custom variables. Variables need to be configured for each environment.

Leading and trailing spaces
The Control Panel does not strip leading or trailing spaces from values. Be careful when pasting in values that you do not inadvertently include unwanted spaces.

If you get an unexpected error in your logs that includes a reference to an environment variable value with a `%20` character in it - that’s a sure sign that it probably includes an undesired space.

---

**In the local environment**

By default, the `.env-local` file is used to store variables for the local environment (as specified by the `env_file: .env-local` in the `docker-compose.yml` file).

**In the build phase**

Use `ENV` in the `Dockerfile` to set an environment variable that will be used for the rest of the build process, and will also be baked into the image and accessible at runtime.

```
ENV <key>=<value>
```

You can also force a particular command to run with a certain environment variable:

```
RUN <key>=<value> <command>
```

However, the environment variables with which the cloud environments are provisioned (for example, for services such as database and media storage) are not accessible at build time (nor would it be desirable to rely on them in the build, since the same image will be used in multiple cloud environments).

---

**4.2.7 Working with addons**

**How to package a Django application as an addon**

**Register the addon**

Before your addon can be uploaded, the Divio Control Panel must be ready to receive it.

Select **Add custom addon** from **Personal Addons in the Divio Control Panel**, or simply go straight to **Add custom addon**.

- **Package Name**: must be unique on the system. We recommend prefixing it with your own name, for example `susan-example-application`.
- **Name**: e.g. Susan’s Django Debug Toolbar
- **License**: select a predefined license for your addon (or leave it blank and add your own later.)
- **Organisation**: select an organisation if appropriate.

When you hit **Create addon**, the addon will be registered on the system.

---

**Important**: The package name **must not** contain underscores.
Add the packaging files

We need to work in the project’s `addons-dev` directory. Create a new directory there with the same name as the `Package Name`.

Select `Package Information` from your addon’s menu. Download the packaging files, and add them to the addon. It should look something like this:

```
addons-dev/
    susan-example-application/
    addon.json
    LICENSE
    MANIFEST.in
    README.rst
    setup.py
    susan_example_application/
        __init__.py
```

Now let’s go through the files one by one.

**The `setup.py` file**

All the lines you need in the `setup.py` will be provided automatically in the downloaded version, with the exception of the `install_requires` argument:

**If your addon installs an application**

In this case, you will need to add the package to be installed to the `install_requires` argument, for example: `install_requires=['example_application==1.8.3']`.

**If your addon contains an application**

If on the other hand, for example if the application is not available on PyPI, simply add it as the inner application directory.

**Important:** The inner application directory, in this case `susan_example_application`, should have a name that matches the package name (`susan-example-application`), with underscores substituted for the dashes.

This will allow the Control Panel to copy the application’s templates into the project’s Git repository when the addon is first installed in a project. If the names don’t match, the project will still work, but the templates will not be made available for easy editing.

The addon will then contain some additional files:

```
addons-dev/
    susan-example-application/
        ...
        susan_example_application/
            __init__.py
            admin.py
            apps.py
            migrations/
```

(continues on next page)
Add any dependencies of the application to `install_requires` of `setup.py`.

### The `__init__.py` file

`setup.py` expects to find a version number in the addon, at `tutorial_django_debug_toolbar.__version__`:

For an addon that installs a package

We recommend providing a version number that tracks the package’s version number - for example, if the addon installs version 1.8.3, the addon’s `__version__` numbers should be 1.8.3.1, 1.8.3.2, and so on.

For an addon that includes a package

We recommend some form of semantic versioning.

### The other packaging files

The other packaging files are simpler:

- `README.rst`: If you haven’t already provided a description via the Control Panel, this will be empty. If you plan to share your addon with other users, it’s important to provide a useful README.

- `MANIFEST.in`: The default `MANIFEST.in` takes care of most non-Python files that an addon is likely to need the setup tools to take care of: `LICENSE`, plus directories for `LICENSE`, plus directories for `boilerplates`, `templates`, `static` and `locale` files.

- `LICENSE`: Make sure the license terms are appropriate.

- `addon.json`: We recommend leaving this as it is. Although you can use it to add multiple packages to `INSTALLED_APPS`, it’s better to do this in `aldryn_config.py` (see below).

### Add configuration

Create `aldryn_config.py`

If your application requires any settings of its own, you will need to manage them in `aldryn_config.py`, placed at the root of your application. The general form is:

```python
from aldryn_client import forms

class Form(forms.BaseForm):
    def to_settings(self, data, settings):
        settings['INSTALLED_APPS'].extend([SOME_APPLICATION])
```

(continues on next page)
settings['ENABLE_FLIDGETS'] = True
return settings

See how to configure settings in aldryn_config.py for more details and examples.

Provide form-based configuration

You can use the Form class to allow configuration via the Control Panel.
See adding form fields for user configuration for more information.

Provide URL configuration

Not all addons will have their own URL configurations that need to be included in a project, but if they do, you can add them. See how to include an addon’s URL configuration for more details.

Check the addon

Test it

Your addon is now ready to be tested.
Add the package name to the INSTALLED_ADDONS in settings.py. This adds it to the list of addons that the project will “watch”.
Run:

divio project develop <package name>

You can test that the project now works as expected.

Validate it

You will need the python package aldryn-client installed to be able to validate your addon.
If you have not already installed it, run:

pip install aldryn-client

Now make sure you’re in the addons-dev/<package name> directory.
Now, running divio addon validate should now confirm that the addon is valid:

divio addon validate
Addon is valid!
Upload the addon

Upload with `divio addon upload`.

This version of the addon will be placed into the Alpha release channel. If you visit the its Versions page, you’ll be able to change the release channel.

Your addon is now available for installation into projects via the control panel. If you make it public, other users will be able to install it too.

You can continue uploading new versions of it, as long as each has its own unique version number.

How to update an existing addon

Addons will need to be updated now and then. The basic process for updating an addon is to:

- *set it up in a project as if you were creating one*, in the `addons-dev` directory
- make and test your changes
- upload the new version.

Choose a local project to work with

Ideally, select a project that already works with an existing version of the addon. This way, you can check that the new version continues to work as expected, and that migrations for example run correctly.

If you don’t have such a project, the next best thing is to create a new project on the Control Panel containing the addon, and then set that up locally.

Uninstall the addon locally if necessary

You will find the addon listed in `requirements.in` - remove it from there, so that when you build the local container it will no longer try to install the old version.

You will also find it listed in `INSTALLED_ADDONS` in `settings.py` - *leave it there*.

Clone the addon repository to `addons-dev`

In `addons-dev`, clone the addon from its VCS repository. It should look something like this:

```
addons-dev/
   susan-example-application/
      addon.json
      LICENSE
      MANIFEST.in
      README.rst
      setup.py
      susan_example_application/
         __init__.py
```

Check that you have the appropriate version cloned.

Placing the addon into `addons-dev` will override any version that has been installed into the project using the requirements file.
Run `divio project develop`

If the new version is different from the previously installed version and includes changed dependencies, or you want to check exactly what it will do when the project is built, you should run:

```
divio project develop <package name>
```

This processes the addon, adding:

```
-e /app/addons-dev/<package name>
```

to the `requirements.in`.

You will need to run this again if you make any changes in the addon that involve dependencies or installation of components.

Work on the code

Restart the runserver (`docker-compose up`), and check that the addon continues to work as expected (or fails to work, if that is what previously happened).

Make and test any changes you want to make.

Push your changes

When you’re satisfied, you’re ready to update the addon.

Don’t forget to bump the version number in the `__init__.py`.

When you have finished all your updates, commit and push your changes to the addon’s repository (or make a pull request if it’s not your own).

Remember that if your addon is a wrapper for installing a reusable application, its version number should track the version number of the application, in an additional dotted increment - for example, the addon version `1.5.4.2` tracking application version `1.5.4` should become `1.5.4.3`.

Upload the new addon version

Finally, in the addon directory, run:

```
divio addon validate
```

to check it, and:

```
divio addon upload
```

to push it to the addons system.

Test it on the Control Panel

For completeness, check that the new version of your addon can be installed and deployed in a project.
Place the new version in the appropriate channel

By default, your newly-uploaded addon version will be placed in the Alpha channel. In the Addons section of the Control Panel, put it in the Beta or Stable channels if appropriate.

4.2.8 uWSGI Configuring uWSGI

How to manage uWSGI configuration

In our Django projects, the uWSGI gateway to the load balancers is part of the customer application: uWSGI is already configured and optimised in these projects. Most of this configuration is managed by Aldryn Django; see also How to fine-tune your server’s performance.

All of uWSGI’s configuration can be managed entirely within the project, according to your own requirements, but be warned that misconfiguration can be severely detrimental to an application’s performance.

uWSGI environment variables

uWSGI offers a vast number of configuration variables. Any one of them can be set using an environment variable starting UWSGI_, followed by the name of the variable in uppercase.

For example, the processes variable can be configured by setting UWSGI_PROCESSES.

Generally you will not need to touch these variables, and we recommend leaving them alone unless you need to change something and you know what you are doing.

uWSGI buffer size

One setting that occasionally needs to be adjusted is UWSGI_BUFFER_SIZE. The default value is 4096. If your site has to deal with very large request headers, you may receive a web invalid request block size error in your project’s logs.

In this case, you can increase the buffer size to allow larger request headers. (You may also want to find out why your site is running into such large request headers - for example, its cookies may be excessively large.)

More complex configuration

The uWSGI gateway can handle requests before they reach your application’s code, which is faster and less expensive than doing it in, say, Django (though not as convenient).

The configuration of this behaviour is often too complex to be expressed in simple environment variables; instead, it can be achieved by including a uWSGI configuration file.

The uWSGI configuration file needs to be specified using uWSGI’s include option, i.e. with the UWSGI_INCLUDE environment variable. For example:

```
UWSGI_INCLUDE=uwsgi.ini
```

Then the file can contain the additional configuration required.

It is beyond the scope of this documentation to discuss this in detail, but a typical use-case would be to perform a redirect. In such a case you could add a rule to the uwsgi.ini file specified above:
How to fine-tune your server’s performance

In the Divio Control Panel for your project, you can easily change the resources (instances, RAM, storage, transfer) allocated to your project to adjust for its needs.

In most cases, having chosen suitable values for these, you won’t need to make further changes.

If however your project has unusual demands or sees unusual traffic, you can also fine-tune some settings to match its needs even better.

Optimising the settings for a high-traffic site isn’t just a case of allocating more resources indiscriminately on the basis that “more is better”.

For example, a site that sees large numbers of requests that each represent a low load will benefit from being allowed to serve more concurrent requests, but that could make things worse for a site that sees a smaller number of requests each representing a heavier load.

Monitoring and profiling

Our Control Panel’s metrics will give you a basic idea of RAM usage. However, there are more sophisticated tools available that can give you much more, and more finely-grained, information.

These range from Python libraries (such as memory-profiler or line_profiler) to third-party monitoring/profiling services.

Aldryn Django settings

The Aldryn Django addon is responsible for ensuring that uWSGI (the application gateway server) starts up with appropriate parameters, for example for static file caching control.

These parameters can be controlled by environment variables.

Note: Some of these environment variables represent uWSGI run-time options. It’s also possible to configure other uWSGI settings, but we don’t recommend doing so unless you are familiar with them already and understand their implications.

Increase DJANGO_WEB_WORKERS

By default, each instance can run three concurrent uWSGI web workers, in other words, three concurrent web processes (note that your plan may include multiple instances). This is determined by DJANGO_WEB_WORKERS.

Increasing the number of web workers will allow you to serve more concurrent requests. Note that this will increase the RAM consumption of each instance, so make sure you monitor the results over time (briefly exceeding your hosting plan’s RAM limits might not matter, but if this is sustained, you will need to upgrade to a higher plan).

You can calculate as follows: requests per second * average time to serve a request (in seconds) will give you a rough idea how many concurrent web workers your site needs.
Three uWSGI web workers is quite a conservative number. You are likely to find that you can safely double the number of web workers. Increase them in small increments, and back off if you find that RAM consumption becomes excessive.

You are unlikely to see any benefit from lowering DJANGO_WEB_WORKERS below the default.

**Increase or decrease DJANGO_WEB_MAX_REQUESTS**

If you see more RAM use than expected, try lowering DJANGO_WEB_MAX_REQUESTS. This controls the number of requests each worker serves before it is recycled (replaced by a fresh one).

There is a small overhead in replacing the worker. Typically - unless you are seeing excessive RAM usage - increasing this number can improve performance. A value of 3000 is a reasonable starting-point for experimenting with different values.

*Lowering* this number can help when requests are memory-hungry, because recycling the worker releases the RAM.

**Decrease DJANGO_WEB_TIMEOUT**

The default allowable lifetime of a web process is 120 seconds. After this period, the worker handling it will be recycled.

You can lower this, if you know that no request served by your site should expect to take longer under normal circumstances. It means that the process that is slow or held up will be dropped, but will allow the site to try again, or serve other requests.

Taking this as low as 10 seconds may have benefits with no adverse effects. If your site occasionally needs to serve views that entail long processes (for example, applying a filter on a huge admin list) then you will need to adjust it upwards appropriately.

**Disable or reconfigure UWSGI_CHEAPER (uWSGI cheaper mode)**

By default, projects use uWSGI’s cheaper mode.

When the site is idling in cheaper mode, uWSGI will dismiss unneeded web workers. This saves RAM, and is the recommended configuration for most projects. In some circumstances however it can be advantageous to disable this mode, or adjust its settings.

This is generally only applicable to constant high-traffic sites. Please contact Divio support if you feel you need to disable cheaper mode or modify its settings, as misconfiguration can have adverse results.

### 4.2.9 Troubleshooting

**How to debug Cloud deployment problems**

Start with the *debugging checklist*. Work through the checklist by selecting the most appropriate answer for each question until you arrive at a probable fault for the symptoms you’re seeing.

There is also a *complete decision tree* for the debugging process.
Debugging checklist

Deployment on the Cloud has not worked as expected

Does the Control Panel show a “Last deployment failed” error message?

- The error message is shown
- The error message is not shown

The Control Panel shows a Last deployment failed message

Open the log. The relevant section will be towards the end, so work backwards from the end. Any error will be clearly stated.

What does the deployment log contain?

- The log appears to be empty
- The log appears to contain no errors
- The log refers to an error

Restart the checklist

Probable fault: temporary problem

Please try again. This is a rare and usually temporary problem. You may need to wait a few minutes for the condition to clear. If the issue is urgent, or you have already tried again, please contact Divio Support.

Restart the checklist

The deployment log contains no obvious error

Check the site’s runtime logs (via the Logs menu).

Do you see any obvious errors in the runtime logs for the web container (of the appropriate server, Test or Live)?

- The runtime log contains errors
- The runtime log contains no obvious error

Restart the checklist

Probable fault: application is too slow to start and times out

Probably your application took so long to start up that it triggered a timeout condition. On our platform, if a site is not up and running within a certain period after its build has completed, then the deployment is marked as failed.

This could happen because it is waiting for another external resource to become available, or the processing it needs to do at start-up is excessive. These issues generally represent a programming problem that needs to be resolved.

Build the site locally and start up the application to investigate why it is taking so long.

If the start-up processes can’t be made faster or more lightweight, investigate an asynchronous processing option such as How to configure Celery to allow them to go on in the background while the project starts up.
Restart the checklist

Probable fault: programming error in runtime code

Probably the issue is a programming error in the site that takes down Django as it launches (typically, this will be an ImportError). The runtime log will reveal the error.

Restart the checklist

The deployment log contains an error

The end of the log will contain the key error.

What does the error most closely resemble?

- Could not find a version that matches [...]  
- npm ERR! [...] ERR! /npm-debug.log  
- ImportError  
- ReadTimeoutError  
- The error does not seem to be any of the above

Restart the checklist

Probable fault: dependency conflict

An error that starts:

```bash
Could not find a version that matches [...]  
```

indicates that two or more of the components in your system have specified incompatible Python dependencies.

See How to identify and resolve a dependency conflict.

Restart the checklist

Probable fault: A Node error has halted the build

Example:

```bash
npm ERR! There is likely additional logging output above.  
[nm]91m  
[nm]91m npm ERR! [91m Please include the following file with any support request:  
[nm]91m npm ERR! /npm-debug.log  
```

In this case one of the Node component installation processes has failed. If the error is not clear from the log, contact Divio support for advice.

Restart the checklist

4.2. How-to guides
Probable fault: An import error halts one of the site build routines

Example:

```python
Step 8/8 : RUN DJANGO_MODE=build python manage.py collectstatic --noinput
[...] ImportError: No module named django_select2
```

In this case a Python application launched by an instruction in the Dockerfile has caused Django to halt with an error while it was trying to run the `collectstatic` command. This is a programming error. The traceback will show where it occurred.

*Restart the checklist*

Probable fault: temporary timeout error (read timeout)

Example:

```python
ReadTimeoutError: [...] Read timed out.
```

This may occasionally occur when our deployment infrastructure is under heavy load. In most cases you can simply try again. If the issue is urgent, or you have already tried again, please contact Divio Support.

*Restart the checklist*

Probable fault: A runtime error

If you are not sure what the error message reveals, please contact Divio support for assistance.

*Restart the checklist*

Probable fault: programming error at runtime

Sometimes there is no failed deployment log, but the site fails to start. This is typically caused by a programming error that becomes apparent at runtime.

Usually, the browser will show a Django traceback, if the site is in `DEBUG` mode (this is the default for the `Test` server). Under some circumstances, it might not, but the error will be shown in the site’s runtime logs, available from the `Logs` menu in the Control Panel.

*Restart the checklist*

Decision tree

This tree represents the logic of the debugging checklist.

- **Deployment on the Cloud has not worked as expected:**
  - The Control Panel shows a Last deployment failed message
    - The deployment log appears to be empty: Probable fault: temporary problem
    - The deployment log contains no obvious error
    - Runtime log contains no errors: Probable fault: application is too slow to start and times out
Runtime log contains errors: *Probable fault: programming error in runtime code*

- The deployment log contains an error
  - Could not find a version that matches [...] *Probable fault: dependency conflict*
  - npm ERR! [...]. ERR! /npm-debug.log: *Probable fault: A Node error has halted the build*
  - ImportError: *Probable fault: An import error halts one of the site build routines*
  - ReadTimeoutError: *Probable fault: temporary timeout error (read timeout)*
  - An error not listed above: *Probable fault: A runtime error*

- The Control Panel does not show a *Last deployment failed* message: *Probable fault: programming error at runtime*

### How to identify and resolve a dependency conflict

Occasionally when running a deployment on the Cloud or building a project locally, the process will fail, with a message like:

```shell
ERROR: Service 'web' failed to build: The command '/bin/sh -c pip-reqs compile && pip-reqs resolve && pip install --no-index --no-deps --requirement requirements.urls' returned a non-zero code: 1
```

This tells us that pip ran into problems while processing the project’s requirements. Resolving this requires a little detective work. The good news is that the information you require is provided, and the process for working through it to find the answer is set out below.

### Identify the conflict

Looking a little further up the log, we’ll see something like (this is just a representative example):

```shell
found candidate dj-redis-url==0.1.4 (constraint was <any>)
found candidate dj-static==0.0.6 (constraint was <any>)
Could not find a version that matches Django<1.11,<2.0,<2.1,<2.2,==1.8.18,>=1.11, >=1.6,>=1.8
Tried: 1.1.3, 1.1.4, 1.2, 1.2.1, 1.2.2 [etc]
```

The highlighted line tells us what the problem is: pip could not find a version of Django that matched all the listed constraints - naturally, because it’s impossible to have a version of Django that equals 1.8.18, is less than 1.11, and is also greater than or equal to 1.11.

So, between them, the packages being installed in the project have some mutually incompatible requirements. This can often be caused by *unpinned dependencies*, when a package is listed as a requirement without specifying a version.

In this example, we can see that the conflict is between Django <1.11 and ==1.8.18 on one hand, and >=1.11 on the other.

In your case, the packages and version numbers affected will be different, but the principle is the same.

**But I didn’t change anything in my project!**
Because of the way pip works, even if you don’t change anything at all in your project, simply rebuilding it can pull in new packages, if they were unpinned. Whenever the project is built, it will select the latest versions of unpinned packages, and those versions may introduce new, incompatible, requirements of their own.

Identify the problem requirement

The question now is to ascertain which of these requirements we will accept and which we will change.

In example above, we have a strong clue. The most firmly-pinned of these requirements is ==1.8.18. All the others are more loosely pinned. That suggests that Django 1.8.18 has been specified for a good reason.

In this example, searching through the log for Django==1.8.18 will reveal:

```
adding Django==1.8.18
   from aldryn-django==1.8.18.1
```

which means that the requirement for Django 1.8.18 has come from the Aldryn Django addon. So, while that version of the addon is specified, it’s the requirement for Django>=1.11 that is the problem.

There is a second clue in the log that indicates which requirement is the problem. As well as being incompatible with Django==1.8.18, the requirement for Django>=1.11 is also incompatible with another requirement: Django<1.11. A requirement that conflicts with multiple other requirements is most likely to be the one we should address.

So in this case, we now know that the Django>=1.11 requirement is the one to tackle.

Identify where the requirement comes from

The next question is: where does the requirement for Django>=1.11 come from? A search in the log for the string >=1.11 will reveal this - for example (again, your own results will be different, but you will see something in this pattern):

```
adding django<2.2,>=1.11
   from djangocms-attributes-field==0.4.0
```

meaning that djangocms-attributes-field==0.4.0 wants to install a version of Django greater than or equal to 1.11 but less than 2.2.

We can quickly verify this by checking the setup.py in the 0.4.0 branch of the djangocms-attributes-field repository, where the incompatible requirement is introduced (it’s not present in earlier versions).

Now we know that djangocms-attributes-field 0.4.0 has an incompatible Django requirement, so specifying a version lower than 0.4.0 (djangocms-attributes-field<0.4.0) should solve the problem.

Before doing that, it is wise to check what packages require djangocms-attributes-field, and what versions. So repeat the process above: search in the log for djangocms-attributes-field. You might find, for example:

```
adding djangocms-attributes-field>=0.1.1
   from djangocms-file==2.0.2
djangocms-link==2.1.2
djangocms-picture==2.0.5
djangocms-style==2.0.2
djangocms-video==2.0.3
```
meaning that all those packages have specified a version of djangocms-attributes-field greater than 0.1.1. In other words, there is nothing that seems to be incompatible with djangocms-attributes-field<0.4.0, so we can add:

```
djangocms-attributes-field<0.4.0
```

to the project’s requirements.in file (outside the section that will be overwritten) and test it, by rebuilding the project with:

```
docker-compose build web
```

If that completes without an error, you will know that you have successfully identified and addressed the dependency conflict.

**Repeat the process**

Often you will need to repeat the process, as further dependency conflicts will be revealed after you have solved the first one. Each time you will need to pin the problem package in requirements.in and test the build with docker-compose build web, until you have no further conflicts.

**How to prevent this from happening again**

In general, the answer is to pin packages firmly, in each place that requirements are given. A project’s requirements can be specified:

- by the addons system in the Control Panel
- in its requirements.in (addons are automatically listed here too)
- as dependencies of any addons, in their setup.py files
- as any dependencies of dependencies

You have more control over some of these than others. The easiest way to do this is to pin requirements manually as necessary in requirements.in. However, if you want more thorough and precise control, please see *How to pin all of your project’s Python dependencies.*

**How to get help when you have a problem**

In order to help Divio support help you, it’s important to provide us with the information we need, and to be sure it’s something that we can actually help you with.

This page contains a report template and an example of a good report.

**Grant support access**

We will need you to grant support access to your account. Please go to your privacy settings, and grant consent for Support Access.

Preferably, grant support for one year so that on future occasions we will not need to ask you again. The access you grant will be used exclusively for support purposes within Divio.
Provide key information

- The dashboard URL of each project - for each project you are referring to, we need its dashboard URL, in the form https://control.divio.com/control/...
- The precise steps to replicate your issue - we may need to be able to replicate your issue - what action(s) must we take to see it for ourselves? If there are multiple steps involved, please list them.
- What you expect to happen - we need to understand what you expected to happen. If you expected some output or result, describe it, bearing in mind that we may not be familiar with your project.
- What actually does happen - describe the unexpected output or result. Include logs, error messages from the server and browser and so on.
- Tell us what troubleshooting steps you have taken so far - please check:
  - Have you set up the project locally?
  - Does the issue present itself there?
  - If appropriate, test with your local project in live configuration.
  - Are you using logging to help understand the behaviour of the program?
Include any relevant information from troubleshooting in your report.

Template

You may find it helpful to copy and paste this template into your support requests:

Dashboard URLs
  - https://control.divio.com/control/...

Steps to replicate
  - step 1
  - step 2
  - step 3

What we expected to happen
[description]

What actually happened
[description]

Troubleshooting information
The issue [does/does not] occur when running the project locally in live configuration.

Additional information
[include error messages, links to logs, etc]

Example report

A good report might look something like this:

We are having problems with a form in our DynaCorp Global project https://control.divio.com/control/3097/edit/50704/. To see this:
• log in at https://example.com/clientarea with the username “diviosupport” and password “T3mpP4sswd” that we have prepared for you

• in the “Manage your holding” page at https://example.com/clientarea/holdings/edit, change the text in the “Name” field in the form that appears, and press “Update”

At this point you should be redirected to https://example.com/clientarea/holdings, with the data you entered.

Instead what actually happens is that sometimes the expected data will appear, but approximately 50% of the time, it will not.

There are no errors in the logs or browser. The error occurs only on the Live server, not on Test or locally (even when running in Live configuration).

As well as giving us enough information to continue investigating further, the inclusion of information about where it has been tested provides some valuable clues as to the nature of the problem.

Ensure that we have access to your project

If your project uses the Divio Git server, we’ll be able to set it up locally for testing if we need to. However if you use a private remote Git repository, this won’t be possible unless you can provide us with access to the repository. Usually we will provide you with a public key to add to the repository.

What we can and can’t help with

Our technical support is limited in scope to our platform and infrastructure:

• general questions about the use of the platform and its tools
• use and configuration of the local development environment
• best practices for project configuration on Divio
• best practices for project migration to Divio
• deployment issues related to our infrastructure

We are unable to provide support for:

• general questions about development
• debugging of user applications or third-party software
• users’ local hardware/software set-up

Please note that we are able to provide more in-depth technical support for Business-class projects than those on Economy plans. For more information please see our full support policy.

4.3 Technical reference

4.3.1 Divio software components

The Divio application

The Divio app is a GUI application for interacting with local projects and the Cloud. It also sets up the Divio Shell.
Controls

Most of the controls in the Divio app correspond to Divio CLI or docker-compose commands.

Project list

The Divio application opens with a list of the projects available to you.

Select from All projects, or the projects in your Personal/Organisation views. You can also filter projects by name using the search field. The equivalent command is divio project list.

Select a project to see its status in your local environment.

Toolbar controls

The toolbar at the bottom of the application interface is always available and provides some useful controls:

Open shell will open a shell environment, running in Docker itself, with your keys and access to projects set up automatically.

The other icons give you:
• Docker status
• help options
• Divio application preferences

Setting up a project

When you select a project for the first time, you will be given the option to set it up.

When you select **Set up project**, the Divio application will clone the project’s repository to the directory specified in the application’s preferences, build it, and finally pull down its media and database. While it does this, it will also display the local build log.
The command-line equivalent is `divio project setup <project slug>`; the process is described in more detail in the project build process. The process can take a few minutes.

Once successfully set up, the application will show more information and options for managing the project.
Managing a project

Project list/open Dashboard

Return to list of projects; open the project Dashboard in the Control Panel (equivalent to divio project dashboard).

Download/upload

4.3. Technical reference
Download and Upload controls for:

- code (equivalent: using `git` commands to interact with the repository)
- media (equivalent: using `divio project push media` and `divio project push media` commands)
- database (equivalent: using `divio project push db` and `divio project push db` commands)

Open local files

Open the local project directory for access to the files.

Start/stop

Start to launch the local project (or Stop if it is already running).

Options when running

The other options are only available when the project is running locally:

- open the local site in your browser
- open a console displaying the site runtime logs in a Divio shell; equivalent to `docker-compose logs -f --tail=100`
- open a shell inside the local site’s web container; equivalent to `docker-compose exec web /bin/bash`
- present additional options for managing the Docker build:
  - Update - equivalent to `divio project update` (pulls latest Git commits, rebuilds)
  - Rebuild - equivalent to `docker-compose build web`
  - Reset - equivalent to `docker-compose rm` to tear down the project, followed by `divio project setup` to rebuild it.
First run

When first run, the Divio app will:

- download Docker
- install Docker
- launch it
- set up a local Docker image that provides a Bash shell interface for issuing `divio` commands, even if the Divio CLI has not been installed globally

The Divio Shell

The Divio Shell is a pre-configured shell environment for interacting with local Divio projects. It’s launched with the **open shell** button in the toolbar of the Divio app, and drops you in a bash prompt, in your Divio workspace directory.

The Divio Shell is configured with the SSH keys required to give you access to our Cloud servers. You don’t have to use the Divio Shell (you can use an ordinary session in your terminal) but in that case you will need to set up keys yourself.

**Important:** The Divio Shell and the local container’s bash shell are quite different.

- The Divio Shell is for interacting with your Divio projects.
- The local container bash shell is inside an instance of a particular project, allowing you to interact with its program code and operations.

Creating the Divio shell

The Divio app creates the shell by running a sequence of commands, expanded here for clarification:

```bash
# clear the terminal window
clear
# set the path for this shell
PATH=$HOME/.local/bin:/usr/local/bin:$PATH
# clears the DOCKER_HOST environment variable, in case something else has set it
unset DOCKER_HOST
# runs a docker command in a new container, with interactive TTY access, removing it on exit
docker run -it --rm \
  # ... mounting these volumes in the container:
  -v '/var/run/docker.sock:/var/run/docker.sock:rw' \
  -v '/Users/daniele/.netrc:/home/divio/.netrc:rw' \
  -v '/Users/daniele/.aldryn:/home/divio/.aldryn:rw' \
  -v '/Users/daniele/divio-cloud-projects:/Users/daniele/divio-cloud-projects:rw'
  # ... using the image:
divio/divio-app-toolbox:daniele-0.10.5-daniele.procida_divio.ch
  # ... and in the new container, run the following commands:
  cd  /Users/daniele/divio-cloud-projects
  divio doctor
  bash
```

4.3. Technical reference
**Divio CLI**

*divio-cli* is a Python-based command line application, and can be installed via pip:

```bash
pip install divio-cli
```

The *divio-cli source code* is available on GitHub.

It’s also available:

- in the Divio Shell
- at the bash prompt in Divio web containers

**Important:** When using the *divio-cli* other than in a Divio Shell or web container, you will need to install the *aldryn-client* as well to allow the *divio-cli* to perform certain operations related to addons. To install:

```bash
pip install aldryn-client
```

### divio-cli command reference

The *divio-cli* is invoked with the command *divio*.

Its syntax is:

```bash
divio [OPTIONS] COMMAND [ARGS] ...
```

#### Options

Options are:

- `-d`, `--debug` Drop into the debugger if the command execution raises an exception.
- `--help` Show a help message (most commands also include help messages of their own)

#### Commands

*addon*

The *addon* command is used when in the directory of a local addon.

It take one of three commands as an argument:

- `register` Registers an addon with the Divio addons system
- `upload` Uploads an addon to Divio
- `validate` Validates basic aspects of an addon’s configuration
backup

The backup command manages project backups.
It take one command as an argument:

**decrypt** Decrypts an encrypted backup file.

`decrypt` takes three arguments: `KEY BACKUP DESTINATION`.

Example:
```
divio backup decrypt key backup destination
```

boilerplate

The boilerplate command is used when in the directory of a local boilerplate.
It take one of two commands as an argument:

**upload** Uploads a boilerplate to Divio

**validate** Validates basic aspects of an boilerplates’s configuration

doctor

The doctor command checks that your environment is correctly configured.

```
divio-cloud-projects divio doctor
```
Verifying your system setup
✓ Login
✓ Git
✓ Docker Client
✓ Docker Compose
✓ Docker Engine Connectivity
✓ Docker Engine Internet Connectivity
✓ Docker Engine DNS Connectivity

login

Authorise your machine with Divio. `divio login` opens your browser at https://control.divio.com/account/desktop-app/access-token/, where you can copy an access token to paste into the prompt.

project

The project command helps manage projects locally and on the Cloud.
Its general syntax is `divio project [OPTIONS] COMMAND [ARGS]...`

Except where specifically indicated below, the project command is specific to a particular project and must be executed within an existing project directory.

**cheatsheet** Opens the project’s cheatsheet page in the Control Panel.

**dashboard** Opens the project’s Dashboard in the Control Panel.
**deploy**  Deploys the project’s Test or Live servers.

`deploy` takes `test` or `live` as an argument, for example:

```
divio project deploy test
```

Options:

---remote-id INTEGER  Remote Project ID to use for project commands. Defaults to the project in the current directory using the .aldryn file.

**deploy-log**  Returns the latest deployment log for the project’s Test or Live servers.

`deploy-log` takes `test` or `live` as an argument, for example:

```
divio project deploy-log test
```

**develop**  Adds a package in development to the project.

Usage: `divio project develop [OPTIONS] PACKAGE`, where `PACKAGE` is the name of the addon package.

Options:

--no-rebuild  Don’t rebuild the Docker container

**Note:**  What `divio project develop <addon>` actually does is:

- checks `addons-dev` for the named addon
- puts the addon on the Python path
- adds the addon to `requirements.in`, as `-e /app/addons-dev/tutorial-django-debug-toolbar`
- adds any dependencies
- runs `docker-compose build web`.

**env-vars**  Get and set environment variables. By default, these operations work on the Test server (e.g. `divio project env-vars --set SOMEKEY somevalue` will be applied to the Test server, and will appear there).

Note that this command applies only to the Live and Test servers, not the local server. See *Local environment variables*.

Usage: `divio project env-vars [OPTIONS]`

Options:

-s, --stage TEXT  Get data from sever (test or live)
--all, --custom  Show all or only custom (the default) variables
--json  Use JSON output
--get  Get a specific environment variable (get VARIABLE)
--set  Set a specific custom environment variable (set VARIABLE VALUE)
--unset  Unset an environment variable (unset VARIABLE)
--help  Show a help message
export  Exports the local database to local_db.sql.

Usage: divio project export db

import  Imports a database dump file into the local database.

Usage: divio project import db [path]

If the path argument is not supplied, it will expect a file local_db.sql.

list  Lists your Divio projects.

Not specific to a particular project.

live  Opens the project’s Live site in the browser.

open  Open the local project’s site in the browser.

pull  Pulls the database or media files from the Divio cloud environment.

Takes a required argument, db or media, followed optionally by test or live (if not specified, defaults to test), and by --remote-id <project id> to pull from another project.

push  Pushes the database or media files to the Divio cloud environment.

Takes a required argument, db or media, followed optionally by test or live (if not specified, defaults to test), and by --remote-id <project id> to push to another project.

setup  Replicates and builds a Divio project locally.

Takes a single argument, the slug of the project.

Can be run outside a project folder.

status  Shows the status of the local project, shutting down its containers.

stop  Stops the local project (if it is running).

test  Opens the project’s Test site in the browser.

up  Starts up the local project.

update  Updates the local project with new code changes from the Cloud, then builds it. Runs:

```
git pull
docker-compose pull
docker-compose build
docker-compose run web start migrate
```

version

Returns version information about the divio-cli.

The Cloud and local shells

To interact with the environment in your project, for example to run commands using its code-base, you can make use of a shell.

On the Cloud you can SSH into a Cloud shell instance (available on Managed Cloud projects only). Locally, you can use the local Shell.

Once inside the shell, you can inspect the environment, run commands (such as python manage.py migrate). It’s particularly useful to be able to drop into a:
• Python console: python
• Django shell: python manage.py shell
• database shell: python manage.py dbshell

**Important:** The Cloud and local shells described here are not to be confused with the *Divio Shell.*

The Divio Shell is a convenient environment on your own computer, configured for interaction with your Divio account and projects, and the local development environment. The Cloud and local shells will provide you with terminal access inside a running project.

### Using a Cloud shell

The Cloud shell is only available for Managed Cloud projects.

Your Cloud server must be deployed in order to reach it via SSH. You can copy SSH connection details from the appropriate server pane in the Control Panel:

The command can be pasted into a terminal session on your own machine. An instance of your web application will be spun up in a new container, and after a moment you'll be logged in to it as `root`.

SSH sessions are limited to 30 minutes, regardless of any activity.

### The Cloud shell instance

The container you’re connected to is a brand new instance. It will not be one actually serving your site on the web, but a new one that uses the same configuration, database and so on.

**Files** - any files you create or change on this instance will not affect those on any other containers.

**Processes** - each session is isolated from any other extant processes (web processes, workers, other shell sessions).

**Caches** - if your site’s cache relies on the database (the default in Divio projects) then your container will be able to make use of it, clear it and so on. However, if you’re using for example a `locmem` cache, it will not be available to your container.

### The local shell

**Open the shell**

The Divio app provides a convenient short-cut in its toolbar:

Alternatively, you can use the command-line. In the project directory, run:
```bash
docker exec web bash  # if the web container is already running
```

or:

```bash
docker-compose run --rm web bash  # if you need to start the container too
```

The `.aldryn` file

Your local Divio set-up can be configured via the `~/.aldryn` file.

It contains a JSON dictionary, for example:

```json
{
   "update_check_timestamp": 1501185567,
   "skip_doctor_checks": [
      "docker-machine"
   ]
}
```

The `skip_doctor_checks` are particularly useful, and can be used to disable certain automated checks run by the Divio app and the `divio doctor` command (for example, if you are working offline, or with network restrictions).

The checks that `divio doctor` runs are classes in `check_system.py`. Add them to the list in `skip_doctorChecks` to disable them:

- **login** LoginCheck
  - Can we log in to the Control Panel?

- **git** GitCheck
  - Is Git is installed locally?

- **docker-client** DockerClientCheck
  - Is the `docker` command available?

- **docker-machine** DockerMachineCheck
  - Is the `docker-machine` command available?

- **docker-compose** DockerComposeCheck
  - Is the `docker-compose` command available?

- **docker-server** DockerEngineCheck
  - Is the Docker daemon running, and can we connect to it?

- **docker-server-ping** DockerEnginePingCheck
  - Does the Docker container have connectivity to the outside world? (Checks by running `docker run --rm busybox sh -c "ping -c 1 -w 5 8.8.8.8"`.)

- **docker-server-dns** DockerEngineDNSCheck
  - Does a Docker container have connectivity to the outside world? (Checks by running `docker run --rm busybox sh -c "timeout -t 5 nslookup control.divio.com"`.)
4.3.2 Docker commands and configuration

Docker commands

Note: It is beyond the scope of this documentation to provide a complete reference for Docker-related commands. This page is concerned only with providing basic reference for these commands in the context of Divio projects.

The docker-compose command

Docker Compose, invoked as docker-compose, is used to manage Docker applications.

The command is executed in a Docker application directory, and makes use of the project’s docker-compose.yml file. Its general form is:

```
docker-compose <command>
```

**build** Builds the services (i.e. containers) listed in the docker-compose.yml file. Optionally, takes the name of a particular service to build as an argument.

**rm** Removes (i.e. deletes) the project and its containers.

The docker command

The docker command is used to manage images and containers.

Usage:

```
docker <command>
```

**ps** List running Docker containers. Example:

```
docker ps
CONTAINER ID   IMAGE          COMMAND                  CREATED           STATUS  PORTS               NAMES
27ff3e661027   postgres:9.4   "docker-entrypoint..."   6 days ago        Up 8    5432/tcp            demoproject_db_
```

The Dockerfile

Each Docker image is defined by a Dockerfile, that describes what is in the image and how containers created from it should be built.

The Dockerfile is simply a text document, containing all the commands that would be issued on the command-line to build an image - in short, the Dockerfile defines an environment.

The Dockerfile is built automatically, and populated with appropriate commands (see below). However, you can also add any commands you wish to the Dockerfile, for example to install system packages, or to configure the environment.
**Important:** The Dockerfile executes its commands in sequence. This means that commands to install Node (for example) must come before commands to run Node packages.

### How the Dockerfile is automatically populated

Hashes (#) in the Dockerfile indicate a comment. Sections within angle brackets are autogenerated by the Divio Control Panel, and may be updated or changed on deployment without warning.

Removing these wrapping tags will prevent a section being populated or changed.

### The empty Dockerfile at project creation

The Dockerfile starts life at project creation thus:

```bash
# <WARNING>
# </WARNING>

# <DOCKER_FROM>
# </DOCKER_FROM>

# <NPM>
# </NPM>

# <BOWER>
# </BOWER>

# <PYTHON>
# </PYTHON>

# <SOURCE>
# </SOURCE>

# <GULP>
# </GULP>

# <STATIC>
# </STATIC>
```

These sections are in effect placeholders for Docker commands and configuration that will be used to define the project later.

### The <WARNING> section

The <WARNING> is always populated.

```bash
# <WARNING>
# Everything within sections like <TAG> is generated and can
# be automatically replaced on deployment. You can disable
# this functionality by simply removing the wrapping tags.
# </WARNING>
```
The `<DOCKER_FROM>` section

This is determined by the project’s base project version. If you update the base project in the project’s General Settings in the Control Panel, this will be updated on the next deployment.

For a project built on the aldryn/base-project:py3-3.23 image, corresponding to the Base Project: Python 3 v3.23:

```
FROM aldryn/base-project:py3-3.23
```

The `<NODE>` section

This section will be supplied by a Boilerplate that includes Node components, for example in the django CMS Sass Boilerplate.

An example that uses other files supplied by the Boilerplate (such as install.sh) to set up the Node environment:

```
ADD build /stack/boilerplate

ENV NODE_VERSION=6.10.1 \
    NPM_VERSION=3.10.10

RUN bash /stack/boilerplate/install.sh

ENV NODE_PATH=$NVM_DIR/versions/node/v$NODE_VERSION/lib/node_modules \
    PATH=$NVM_DIR/versions/node/v$NODE_VERSION/bin:$PATH
```

The `<NPM>` section

If package.json (specifying Node packages that should be installed) is present in the root of the project, then instructions will be inserted to copy it to the root of the image and install the packages.

```
COPY package.json /
RUN (cd / && npm install --production && rm -rf /tmp/*)
```

The `<BOWER>` section

If both bower.json and .bowerrc are present in the root of the project, then the deployment process will insert:

```
COPY bower.json .bowerrc /app/
RUN bower install \ 
    --verbose \
```

(continues on next page)
The `<PYTHON>` section

If `requirements.in` is present in the project, then at deployment time the Control Panel will ensure that this section contains appropriate instructions to handle installation of Divio Cloud addons and other packages. The exact contents of this section will depend on the project, for example:

```plaintext
# <PYTHON>
ENV PIP_INDEX_URL=${PIP_INDEX_URL:-https://wheels.aldryn.net/v1/aldryn-extras+pypi/}
  -WHEELS_PROXY_URL=${WHEELS_PROXY_URL:-https://wheels.aldryn.net/v1/aldryn-
    extras+pypi/${WHEELS_PLATFORM: aldryn-baseproject-py3/}}
COPY requirements.in /app/
COPY addons-dev /app/addons-dev/
RUN pip-reqs compile &&
    pip-reqs resolve &&
    pip install --no-index --no-deps
    --requirement requirements.urls
# </PYTHON>
```

If `requirements.txt` is present in the project, then the `pip-reqs compile` instruction will be removed. See *How to pin all of your project’s Python dependencies* for why you might want to do this.

The `<SOURCE>` section

The `<SOURCE>` section copies the project files to the `/app` directory of the container.

```plaintext
# <SOURCE>
COPY . /app/
# </SOURCE>
```

We do this late in our Dockerfile by default. This is because it copies the entire repository into the container, meaning that if anything is changed in the repository, it would invalidate all the following layers, which would have to be rebuilt from scratch rather than using cached layers. For reasons of economy, we keep this as late as possible.

If other parts of the repository need to be copied into the container earlier in the process, these should be explicitly specified as required.

The `<GULP>` section

If `gulpfile.js` is present in the root of the project, then instructions will be inserted to run the `gulp build` process:

```plaintext
# <GULP>
ENV GULP_MODE=production
RUN gulp build
# </GULP>
```
The <STATIC> section

<STATIC> is always populated, with a command to copy static files to the location from where the web server will serve them:

```bash
# <STATIC>
RUN DJANGO_MODE=build python manage.py collectstatic --noinput
# </STATIC>
```

The docker-compose.yml file

In order to do something useful with containers, they have to be arranged as part of a project, usually referred to as an ‘application’. This is what a docker-compose.yml file does, specifying what images are required, what ports they need to expose, whether the have access to the host filesystem, what commands should be run, and so on.

Function of docker-compose.yml

In the our project architecture, the docker-compose.yml file is not used for cloud deployments, but only for configuration of the local server. On the cloud, the deployment is taken care of by dedicated systems on our servers. This means that entries in or changes to docker-compose.yml will not affect cloud deployments in any way.

Services defined in docker-compose.yml

By default, the docker-compose.yml in Divio projects builds a web service in a container using its Dockerfile. It also builds a db service, from a standard postgres or other database image. Most Divio projects will use a docker-compose.yml along these lines.

```yaml
web:
  build: .
  links:
    - "db:postgres"
  ports:
    - "8000:80"
  volumes:
    - ".:/app:rw"
    - ".:/data:/data:rw"
  command: python manage.py runserver 0.0.0.0:80
  env_file: .env-local
db:
  image: postgres:9.4
  volumes:
    - ".:/app:rw"
```

Some projects will have additional services (such as Celery for example) defined.

The web service

The first definition in the file is for the web service. In order, the directives mean:

- **build**: build it from the Dockerfile in the parent directory
Divio Cloud developer handbook Documentation, Release 1.0

• links: link to the database container
• ports: map the external port 8000 to the internal port 80
• volumes:
  – map the parent directory on the host to /app in the container, with read and write access
  – map the data directory on the host to /data in the container, with read and write access
• command: by default, when the command docker-compose run is issued, execute python manage.
py runserver 0.0.0.0:80
• env_file: use the .env-local to supply environment variables to the container

The volumes directive

When you execute a docker-compose command, the volumes directive in docker-compose.yml file mounts source directories or volumes from your computer at target paths inside the container. If a matching target path exists already as part of the container image, it will be overwritten by the mounted path.

For example:

```yaml
volumes:
  - ".:/app:rw"
  - ".:/data:/data:rw"
```

will mount the entire project code (at the relative path .) as the /app directory inside the container, even if there was already an "./app" directory there, in read-write mode (i.e. the container can write as well as read files on the host).

This allows you to make changes to the project from your computer during the local development process, that will be picked up by project inside Docker. These changes will be available to the project only as long as the host directory is mounted inside the container. In order to be made permanent, they need to be committed into the repository so that they will be picked up when the image and container are rebuilt.

Implications for local testing

Nearly everything in /app in the container is also present in the project repository and thus on the host machine. This means that it is safe to replace the container’s /app files with those from the host.

However, any files in /app that are placed there during the build process, i.e. the execution of the Dockerfile, will not be available in the local environment. For a standard Django project, these will include:

• the compiled pip requirements, in requirements.txt
• collected static files, in static_collected

In most cases, this will not matter, but sometimes these files are required in local development. For example, the requirements.txt may contain useful information about dependency relationships, or the Dockerfile may have performed custom processing of static files.

In that case, the – ".:/app:rw" line can be commented out in docker-compose.yml. In this case, the container will use the files baked into the image, and will not use the local host’s files.

This will allow local configuration to replicate the cloud environment even more closely.

4.3. Technical reference
Environment variables

Environment variables are loaded from a file, specified by:

```
env_file: .env-local
```

The db service

The second definition is for the db service. On the cloud, the project’s database runs on an AWS server; locally, it runs on a Postgres instance in db.

The directives mean:

- `image`: build the container from the `postgres:9.4` image
- `volumes`: map the parent directory on the host to `/app` in the container, with read and write access

See *Expose the database’s port* for an example of adding configuration to `docker-compose.yml`.

4.3.3 Python/Django application configuration

The `settings.py` file

Divio Django projects that use our addons framework are shipped with a `settings.py` file that hooks into the framework. The framework allows addon applications to configure their own settings programmatically.

At first sight, this `settings.py` file may seem unusual, but in fact it behaves as a standard Django settings module.

**INSTALLED_ADDONS**

The `INSTALLED_ADDONS` lists the addons installed by the addons framework. The list is populated automatically:

- on the Control Panel, when addons are added or removed
- locally, when the `divio project develop` command is run

Items are inserted between the `<INSTALLED_ADDONS>` tags. If you need to add items to the list manually while developing, add them *outside* the tags, otherwise your changes will be overwritten the next time `divio project develop` is run.

```
INSTALLED_ADDONS = [
    # <INSTALLED_ADDONS>
    ...
    # </INSTALLED_ADDONS>
]
```

Settings in Divio projects can either be *configured automatically via the addons framework*, or set manually.

Automatic settings loading

Using this list of `INSTALLED_ADDONS`, the:

```
import aldryn_addons.settings
aldryn_addons.settings.load(locals())
```
section that follows checks each one for any settings that it has to apply. These settings will be loaded into the settings module at this point. For example, INSTALLED_APPS will be populated appropriately.

Any settings that have been loaded can be manipulated. For example, to add new applications to INSTALLED_APPS, you can add them in:

```
INSTALLED_APPS.extend([
    # Extend the INSTALLED_APPS setting by listing additional applications here
])
```

It’s important to understand which settings are applied automatically.

If you declare a setting such as INSTALLED_APPS **before** aldryn_addons.settings.load(locals()), it may be overwritten by the addon system.

If you declare it **after** aldryn_addons.settings.load(locals()), it will overwrite any configuration performed by the addon system, with possibly unpredictable results.

See How settings are handled in Django addons for an overview of how settings are handled in general, and How to configure Django settings for advice on how to manipulate them.

### 4.3.4 Coding in Divio applications

**Working with your project’s media storage in Python applications**

**Introduction**

Default file storage on Divio projects is handled by dedicated storage systems entirely separate from the application. In our architecture, the same site may be running as several different instances, on several different application hosts (this is one reason why Divio projects can be scaled, because new application instances can be created to meet increasing demand).

Although each of those instances will have its own local file storage, this will be independent of each of the others, and it won’t persist - once that instance ceases to exist, so will the files. That storage will also be inaccessible to any other instances of the application.

This means a project’s applications, cron jobs or other process can’t expect to save files to its local storage, and then expect to find them again.

**Our storage service providers**

Instead, the applications must use our storage services. These are Amazon Web Services’s S3 service, or a generic S3 hosting service via another provider. Currently, most projects use Amazon’s own S3 service, with the exception of projects in our Swiss region.

**Working with our storage backends in Django**

For most Django applications, this won’t require any additional work. Django is able to use multiple storage backends, all addressed through a common API. This is the safe and correct way to handle files in Django, so that applications can abstract from details of the storage implementation, and simply not need to know about it.

As long as an application uses Django’s storage API, rather than attempting to manipulate Python File objects directly, it doesn’t need to do anything differently.

Similarly, an application should not rely on knowing or manipulating a File object’s file path.
Use Django’s defined `DEFAULT_FILE_STORAGE`, not `FileSystemStorage`

Your code may use Django’s `FileSystemStorage`. This provides basic file storage, on a filesystem local to the code. For the reasons described in the *Introduction* it is therefore not suitable for use on Divio.

Instead, you must use the storage as defined by Django’s `DEFAULT_FILE_STORAGE` - which you can do simply by not explicitly specifying a storage system, and using `django.core.files.storage.default_storage`.

See also Django’s discussion of the subject.

File storage in third-party applications

Ideally, third-party applications in your project should respect this for their own file handling. This is not always the case however. In some cases the application may need to be configured explicitly. More problematically, some applications may have hard-coded expectations for the file storage system, and these will need to be rewritten.

Private file storage

Our storage backend does not support private file storage (i.e. requiring authentication) on S3 objects.

If you need private storage, you can define an additional Django storage backend in your project, which sets S3 objects to be private as required.

Whenever you need to manage private files, you will need to invoke this custom backend.

The backend can use the buckets we provide to do this, but please be aware that if you restore a backup, or use our tools to push files, *all the files will become public.*

Alternatively, you can use a bucket of your own with this backend.

Using Easy Thumbnails

Easy Thumbnails is the most widely-used image processing application in the Django ecosystem.

On Divio, `THUMBNAIL_DEFAULT_STORAGE` for Easy Thumbnails needs to be set explicitly, even if `DEFAULT_FILE_STORAGE` has been set.

In most projects on Divio, Django Filer is installed. This takes care of the `THUMBNAIL_DEFAULT_STORAGE` - if Django Filer is installed, you don’t need to do anything else to use Easy Thumbnails correctly.

In the cases where it’s not, it’s necessary to do the same thing manually in the `settings.py`:

```python
# If the DEFAULT_FILE_STORAGE has been set to a value known by
# aldryn-django, then use that as THUMBNAIL_DEFAULT_STORAGE as well.
from aldryn_django import storage

for storage_backend in storage.SCHEMES.values():
    if storage_backend == DEFAULT_FILE_STORAGE:
        THUMBNAIL_DEFAULT_STORAGE = storage_backend
    break
```
Loading media files into your applications' pages

Sometimes an application in your project will need to load media files using JavaScript. Since your media files are held on a server under a different domain from the application, browsers may refuse to allow this cross-domain loading for security reasons.

There are two solutions to this.

Load media from static

One is to make sure that all files you need to load are in your site’s static files, rather than media. (The static files are served from the same domain as the application itself, so browsers will be able to load files using JavaScript without complaint).

This has the advantage of not running into the possibility of using JavaScript to load user-submitted material (which could include material uploaded by untrusted users).

Enable CORS headers

The other solution is to enable CORS (“Cross-origin resource sharing”) headers on the media.

This must be done by our infrastructure team, on a per-website basis. Please contact Divio support to request this.

Storage speed and performance

Note that if you need to make many read/write operations to file storage, or are working with very large objects, that the speed you experience on the cloud can be considerably less than what you experience in the local development environment.

The local development environment has the advantage of locally-attached storage, and should not necessarily be taken as a guide to performance on the cloud.

In most cases, this won’t actually matter. However, if your code works very intensively with storage, it can be more efficient and faster to do all that work on the application instance’s own local filesystem, in a temporary directory, and then send the finished work to the remote storage.

File compression in Divio Django applications

Default behaviour in our projects

By default, we apply gzip compression to:

- Django-served content, such as HTML files
- static files, when DEBUG is False

This compression can be disabled by setting the DISABLE_GZIP environment variable to True.

For Django-served content, we apply Django’s GZipMiddleware.

For static files, we use our own STATICFILES_STORAGE classes to gzip static files after they are collected. See GZippedStaticFilesMixin.post_process(), in Aldryn Django’s storage.py.
Using **DISABLE_GZIP**

When gzip is disabled, gzipped versions of static files are not used (though they are still collected). A default Django storage class is used instead.

**Using **DEBUG = True**

When **DEBUG = True**, the non-gzipped versions of the files are loaded by templates, whether or not **DISABLE_GZIP** has been applied.

**Using Django Compressor**

Django Compressor is a popular tool for additional compression functionality (for example, consolidation of multiple files into one).

When using Django Compressor, note that:

- offline compression must be used
- static files must be compressed before collection (python manage.py compress must come before collectstatic in the Dockerfile)

To verify correct and expected operation of Django Compressor, use the local server in live configuration, taking note to disable volume mapping that would affect the collected compressed files.

**Sending email in Divio applications**

Divio does not provide mail services. To send mail from your Django applications, you will need to provide the appropriate configuration.

Django provides email wrappers around Python’s `smtplib` module.

**Configuration**

The following configuration settings can be provided:

- **Basic settings** `EMAIL_HOST` and `EMAIL_PORT` (defaults to 25)
- **Authentications settings** `EMAIL_HOST_USER` and `EMAIL_HOST_PASSWORD`
- **Secure authentication** `EMAIL_USE_TLS` and `EMAIL_USE_SSL`

**Using **EMAIL_URL environment variable**

However, the preferred way to provide these is via an `EMAIL_URL environment variable`, so that your local, Test and Live servers can use their own configuration.

The `EMAIL_URL` is the recommended way of combining the settings into a single variable. For example, suppose you have:

```python
EMAIL_HOST = smtp.example.com
EMAIL_PORT = 25
EMAIL_HOST_USER = janeausten
EMAIL_HOST_PASSWORD = password
```
you can instead use:

```
smtp://janeausten:password@smtp.example.com:25
```

For:

- TLS, add `?tls=True` (and use port 587)
- SSL, add `?ssl=True` (and use port 465)

to the URL. Note that **TLS is preferred**, and you can’t use both. The URL is parsed using the dj-email-url library.

**Additional Django email settings**

Some additional email settings are available in Django. These can be provided as environment variables.

`DEFAULT_FROM_EMAIL` Allows you to specify a default From address for general automated messages from your website.

`SERVER_EMAIL` Specifies a default From address for error messages from your site.

**Usage, testing and troubleshooting**

It’s beyond the scope of this document to discuss usage in detail. The official Django documentation has more information.

It’s useful to be able to test your configuration. You can do this in your project’s local shell or Cloud shell.

Once in the shell, launch the Django shell:

```
python manage.py shell
```

Import the Django `send_mail` function:

```
from django.core.mail import send_mail
```

and try sending a message:

```
send_mail(  
    "Welcome to Divio",  
    "It's great!",  
    "from@example.com",  
    ["to@example.com"],  
    fail_silently=False,  
)
```

The email settings will be taken from the `EMAIL_URL` environment variable, but can be overwritten in the shell - for example:

```
EMAIL_USE_TLS = True
```

### 4.3.5 Addons
Addon configuration with `aldryn_config.py`

A Django application may require some configuration when it is deployed in a project. Typically this will include settings in `settings.py`, but it can also include things like URL patterns that need to be set up.

For Aldryn addons, Divio provides for such configuration through an addon’s `aldryn_config.py` file. This file needs to be in the root directory of the addon.

Through this mechanism you can also allow the user to provide configuration in a simple web form that will be available in the Control Panel.

When the user saves the web form, the data will be stored in the addon’s `settings.json` file in the project repository.

An example from a django CMS addon instance:

```json
{
    "boilerplate_name": "html5",
    "cms_content_cache_duration": 60,
    "cms_menus_cache_duration": 3600,
    "cms_templates": "["content.html", "Content"], ["sales.html", "Sales"]",
    "permissions_enabled": true
}
```

The `aldryn_config.py` file

This file contain a class named `Form` that sub-classes `aldryn_client.forms.BaseForm`:

```python
from aldryn_client import forms

class Form(forms.BaseForm):
    ...
```

The Form class will contain the logic required to manage configuration.

Managing settings

A `to_settings()` method on the `Form` class will be called. Use this to return a dictionary of settings.

It takes two arguments:

- the `cleaned_data` from the form
- a dictionary containing the existing settings

Add or manipulate the settings in the dictionary as required, and return it.

If you wish to accept user-supplied configuration, you will need to add some form fields to the form (see Adding form fields for user-configuration of the Addon below).

Managing URL configuration

`ADDON_URLS` (and `related settings`) to help manage URL configurations via settings.

We can define them in the `to_settings()` method of an application to do this.

Here’s an example of `aldryn_config.py` that inserts URL configurations into a project:
from aldryn_client import forms

class Form(forms.BaseForm):
    def to_settings(self, data, settings):
        settings['ADDON_URLS'] = 'django_example_utilities.urls'
        return settings

See addon URLs for details.

Adding form fields for user-configuration of the Addon

The Form class may contain any number of form fields.

Available fields are:

- aldryn_client.forms.CharField (optional arguments: min_length and max_length)
- aldryn_client.forms.CheckboxField
- aldryn_client.forms.SelectField (required second argument: a list of tuples)
- aldryn_client.forms.NumberField (optional arguments: min_value and max_value)
- aldryn_client.forms.StaticFileField (optional argument: extensions, a list of valid file extensions.)

All fields must provide a label as first argument and take a keyword argument named required to indicate whether this field is required or not.

Here's an example:

class Form(forms.BaseForm):
    # get the company name
    company_name = aldryn_client.forms.CharField("Company name", required=True)

    def to_settings(self, cleaned_data, settings_dict):
        # set the COMPANY_NAME based on company_name
        settings_dict['COMPANY_NAME'] = cleaned_data[company_name]

        # if we are in DEBUG mode, as on the Test server, use the Django console backend
        # rather than really sending out messages (see
        # https://docs.djangoproject.com/en/1.8/topics/email/#console-backend)
        if settings_dict.get('DEBUG'):
            settings_dict['EMAIL_BACKEND'] = 'django.core.mail.backends.console.EmailBackend'

        return settings_dict

Custom field validation

For custom field validation, sub-class a field and overwrite its clean() method. The clean() method takes a single argument (the value to be cleaned) and should either return a cleaned value or raise a aldryn_client.forms.ValidationError with a useful message about why the validation failed.

Example:

4.3. Technical reference
from aldryn_client import forms

class FavouriteColourField(CharField):
    def clean(self, colour):
        colour = super(FavouriteColourField, self).clean(colour)
        if colour == "black":
            raise forms.ValidationError("You can have any colour you like except black")
        else:
            return colour

What configuration method to provide?

There are multiple ways of providing configuration in the addons you create - see How settings are handled in Django addons for an overview. You can choose to provide configuration via any method you like, but some rules of thumb for the appropriate method:

- For highly-sensitive configuration, such as passwords, use an environment variable - it’s safer, because it’s not stored in the codebase.
- For configuration that is specific to each instance of the codebase, or that needs to be different across Local, Test and Live environments, environment variables are recommended.
- For required configuration, it is a good idea to make it visible as a field, so it’s obvious to the user that it needs to be set; similarly if it’s something that a non-technical user might be expected to set.
- If you provide an addon configuration field, make sure it isn’t overridden by other configuration, as that could be confusing to the user.
- The settings.py file makes sense for configuration that isn’t sensitive, and will be the same in different instances of the codebase and can be the same across the different environments.
- The cleaner you keep your settings.py, the better.

Aldryn Django (core Django)

Aldryn Django (aldryn-django) is a wrapper application that installs and provides basic configuration for Django. See aldryn-django.aldryn_config (ensure that you switch to the correct branch) for the all settings it takes and how they are applied.

Most of the key settings are listed below.

Control Panel options

Some settings are exposed in the Aldryn Addon configuration form in the Control Panel. These settings will take priority over those entered as environment variables or in settings.py.

Hash static file names

The Aldryn Django addon includes a Hash static file names option. When selected, Django’s ManifestStaticFilesStorage will be used as the storage backend. This appends an MD5 hash of each file’s contents to its filename, allowing caching headers to be safely set in the far future.
Aldryn Django configures uWSGI to set the Cache-Control header to one year on files with a hash in the filename.

Enable django.contrib.gis

Enables GeoDjango support. Adds django.contrib.gis to INSTALLED_APPS and sets the database engine to django.contrib.gis.db.backends.postgis.

For local development you will need to edit docker-compose.yml, changing postgres:9.4 to mdillon/postgis:9.4. On the Cloud, you will need to make a support request to have the new database enabled for the project.

Remove URL language prefix for default language

When set, will add aldryn_django.middleware.LanguagePrefixFallbackMiddleware to the middleware.

This will cause Django not to use a language prefix in the URL when serving the default language. For example, by default, /about will redirect to /en/about if English is the default language; with this option selected, it will not (and will instead redirect in the other direction).

Note that prior to Django version 1.10, this will not work with projects in which multiple languages are defined.

This is a 302 Temporary Redirect, as determined in Django’s core. It is not safe to use permanent redirects here, because language redirects are content-dependent. A change in the site could cause redirect loops, as described at 301 Permanent vs 302 Temporary redirects, or spurious 404 errors.

Timeout for users session, in seconds

See SESSION_COOKIE_AGE in Session middleware settings.

Environment variable/settings.py options

Security-related settings

Security middleware settings

Each of these settings can be specified as an environment variable (recommended except where indicated otherwise below) or in settings.py. These settings apply to Django’s Security middleware.

- SECURE_BROWSER_XSS_FILTER (default: False)
- SECURE_CONTENT_TYPE_NOSNIF (default: False)
- SECURE_HSTS_INCLUDE_SUBDOMAINS (use settings.py; not available as an environment variable)
- SECURE_HSTS_PRELOAD (use settings.py; not available as an environment variable)
- SECURE_HSTS_SECONDS (default: 0)
- SECURE_REDIRECT_EXEMPT (use settings.py; not available as an environment variable)
- SECURE_SSL_HOST (use settings.py; not available as an environment variable)
- SECURE_SSL_REDIRECT (default: None)
Session middleware settings

- `SESSION_COOKIE_HTTPONLY` (must be `False` for django CMS, default: `False`)
- `SESSION_COOKIE_SECURE` (default: `False`)
- `SESSION_COOKIE_AGE` (also available as a Control Panel setting, default: 2 weeks)

Site-related settings

**DOMAIN_REDIRECTS**

A list of domain names that will redirect to the site’s primary domain name.

By default, this is populated by the Control Panel. If required, it can also be specified as an environment variable on the Live server (recommended) or in `settings.py`.

Setting this manually will allow you to add the internal Divio domain of the site, such as `example.eu.aldryn.io`, to the domains that will redirect to the primary domain. (You may wish to do this if you don’t want users or search engines to discover your site at `example.eu.aldryn.io` rather than `example.com`.)

Note that if you specify `DOMAIN_REDIRECTS` manually, you will need to list all of its secondary domains, as it overrides the setting automatically generated by the Control Panel.

**ALDRYN_SITES_REDIRECT_PERMANENT**

By default, redirects are `302 Temporary Redirect`. When `True`, redirects (where this is appropriate) will be `301 Permanent Redirect`.

Can be specified as an environment variable (recommended) or in `settings.py`.

See [301 Permanent vs 302 Temporary redirects](#) for more information.

Storage settings

**Cache control for static files**

Static files in our Django projects are collected by Django at build time, and served by uWSGI. Aldryn Django configures the command it issues to uWSGI to start static file serving on the basis of project settings. By default, files are served with no `Cache-Control` header applied.

**STATICFILES_DEFAULT_MAX_AGE**

The `STATICFILES_DEFAULT_MAX_AGE` determines the `Cache-Control` header value that uWSGI will use for unhashed files (see the `Hash static file names` option, above). It is not recommended to set this to high values, as the cached versions can continue to be used even after files themselves have been updated.

**DISABLE_S3_MEDIA_HEADERS_UPDATE**

Applications using Aldryn Django will update media file headers by running the command:
python manage.py aldryn_update_s3_media_headers

as a post-build migration/release operation; this can be controlled with the DISABE_S3_MEDIA_HEADERS_UPDATE environment variable. The aldryn_update_s3_media_headers command can cause excessively long deployment times on very large media buckets, so setting this variable to True can avoid that.

Django server settings

See notes on DJANGO_WEB_WORKERS, DJANGO_WEB_MAX_REQUESTS, DJANGO_WEB_TIMEOUT in How to fine-tune your server’s performance.

Email settings

See Sending email in Divio applications for details of available settings.

Miscellaneous settings

- DISABLE_GZIP determines whether Django’s GZipMiddleware will be added to the project’s middleware (default: False)
- TIME_ZONE (default: the appropriate time zone for your server region)
- SENTRY_DSN - if provided, logging to Sentry will be configured automatically

Aldryn SSO (authentication)

Authentication to the Divio platform, and (by default) to user projects running on the platform, is handled by the Divio SSO (single-sign-on) system. This provides a convenient way to authenticate users for Divio projects (whether locally, or on the Test or Live servers) without needing to log in again, as long as they have logged into the Divio Control Panel.

This includes making it possible for users working on projects locally to log in locally with a single click, as they have already been authenticated.

Divio SSO is managed by the open-source Aldryn SSO addon. The system is optional, but is installed by default in all Divio Django projects.

If the addon is uninstalled, then Django’s standard authentication behaviour will apply.

Login methods

The Aldryn SSO addon provides three different login methods to Divio projects:
Depending on how the project is configured, and which environment (local/test/live) it’s running in, different combinations of these options will be shown (you’ll never see all three at once in a real project).

The illustrated options are:

1. **Local development login**

   This is intended to appear on locally-running projects only. The Add user option is a convenient way to add new users to a project.

   See `ALDRYN_SSO_ENABLE_LOCALDEV`.

2. **Django’s standard username/password login form**

   This will not be of any use unless users with passwords exist in the database.

   See `ALDRYN_SSO_ENABLE_LOGIN_FORM`. 
3. Divio single-sign-on

This is intended to appear on projects running in Cloud environments only. It allows users to sign in to their own projects with a single click, once they have authenticated with the Divio control panel.

See `ALDRYN_SSO_ENABLE_SSO_LOGIN`.

Test site protection

By default the Test site is protected so that it’s not publicly discoverable or viewable. Only the owner or an authorised user of the project can view its contents.

This is controlled with the `ALDRYN_SSO_ALWAYS_REQUIRE_LOGIN` environment variable, which is `True` by default and can be overridden by setting it manually.

See also how to apply/remove password protection to Django sites.

Aldryn SSO configuration options

**Important:** The preferred way to set these options is as environment variables.

If you supply them as Django settings declared in `settings.py`, they must appear before `aldryn_addons.settings.load(locals())`. This allows them to be processed correctly by the addons system.

The exception is `ALDRYN_SSO_HIDE_USER_MANAGEMENT`, which is configured via the Control Panel, or by adding the variable after `aldryn_addons.settings.load(locals())`.

More details of how Aldryn SSO processes these settings can be studied at `aldryn-sso.aldryn_config.py`.

**ALDRYN_SSO_ALWAYS_REQUIRE_LOGIN**

Controls whether visitors need to be logged-in. Available options are:

- **True:** Users will need to log in via the SSO system in order to access the site (default for test site).
- **False:** No login is required (default for local and live environments).
- **basicauth:** The site will be protected by basic HTML access authorisation. See `basicauth`.

Can also be specified as an environment variable or in `settings.py`.

**ALDRYN_SSO_ENABLE_LOCALDEV**

Enables Local development login.

When `True` (default for the local environment only) enables the Add user pane in the login form, providing a convenient way to add a new user to the database.

Can also be specified as an environment variable or in `settings.py`.

**Warning:** For obvious reasons, enabling this is strongly not recommended on the Test and Live sites, and there is generally no good reason to manipulate this setting.
**ALDRYN_SSO_ENABLE_SSO_LOGIN**

Enables *single-sign-on*.

Requires a value to be present in *SSO_DSN*, and is automatically set when there is. If enabled when no *SSO_DSN* value has been set, an error will be raised.

Can also be specified as an environment variable or in *settings.py*.

**ALDRYN_SSO_ENABLE_LOGIN_FORM**

Enables *Django's standard username/password login form*.

By default, is enabled when *Hide user management* is *not* enabled.

Can also be specified as an environment variable or in *settings.py*.

**ALDRYN_SSO_ENABLE_AUTO_SSO>Login**

When *True* (the default on all sites) then if SSO login is the only login method enabled, the user will be automatically logged-in via SSO (assuming of course that the user is authorised to do so).

The logic for this condition is:

<table>
<thead>
<tr>
<th>ALDRYN_SSO_ENABLE_SSO_LOGIN</th>
<th>True</th>
</tr>
</thead>
<tbody>
<tr>
<td>ALDRYN_SSO_ENABLE_AUTO_SSO_LOGIN</td>
<td>True</td>
</tr>
<tr>
<td>ALDRYN_SSO_ENABLE_LOGIN_FORM</td>
<td>False</td>
</tr>
<tr>
<td>ALDRYN_SSO_ENABLE_LOCALDEV</td>
<td>False</td>
</tr>
</tbody>
</table>

Can also be specified as an environment variable or in *settings.py*.

**ALDRYN_SSO_HIDE_USER_MANAGEMENT**

This option is presented in the configuration form for the Aldryn SSO addon on the Control Panel (as *Hide user management*). Its effect is to unregister the *User* and *Group* models in the Django admin.

Setting it as an environment variable will have no effect.

Specifying it in *settings.py* will only have an effect if it is declared *after* *aldryn_addons.settings.load(locals())*. This is not recommended except for testing purposes.

For local testing, the *hide_user_management* value in *aldryn-addons/aldryn-sso/settings.json* can be changed, mimicking the effect of the form value.

**Basic access authentication**

Basic access authentication is configured using *ALDRYN_SSO_BASICAUTH_USER* and *ALDRYN_SSO_BASICAUTH_PASSWORD*.

When *ALDRYN_SSO_ALWAYS_REQUIRE_LOGIN* is set to *basicauth*, access to the entire site will require user and password details. This is an *additional layer* of authentication. Access to the admin will still require login by an admin user, and even a logged-in admin user will need to supply the username and password.

See also:
Basic access authentication.

Though the username and password can be specified as an environment variable or in settings.py, the latter is not good practice.

SSO_DSN

The Data Source Name for single-sign-on.

This is set as an environment variable automatically in Cloud Projects, adding the SSO authority to the URL configuration for the project.

If you are providing your own single-sign-on, SSO_DSN can also be specified as an environment variable or in settings.py.

LOGIN_REDIRECT_URL

After login, redirect to the specified URL (by default, to /).

Specifying LOGIN_REDIRECT_URL in settings.py will only have an effect if it is declared after aldryn_addons. settings.load(locals()).

ALDRYN_SSO_LOGIN_WHITE_LIST

A list of internal endpoints that don’t require authentication. Defaults to an empty list.

For example:

```python
from django.core.urlresolvers import reverse_lazy
ALDRYN_SSO_LOGIN_WHITE_LIST = [reverse_lazy('my_whitelisted_endpoint')]
```

Can be specified as an environment variable or in settings.py, or manipulated programmatically in other applications:

```python
if 'ALDRYN_SSO_LOGIN_WHITE_LIST' in settings:
    settings['ALDRYN_SSO_LOGIN_WHITE_LIST'].extend([reverse_lazy('my_whitelisted_endpoint')])
```

ALDRYN_SSO_OVERRIDE_ADMIN_LOGIN_VIEW

We override Django’s admin login view by default, as long as one of the three login options is enabled. This takes better care of logged-in users who are not staff (admin) users.

The standard Django administration login view is available by setting this to False as an environment variable or in settings.py.

Aldryn Addons (addon integration)

The Aldryn Addons framework helps integrate addons and their settings into a Django project.

It’s an open-source package, and is itself an addon. The addons framework is installed by default in all Divio Cloud Django projects.
Aldryn Addons configuration options

Addon URLs

A project, or an addon in it, may need to specify some URL patterns. They could simply be added to the project’s urls.py manually. However, it’s also convenient for addons to be able to configure URLs programmatically, so that when an addon is installed, it will also take care of setting up the relevant URL configurations.

Aldryn Addons provides a way to do this. A Divio project’s urls.py contains:

```python
urlpatterns = [
    # add your own patterns here
] + aldryn_addons.urls.patterns() + i18n_patterns(
    # add your own i18n patterns here
    *aldryn_addons.urls.i18n_patterns()  # MUST be the last entry!
)
```

As well as indicated places for manually-added patterns, it calls aldryn_addons.urls.patterns() and aldryn_addons.urls.i18n_patterns(). These functions, in the urls.py of Aldryn Addons, check for and return the values in four different settings:

**ADDON_URLS and ADDON_URLS_I18N**

These are expected to be lists of URL patterns. Each addon that needs to add its own URL patterns should add them to the lists.

For example, in Aldryn django CMS:

```python
settings['ADDON_URLS'].append('aldryn_django_cms.urls')
```

**ADDON_URLS_LAST and ADDON_URLS_I18N_LAST**

These are not lists, and only one of each can be set in any project - it’s not possible for two applications both to specify an ADDON_URLS_I18N_LAST for example.

django CMS sets settings['ADDON_URLS_I18N_LAST'] = 'cms.urls' - so in a project using django CMS, no other application can use ADDON_URLS_I18N_LAST.

4.3.6 Quick reference

Local commands cheatsheet

Project resource management

Set up a project divio project setup <slug>

Deploy Cloud server divio project deploy (applies to test server by default; optionally, specify live)

Update local project code: divio project update (pulls code, updates and builds local images, runs migrations)
Build local web image (e.g. after updating requirements.in or the Dockerfile) docker-compose
build web

Push/pull code Use git commands as appropriate.

Pull or push media or database divio project pull media or divio project pull db

These commands apply to the test server by default; optionally, you can specify live, e.g. divio
project pull media live).

Running the local server

Start a project divio project up, docker-compose up or docker-compose run --rm
--service-ports web

Stop a project divio project stop, or exit the command with Control-C.

Working inside the containerised environment

Run a specific command inside the web container docker-compose run --rm web <command>, for ex-
ample docker-compose run --rm web python manage.py shell

Run a specific command inside the web container, exposing the ports listed in the Dockerfile
docker-compose run --rm --service-ports web <command>

Docker management

List running containers docker ps
List all containers docker ps -a
List images docker image ls
Remove all stopped containers docker container prune
Remove all unused containers and images docker system prune

4.4 Background information

4.4.1 Our infrastructure

Divio platform overview

Divio is a cloud management platform, providing Docker-based containerised web application hosting.

See Docker basics for an introduction to Docker and its key components.

Divio offers a local development environment that replicates almost perfectly the Cloud environments in which appli-
cations run, eliminating many of the pain-points of deployment caused by having to deal with different environments
in development and production.

In our architecture, we abstract functionality from configuration so that functional components can be made immutable
and stateless wherever possible. This enables them to be replaced, added, moved and so on simply by spinning up new
instances, without requiring manual configuration.
Cloud infrastructure

Our cloud is vendor-neutral, and can be run on AWS, MS Azure and other infrastructures. Both our Control Panel and customer applications can be deployed on public, private and on-premise infrastructures.

Our Control Panel and the cloud management architecture are built on a Python/Django stack. Our client sites run in Docker containers. More information about our infrastructure can be provided on request.

Local development environment

Thanks to Docker containerisation, we are able to provide multiple environments for each project, including a local development environment that replicates the project’s cloud environments.

Our toolchain helps set up the local environment, transfer data and media files to and from the cloud, and manage deployments, from the command line.

Docker basics

Docker is a containerisation system. Containerisation is also known as operating-system-level virtualisation. It allows multiple independent containers to run on a single host. The containers are isolated from each other and from the host.

Resource isolation features make it possible for the containers to share underlying operating system resources. Whereas more traditional virtual machines replicate an entire operating system, containerisation can provide a much more lightweight solution to virtualisation, containing only the specific stack layers required for a particular application.

Docker containers are thus smaller and consume fewer resources, avoiding the memory and CPU overheads of full virtualisation. They are faster to start up, manage and scale, and easier to move, around than full virtual machines.

Docker on Macintosh and Windows

Docker requires a Linux host for its containers. On Linux systems, containers will simply use the Linux operating system’s resources. Macintosh and Windows need to run a single Linux virtual machine to serve as the host.

This can be done in two ways:

- **On newer systems**, the Alpine Linux host is provided through native operating system virtualisation.

  (On Macintosh, it’s provided by through HyperKit, a lightweight virtualisation system built on top of the Hypervisor framework (macOS 10.10 Yosemite and higher).

  On Windows, it’s provided through a similar system, Microsoft’s Hyper-V (Windows 10 Professional, Enterprise and Education editions).)

- **On older systems**, it requires a Linux virtual machine running in VirtualBox. This is managed by a tool called Docker Toolbox.

Key components

The two key components in Docker are:

- **Docker Engine**, the underlying daemon running on the host. Docker Engine is also confusingly sometimes referred to simply as Docker (to make things worse, there is also a command-line tool named docker).
• **Docker Compose** (invoked as `docker-compose`) is a tool for defining and managing multi-container applications; Divio projects use Docker Compose only in the local environment.

See [Docker commands](#) for the basics of `docker` and `docker-compose` usage.

### Glossary

**Application** Docker terminology uses “application” in much the same way that Django uses “project”, a collection of components that together form a complete and self-contained system.

In our case, a Docker application is the collection of components that is responsible for a website and its functionality, including everything from the database to the frontend code.

Docker applications are typically managed using **Docker Compose**, and configured in a `docker-compose.yml` file.

A Docker application will typically include multiple **containers**.

**Container** A Docker container is virtualised application environment. Unlike a virtual server, it doesn’t need to provide every layer in a full working system. Instead, it encapsulates only the layers required to run an application.

**Image** An image is a template. Each container is based on an image. Once an image has been created, each container created from it will provide exactly the same environment, and the applications in it will behave identically. An image is defined by its **Dockerfile**.

### IP addresses

Divio’s containerised infrastructure is distributed over a number of geographical and vendor based regions. At any one time, the servers within each region are allocated an IP address from that region’s range.

Servers in our infrastructure (application builders and runners, and other services) are not permanent hosts but short-lived instances that can be provisioned at a moment’s notice and are regularly recycled.

An IP address within a particular pool can therefore be reallocated to a different server at any time, and there is no guarantee that a server’s address will remain the same from one moment to another.

In addition, vendors’ IP ranges themselves are not guaranteed and can be subject to change regularly and frequently, sometimes without notice.

Finally, even if a server’s IP remains the same, an application might be moved to a different server at any time.

This means that **IP addresses are not to be relied upon as a means of reaching or identifying Divio applications or servers**.

### Load-balancer IP addresses

A partial exception to this is the IP addresses of our load-balancers, which can be regarded as semi-static.

Customers’ projects may be attached to domains for which CNAMES cannot used (that is, bare domains such as `example.com`, rather than sub-domains such as `www.example.com`). In this case we recommend using ALIAS records. Not all DNS providers support ALIAS, and in those cases it is necessary to use A records, which require IP addresses.

We take efforts to keep our load-balancers in each region on a very small number of IP addresses so that A records can rely on them. All the same, even these occasionally must be amended. In that event affected customers are informed in advance to minimise disruption.
Implications for customer projects

Occasionally a fixed IP address for a customer’s project might be desired for:

- incoming connections, such as a client connection to an API running in the project
- outgoing connections, such as a connection to an external API, where the IP address is required for whitelisting purposes

For in-bound connections, our semi-static load-balancer IP addresses can be an adequate way of routing connections to a customer application, as long as the user remains aware that these addresses may need to be changed in the future.

Out-bound connections present more challenge. Cloud-based hosting is very well established, and very few services still insist upon using IP addresses as a way of guaranteeing the originator of a connection. However, IP addresses are widely used in firewall rules, and this can still present an issue (see below for other options).

In general though, an alternative, better of achieving a connection to or from Divio applications can and should be found.

Other options

In some cases, it may be strictly necessary for an application running on Divio to present an unvarying IP address or range of addresses to an external service.

In such cases, for projects running in private regions only, a number of solutions can be implemented:

- a dedicated proxy to redirect traffic from a fixed IP address to the application
- a NAT gateway with a fixed external IP address
- a dedicated site-to-site VPN

Please contact Divio support to discuss these options.

IPv6 support

We do not currently support IPv6 on our infrastructure, as IPv6 is not yet supported in key components provided by our vendors. Future IPv6 support will be introduced when possible.

4.4.2 Services

Service management (Beta)

Note: The Services view is currently provided as a Beta feature, and is available only to users who have signed up for access to Beta-release features.

As well as its application code, a Divio project can include various services that are provided independently, such as a database, media storage, a message queue and so on. These can be added, removed and configured in the Services view of any project.
Projects include a Postgres database and S3 media object storage by default; additional services such as alternative databases, Elasticsearch and so on are also available.

See Available services for an outline of services currently provided.

Multiple instances of a service - for example, two Postgres databases - may be used at the same time.

Service management via the Control Panel

Before services can be configured for a project, the project’s subscription must be updated to include them. For example, to add Elasticsearch, you would add the requisite number of Elasticsearch instances via the Subscription view. Once added, those services will be available to add in the Services view.

Adding and attaching

Making a service available to an application is a two-stage process:

First, the service must be added to each environment that requires it. A unique prefix should be provided in case other instances of the same service have already been applied.

Next, the environment must be deployed. Deployment provisions the service, and attaches it to the application.

If required, the option exists to provision a service independently of attachment. In this state, the application has not yet been deployed with the environment variables it needs to use the service, but the service itself is functional and usable. In the case of a media storage or database service, for example, this would allow you to populate it in advance of the application’s next deployment.

Detaching and removing

A service may be detached if it is no longer to be used by the application. Like attachment, this requires a deployment to take effect.

If a service is no longer required, it can be removed. An attached service will need to be detached before it can be removed.

Removing a service is a destructive operation. It will permanently delete any data used by that service instance.

Detaching a service is non-destructive. However if the application depends on the service, detaching it may cause a deployment or runtime error.

In the case of a deployment error following a detachment command, the service will not be detached, and the application will continue running in its previous configuration. This safeguards a running application.
States

Services will exist in a number of states across their lifetime:

<table>
<thead>
<tr>
<th>new</th>
<th>provisioning</th>
<th>pending attachment</th>
<th>attached</th>
<th>pending detachment</th>
<th>detached</th>
<th>removed</th>
</tr>
</thead>
<tbody>
<tr>
<td>not functional</td>
<td>functional</td>
<td>usable by the application</td>
<td>functional</td>
<td>not functional</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Available services

See also:

*Service management (Beta)*

Database

We provide Postgres and MySQL databases by default; other database systems can be provided on request. Postgres is our database of choice, and configured by default with all projects. Databases can use public (shared) or private clusters in the same region as the web application.

Media object storage

Default file storage in Divio projects is handled by dedicated storage and hosting providers. Depending on the project’s region, these can be S3 providers such as Amazon Web Services’s S3 service or a generic S3 hosting service via another provider, or MS Azure Blob storage. By default, media files are served by a Content Delivery Network in order to provide better performance.

See also:

- *Working with your project’s media storage in Python applications*
- *How to interact with your project’s cloud media storage*

Elasticsearch

Elasticsearch is provided as our default search engine, running on public (shared) or private clusters in the same region as the web application. We support multiple versions of Elasticsearch.

RabbitMQ

We provide RabbitMQ for messaging.
4.4.3 How it works on Divio

HTTP redirects

Redirects in Divio projects will typically take place at one of two different levels:

- in the application code itself (for example, in Django or PHP)
- in the application’s web gateway or server (for example, in uWSGI or Nginx)

An application can blur the distinction between these two levels, but most will include distinct code and gateway/server layers. Gateway/server-level redirects take place before a request reaches the application code (and will therefore be faster and less expensive). However, code-level redirects are generally easier to implement.

Protocol redirects

Our projects are HTTPS-ready by default, and we provide free SSL certificates even on free projects.

To force redirect from HTTP to HTTPS: Django, Express JS.

301 Permanent vs 302 Temporary redirects

You will sometimes see online site-checking tools encouraging the use of permanent redirects and even flagging temporary redirects as an issue. It is true that a permanent redirect is sometimes more appropriate, but only when it really should be permanent, and is guaranteed not to change.

Protocol, domain and language directs are 302 Temporary by default. 301 Permanent redirects are cached by browsers - some even update their bookmarks if they encounter a 301. This can cause problems if the redirects change, potentially causing redirect loops for users (which site owners will not be able to replicate).

Caching and CDN in Divio projects

Infrastructure-level caching

As well as the caching and CDN (Content Delivery Network) we provide, you can also apply your own. Caching is generally provided by Cloudflare.

Caching for media storage is available on all projects and plans, and on Test as well as Live servers. It’s automatic and requires no additional action or configuration. Note that this only includes files served from our S3 media buckets (typically, images). Resources served by the application instances (HTML, static files) are not cached.

Caching is URL-based, and wholly content-unaware - caching does not detect when files have changed.

Options

However, other options for caching are available on request (on eligible projects only). This includes:

- caching of application content as well as media files
- Cloudflare Polish and Mirage optimisation for images
- cache invalidation control via the Divio project Dashboard
- custom Cloudflare page rules
Media file caching

Typically, the bulk of a page web’s transfer load is accounted for by its media files, most of which will be images in the page.

All media files are handled by our dedicated storage and hosting providers, using S3 buckets. If your application uses the aldryn-media.io domain for S3 buckets, then the files are not cached.

Delivery of these files is handled by Cloudflare’s CDN, which also caches the files.

Controlling caching headers

Cloudflare will cache media files according to the Cache-Control header applied to the files. For example:

```text
cache-control: max-age=3600
```

will set a TTL of one hour (3600 seconds).

Your application can set these headers when managing the file storage. Aldryn Django does this by default, also applying some sensible default values (see also DISABLE_S3_MEDIA_HEADERS_UPDATE).

Our Aldryn Django Filer addon applies a one-year TTL to its public thumbnail files using the MEDIA_HEADERS setting. In turn, our Aldryn Django addon applies the MEDIA_HEADERS values it discovers to the media storage class that configures the S3 bucket.

Any application that needs to control the behaviour of cached media will need either to make use of provided functionality (for example, such as in Aldryn Django), or configure the S3 bucket directly itself.

Applying your own caching/CDN

There is nothing to stop you applying your own caching and CDN.

For media files, if you are using our media domain (the default), your projects will automatically use the CDN we provide and this cannot be changed. However, it’s also possible to use your own domain (see for example How to configure media serving on a custom domain) for media, in which case you are free to use what you wish.

For the rest of the project, you can set up another CDN, for example using your own Cloudflare account. In such a case you should inform us so that instead of providing a certificate automatically ourselves, you can upload your own manually.

Application-level caching

Django applications

Caching in Divio Django applications will typically make use of Django’s own caching framework.

Caching in django CMS

The Aldryn django CMS addon applies caching rules by default, via the CMS_CACHE_DURATIONS setting.

Control over caching settings is exposed in the Divio Control Panel in the configuration options for Aldryn django CMS. Defaults are to cache content for 60 seconds and menus for one hour.

It is often convenient to disable caching while developing or working intensively on content. Adding:
import os
env = os.getenv
STAGE = env('STAGE', 'local').lower()
if STAGE in {'local', 'test'}:
    CMS_PAGE_CACHE = False
    CMS_PLACEHOLDER_CACHE = False
    CMS_CACHE_DURATIONS = {
        'menus': 0,
        'content': 0,
        'permissions': 0,
    }

to the project’s settings.py will disable all caching in the CMS in the local and Test environments.

**Application caching options**

**What not to use in your code**

Caching should rely on a shared store that persists for all containers. For example, caching that relies on a container’s local file-system or local memory should not be used, as only that container (and not a container running in parallel, or one instantiated later) will be able to access the items it stores.

In some cases, this can simply lead to inefficiency (not using cached data). In other cases, it could cause malfunction or even data-loss, if two instances are working with inconsistent data.

**Database caching**

Database caching is shared by all instances of an application server, making database caching suitable for many use-cases.

It’s our default cache backend for Django projects - all Divio Django projects are set up with Django’s database caching configured and ready to use.

This is a fast, scalable option, and is suited to most needs.

**Third-party caching backends**

Other backends, such as Redis (a popular open-source database) can be used as caching backends for Django.

If it suits your needs, you can procure a Redis or other caching instance from a provider and use it with your Divio project.

**Python package installation**

Python packages in Divio projects are installed using pip. Our Dockerfile for Python projects performs three operations in sequence: it *compiles a list of dependencies*, *creates a list of wheels*, and *installs the wheels*, using the following commands:

```
RUN pip-reqs compile && # compile a list of dependencies
    pip-reqs resolve && # create a list of wheels
    pip install \ # install the wheels
```

(continues on next page)
Selective execution of commands

By removing or commenting out the `pip-reqs compile` instruction after a successful compilation of all requirements, you will prevent successive builds from reprocessing `requirements.in`, pinning all the dependencies to exact versions. This can be valuable, as upstream dependencies of dependencies may change at any time, and can cause a failure on successive deployment - even if you haven’t changed anything in the project yourself. See *How to pin all of your project’s Python dependencies* for more.

To isolate the behaviour and effects of any of these three commands, they can be run individually, for example:

```
docker-compose run web pip-reqs compile
```

Alternatively, if you’re not able to run commands in a container, comment out the ones you don’t want to execute, and try building with `docker-compose build`.

Summary of commands

<table>
<thead>
<tr>
<th>command</th>
<th><code>pip-reqs compile</code></th>
<th><code>pip-reqs resolve</code></th>
<th><code>pip install</code></th>
</tr>
</thead>
<tbody>
<tr>
<td>input</td>
<td><code>requirements.in</code></td>
<td><code>requirements.txt</code></td>
<td><code>requirements.urls</code></td>
</tr>
<tr>
<td>action</td>
<td>creates a complete dependency list</td>
<td>creates a list of wheels</td>
<td>installs the wheels</td>
</tr>
<tr>
<td>output</td>
<td><code>requirements.txt</code></td>
<td><code>requirements.urls</code></td>
<td></td>
</tr>
<tr>
<td>fails if</td>
<td>a dependency cannot be found</td>
<td>a wheel cannot be found</td>
<td>pip is unable to install the wheel</td>
</tr>
<tr>
<td>typical cause</td>
<td>dependency conflict or no longer available</td>
<td>the wheels proxy was unable to build a wheel</td>
<td>the wheel requires additional components for installation</td>
</tr>
</tbody>
</table>

Our wheels proxy

We maintain our own DevPi server for private packages, and a Python wheels proxy.

Our wheels proxy creates wheels for all packages on PyPI.

A Python wheel is a pre-compiled package, built for a particular platform (a combination of the target operating system, architecture and Python version). Wheels offer numerous advantages over other Python packaging options. On our platform, they reduce installation times significantly, both locally and on our infrastructure. Using wheels also allows us to perform additional dependency resolution during installation.

Typical issues when installing wheels

You will occasionally see an error in a deployment log that clearly refers to installation of Python packages, occurring after the output:
Step 7/9: RUN pip-reqs compile &&
    pip-reqs resolve &&
    pip install --no-index --no-deps --requirement requirements.urls

This indicates that one of those commands has failed, usually in one of the following ways:

**A dependency cannot be found (from pip-reqs compile)**

Sometimes a dependency cannot be found. This could be because a version has been specified incorrectly, or no longer exists:

```
Could not find a version that matches django==1.11.29,>2.0
```

Most commonly, it’s because different packages in the same project either explicitly or implicitly specify conflicting versions of a dependency (for example, django==1.11.29 and django>2.0 as above) at the same time. See *How to identify and resolve a dependency conflict* for more on this.

**A wheel cannot be built (from pip-reqs resolve)**

Occasionally, a wheel cannot be built for a particular package. This is usually because although the package exists on PyPI, it is not compatible with the particular version of Python specified for that wheel (an example might be a Python 2 package in a Python 3 environment) and the attempt to build the wheel fails.

The error will appear in the logs as:

```
HTTPError: 500 Server Error: Internal Server Error
```

from the wheels server. In such a case, check that the dependency mentioned is in fact suitable for the environment.

**A wheel cannot be installed (from pip install)**

Sometimes, a wheel can be found and downloaded, but fails to install. Example output (in this case for jupyter) might be:

```
Installing build dependencies: started
Installing build dependencies: finished with status 'error'
ERROR: Collecting setuptools
ERROR: Could not find a version that satisfies the requirement setuptools (from versions: none)
ERROR: No matching distribution found for setuptools
```

In this case, the wheel was found and downloaded, but could not be installed because it contained a “hidden” dependency (setuptools). One option is to contact Divio support; we can ensure that the wheel is built with this requirement.

Another is to bypass the wheels proxy, described in *Not using the wheels proxy* below.

4.4. Background information
Not using the wheels proxy

You don’t have to use our wheels proxy, though it’s the default and it’s strongly recommended. To change the way Python installs packages, see the Dockerfile.

Bypassing the proxy for a particular dependency

First, you would remove the dependency from the requirements.in so that it is not processed by the default installation commands; then, it would be added to the Dockerfile just before them, for example:

```
RUN pip install jupyter==1.0.0
```

Bypassing the proxy altogether

This is not recommended, but the default installation commands can be replaced with:

```
RUN pip install --requirement requirements.in
```

Our standard Dockerfile for Python/Django projects contains:

```
ENV PIP_INDEX_URL=${PIP_INDEX_URL:-https://wheels.aldryn.net/v1/aldryn-extras+pypi/$>({WHEELS_PLATFORM:-aldryn-baseproject-py3})/+simple/} \
    WHEELSPROXY_URL=${WHEELSPROXY_URL:-https://wheels.aldryn.net/v1/aldryn- \
        extras+pypi/}{WHEELS_PLATFORM:-aldryn-baseproject-py3}/
```

Removing this will use PyPI instead of our own PyPI server.

Caching

See Docker image/layer caching and re-use for the implications of caching for package installation with pip.

Environment variables

Divio projects allow you to maintain separate configuration for each of the Live, Test and Local environments.

Environment variables are dynamic values that can be used by the processes or applications running on the server. One of the advantages in using them is that you can isolate specific values from your codebase.

Environment variables are a good place for storing instance-specific configuration, such as settings that you don’t wish to hard-code into your project.

Cloud (Live and Test) environments

On Cloud sites, environment variables for a project are managed via the command line, or via the Control Panel. The variables for the Live and Test sites are wholly independent.
Via the command line

The *Divio CLI* allows you to set and check values from the command line with the `divio project env-vars` command, as long as you are within the path of the local version of the project. For example, to see the variables of the *Live* server:

```
divio project env-vars -s live
```

If any exist, they will be displayed thus:

<table>
<thead>
<tr>
<th>Key</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>SECURE_SSL_REDIRECT</td>
<td>True</td>
</tr>
</tbody>
</table>

See the *divio project env-vars* reference for more.

Via the Control Panel

In the project, select *Environment Variables*. Enter the keys and values, and *Save*.

Local environment

Your local site also uses environment variables, contained in the `.env-local` file.

By default these are:

```
DEBUG=True
STAGE=local
DATABASE_URL=postgres://postgres:5432/db
```
Formatting

Lines should not contain spaces or quotation marks (see Docker’s documentation).

Environment variables and Django settings

As you can see from the local environment examples above, environment variables can also be used to apply Django settings, such as `DEBUG`.

To access the environment variable in your own Python code, you could use something like this:

```python
import os
my_variable = os.environ.get('MY_ENVIRONMENT_VARIABLE')
```

It’s important to note that if your variable represents anything other than a string, you will need to interpret the variable appropriately, as `os.environ.get` will only return a string.

You can also use `env()` (from the `getenv` package), which will parse the variable as Python code.

Where and when environment variables are applied

Environment variables should apply only to environments, and not to states or processes that are independent of a particular environment.

- **When a project is running**, it runs in a particular environment, so you can expect environment variables to apply.
- **When a project is being built** (i.e. in the deployment phase), it should not be subject to any particular environment conditions. Even Django operations that take place during deployment (such as collectstatic) should be environment-agnostic - under all environment conditions, you should expect the same result from collectstatic.

  However you can set environment variables during the build phase.

Commonly-used environment variables

Many of the applications packaged for Divio deployment recognise a number of environment variables for your convenience.

See Addons for lists of settings that can be provided as variables in some Divio addons.

They do this in their Addon configuration with aldryn_config.py files. To see precisely how they are handled, refer to the aldryn_config.py file of key addons (important: make sure you are looking at the correct version of the addon, as different versions of the packages will assume different variables and settings):

Our uWSGI application gateway also recognises environment variables that commence UWSGI_.

4.4.4 Projects

Project creation options

Each Divio project is based on a combination of Platform, Type and Boilerplate. You can select from our optional ready-to-go definitions for a quick start. For example:
• **Platform**: Python
• **Type**: django CMS
• **Boilerplate**: HTML5

Alternatively, you can build your own from scratch, by choosing the *No platform* option.

It’s important to understand that these pre-built project templates are simply there to provide you with a quick way to get started with a particular stack. They don’t prevent you from adding other components; for example, you might decide to *add Sass CSS compilation using Node to a project that doesn’t already include Node*.

**Platform**

Options include Python, Node, PHP, Java and others.

If you prefer to build your own Dockerfile and project components from scratch, select *No platform*. We recommend becoming familiar with Divio projects before doing this.

**Type**

The available project types depend on the selected Platform.

For the Python platform, examples include Django, django CMS and Flask; for the PHP platform, Laravel and Symfony, and so on.

**Boilerplate**

Available Boilerplates depend on the selected project Type.

Boilerplates provide additional functionality baked into the project, typically related its frontend configuration. For example, a django CMS project Type can be launched with Boilerplates for Bootstrap, Foundation and other technologies.

**Git repository manager**

By default, your project will use Divio’s own private Git server. Alternatively you can select a Git provider of your choice. You can *migrate a project from our Git server to an external provider at any time*.

For a quick start, use Divio’s Git server.

**Anatomy of a Divio project**

A Divio project is contained in a Git repository. The files in the repository are used to build the project image, from which its containers are instantiated.

**Dockerfile**

The key file is the *Dockerfile* that defines the project.
Local development files

Some files are only present, or only used, when in the local development environment.

.aldryn

Added to the local project by the `divio project setup` command, to provide an identifier for the corresponding cloud project. Not part of the repository.

.env-local

On the cloud, environment variables can be set via the Control Panel. Locally, they can be supplied in `.env-local`. This is part of the repository, so it is recommended not to commit sensitive configuration values.

docker-compose.yml

The `docker-compose.yml` file describes the Docker configuration of the local environment, and the services compose the whole Docker application. It’s part of the repository, but is ignored on the cloud.

data

For local development use only. In `data`, a `media` directory functions as the local analogue of Cloud project’s S3 storage bucket (see *How to interact with your project’s cloud media storage*).

Aldryn Addons directories

Not all projects use the Aldryn Addons system. Those that do will contain:

addons

For each addon in your project, a directory will be created in `addons`, containing:

- `addon.json`: basic metadata for the addon (generally, there is no need ever to edit this file)
- `aldryn_config.py`: optional; manages settings for the addon (see *aldryn_config.py explanation* and *how to create an aldryn_config.py file*)
- `settings.json`: any settings that were applied via the Control Panel, so that they can be used locally

addons-dev

For local development use only. An addon can be placed here for development purposes. `addons-dev` contains a little magic; any packages within directories in `addons-dev` will automatically be placed on the Python path for convenience.

Running `divio project develop <addon>` for an addon in `addons-dev` will add it to the project’s `requirements.in` and `settings.INSTALLED_ADDONS`, then attempt to build the project.
The project deployment process

Deployment steps

1. The Control Panel checks that required services (such as the database) are available.
2. The project’s Git repository is checked out.
3. An image is built, using the instructions in the Dockerfile.
4. A container is deployed from the image.
5. Any migration commands (post-build instructions) defined by the project are executed.
6. Additional containers are deployed according to the project’s configuration.
7. The Control Panel tests that the application is responding.

Zero-downtime cloud deployments

If all of the steps above are successful, then the deployment is marked as successful, and requests will be routed to the new containers, and the old containers will be shut down. They are never shut down until the new containers are able to respond to requests without errors. This allows us to provide zero-downtime deployments - in the event of a deployment failure, the old containers will simply continue running without interruption.

Differences between cloud deployment and local builds

- **Orchestration**: on the cloud the Control Panel manages orchestration; locally, it’s handled by docker-compose according to the `docker-compose.yml` file.
- **Services**: on the cloud, backing services such as the database and media storage - and if appropriate, optional services such as a message queue - are provided from our cloud infrastructure. Locally, these must be handled differently (your computer doesn’t contain a Postgres cluster or S3 bucket): the database will be provided in a separate Docker container, the media storage will be handled via local file storage, and so on. docker-compose will configure this local functionality.
- **Docker layer caching** on the cloud we don’t cache, locally we do.
- **Migration commands**: locally these are not executed by default; execute them with `docker-compose run --rm web start migrate`.

Notes on Docker image building

**Docker image/layer caching and re-use**

Images and image layers are:

- not cached in cloud deployments
- cached by default in local builds
Cloud deployments

We don’t use Docker-level layer caching on the cloud because certain cases could produce unexpected results:

- Unpinned installation commands might install cached versions of software, even where the user expects a newer version.
- Commands such as `apt-get upgrade` in a Dockerfile could similarly fail to pick up new changes.
- Our clustered setup means that builds take place on different hosts. As Docker layer caching is local to each host, this could mean that subsequent builds use different versions, depending on what is in each host’s cache.

When an image is built, even if nothing in the repository has changed, the image may be different from the previously-built image. Typically, this can affect project dependencies. If a project’s build instructions specify a component, the installer (which could be `apt`, `pip` or `npm`) will typically try to install the latest version of the component, unless a particular version is selected.

This means that if a new version has been released, the next deployment will use that - without warning, and with possibly unexpected results. It is therefore strongly recommended to pin package versions in your project’s installation lists wherever possible to prevent this. (See also *How to pin all of your project’s Python dependencies.*)

Image re-use on the cloud

In some circumstances, the build process will *not* build a new image:

- If there are no new commits in the repository, and an image has been built already for the *Test* server, that image will be re-used for the *Live* server.
- When deploying a mirror project, the image already created for the original will be re-used.

Local builds

Locally, Docker *will* cache layers by default.

Local image caching can affect components that are subject to regular updates, such as Python packages installed with `pip`. In this case, a new version of a component may have been released, but the local build will continue to use an older version.

To turn off this behaviour, use the `--no-cache` option with `docker-compose build`.

Project environments

Each Divio project includes three environments for the application by default.

The three environments are created in Docker containers from the same images.

- *Local*, running on your own computer
- *Test*, running on our Cloud servers
- *Live*, running on our Cloud servers

In our recommended workflow, development is done locally, before being deployed to *Test* and finally to *Live*.

It is also possible to set up multiple cloud environments for a project. For example, a common scheme is to use four: *Development, Testing, QA, Production.*
Default project conditions

Some of these conditions may be readily altered according your needs, for example the `DEBUG` setting. See also *How to run a local project in live configuration*.

<table>
<thead>
<tr>
<th></th>
<th>Local</th>
<th>Test</th>
<th>Live</th>
</tr>
</thead>
<tbody>
<tr>
<td>STAGE environment variable</td>
<td>local</td>
<td>test</td>
<td>live</td>
</tr>
<tr>
<td>DEBUG environment variable</td>
<td>True</td>
<td>True</td>
<td>False</td>
</tr>
<tr>
<td>static files served by</td>
<td>local server (e.g. Python runserver)</td>
<td>uWSGI</td>
<td></td>
</tr>
<tr>
<td>media files served by</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>database runs in</td>
<td>a local container</td>
<td>our Cloud database cluster</td>
<td></td>
</tr>
<tr>
<td>number of application containers</td>
<td>one according to subscription</td>
<td></td>
<td></td>
</tr>
<tr>
<td>application container sleeps</td>
<td>n/a</td>
<td>after 15 minutes’ activity</td>
<td>never</td>
</tr>
</tbody>
</table>

Version control

See also:

- *How to configure a Git remote for your project*
- *How to use Git to with a Divio project*

Divio projects use Git for version control.

By default, projects use our own Git server.

We also offer the option to manage your project’s codebase on remote Git providers. Currently, we support GitHub, GitLab and BitBucket (other options may also work or can be available on request, including private Git servers, for suitable projects).

Our Control Panel interacts with different Git providers via an abstraction layer that makes it possible to present common user and application interfaces.

All commits target the Test environment

All commits made by our Control Panel are to the branch used by the Test environment. For example, the Addons view in the Control Panel displays (and only affects) the configuration of the Test environment. When you use *Custom Tracking Branches* (below), the Live environment configuration is untouched except by Git operations that explicitly target its branch.

Project repository branches

By default, each project’s code is in its Git `master` branch, and can be deployed directly from the Git server to the Test or Live servers (our strongly-recommended workflow is always to deploy to Test first).

Custom Tracking Branches

Each environment can be configured to track a different Git branch, by editing the `Branch` field in the Environments view. If the branch specified cannot be found, an *Unable to get commit count from repository* message will be shown.
Using custom branches allows (for example) a workflow in which you work on develop before manually merging into master, and then deploying Live.

**Boilerplates**

A **Boilerplate** is a convenient way to define components to be used by a project, so that new projects can be created quickly with the same tooling set up as soon as the project is created.

A Boilerplate can be used to define any project-level components and processes that are to be re-used (note that Python components will be handled by our addon system and Pip, not by Boilerplates) but will typically take care of a project’s frontend set-up.

For example, a Boilerplate will define how the Django templates are structured and make opinionated choices about what JavaScript frameworks and CSS tools are used.

When a project is created, it will always specify a Boilerplate - even if it’s a pre-defined **Blank Boilerplate** that leaves all these choices up to you.

Various Boilerplates are provided as defaults. Some provide only basic HTML and CSS, whereas more sophisticated ones include advanced frontend tooling: NPM, webpack, Sass and other components. It it’s also possible to define and reuse your own.

**Our built-in Boilerplates**

This list represents some key built-in Boilerplates, but others are added regularly.

**Blank Boilerplate**

The **Blank Boilerplate** installs no components. It will be up to you to install anything you need, including templates for your site.

**HTML5 Boilerplate**

Our **HTML5 Boilerplate** implements the HTML5 Boilerplate package. The HTML5 Boilerplate package is a popular starter set of starter files, which includes a generic HTML template, CSS to normalise and set some standard classes, and some JavaScript including jQuery.

Our implementation of it is very standard, and simply adapts it for use in django CMS projects.

**Bootstrap and Foundation Boilerplates**

Our **Bootstrap** and **Foundation** Boilerplates implement these two popular frontend frameworks.

They are both fully-featured frameworks that include opinionated CSS and JavaScript for your own use, and numerous built-in widgets and standardised web components.
Sass Boilerplate

Our Sass Boilerplate introduces compiled components. This is a dynamic Boilerplate, unlike those above, which is to say that it compiles its own materials at runtime (the static Boilerplates by contrast simply use or serve the materials they ship with).

The Sass Boilerplate uses the HTML5 Boilerplate package as its basis. It uses:

- Gulp to run the compilation
- npm as a package manager
- Node as a run-time environment.

The advantage of using a dynamic Boilerplate with compiled components is that it permits you to build a customised frontend, shorn of items your project does not require. Compiled components can also be heavily compressed and optimised, while the source files you work on can remain readable and comprehensive.

Webpack Boilerplate

The Webpack Boilerplate implements the Bootstrap framework as a fully-compiled frontend set-up. It uses:

- Gulp to run the compilation
- Webpack to bundle all the modules
- npm as a package manager
- Node as a run-time environment.

How Boilerplates work

When you create a new project via the Divio Control Panel, you select a Boilerplate at the same time.

The Control Panel examines the Boilerplate to determine what components should then be installed.

It does this by checking:

- The boilerplate.json file (required in all Boilerplates). If templates are specified here, they will be applied to the django CMS configuration as available templates.
- The Boilerplate’s Dockerfile. Sections in the Dockerfile will be copied to the project’s Dockerfile appropriately; for example:

```bash
# <NODE>
ADD build /stack/boilerplate

ENV NODE_VERSION=6.10.1 \
    NPM_VERSION=3.10.10

RUN bash /stack/boilerplate/install.sh

ENV NODE_PATH=$NVM_DIR/versions/node/v$NODE_VERSION/lib/node_modules \
    PATH=$NVM_DIR/versions/node/v$NODE_VERSION/bin:$PATH
# </NODE>
```

will be copied to the:
section.

The Control Panel will then copy all files (other than the two mentioned above) and directories in the Boilerplate into the project, unless they are explicitly excluded in the \texttt{boilerplate.json} file's \texttt{excluded} list.

The Control Panel will also detect and respond automatically to the presence of various other files in the Boilerplate. These files are:

- \texttt{package.json} - will be used by \texttt{npm} to install node packages
- \texttt{bower.json} and \texttt{.bowerrc} - will be used by Bower to install frontend components. Note that this is provided as legacy support. We no longer recommend Bower (use \texttt{npm} instead)
- \texttt{gulpfile.js} - used by Gulp execute specified compilation tasks

The Control Panel will write appropriate commands into the Dockerfile, so that when the project is next deployed, the appropriate components will be installed and tasks run.

Bower commands will be placed inside the \# \texttt{<BOWER>}/\# \texttt{</BOWER>} section of the Dockerfile, and so on.

\textbf{Note:} These operations are performed by the Control Panel only. They will not be performed in the local environment, but only on the \textit{Test} and \textit{Live} servers.

If you make changes to these files locally, in order to see the effect you will need to:

- push your changes to the Cloud, where they will be processed into the Dockerfile
- pull down the project again

\textbf{Multi-site projects on Divio}

Multi-site hosting, also known as \textit{multi-tenancy}, makes it possible to host multiple sites, serving multiple domains, using a single database.

\textbf{Note:} This is distinct from serving a \textit{single site} under \textit{multiple domains} - this does not require django-multisite, and only requires those domains to be set up in the Control Panel.

We support two multi-site options:

- using our Mirrors feature, with multiple projects running the same codebase and using the same content (recommended)
- (for Django sites only) a DIY implementation via the \texttt{django CMS Multisite} package

\textbf{Using Mirrors}

Multiple “mirror” projects can be launched as duplicates of an original. They will all share exactly the same codebase. By default, each mirror is an independent project, with its own database and media storage. In a multi-site arrangement, all the mirrors will share a single database and media storage.

Commits to the base project codebase can be rolled out automatically or manually across all the mirrors.
Advantages

This arrangement has numerous benefits. It brings better sub-site isolation; a problem with one site’s performance will not affect other sites. It also offers better permission management, and better scalability of many small applications across multiple hosts.

Per-site resources can be fine-tuned more effectively (though overall resource usage may be slightly higher).

Caveats

To achieve zero downtime deployments a good policy of maintaining backwards-compatible database migrations is required, as the first project in the mirrored groups to be deployed with the fresh code will apply any database migrations; these will affect all projects. They therefore need to be backwards-compatible, so that others will continue to work with the new database schema. Subsequent code updates, applied once all projects in the group have been deployed, will no longer need to support the old schema.

DIY implementation (for Django sites only)

This option is still available, but is no longer recommended.

You will need to install django CMS Multisite package and manage any settings and configuration manually.

We are able to assist by setting up your Test server domains, but other than that we are not able to provide support for this option.

Logging in to multi-site instances

In all multi-site projects, you will notice that each site requires its own log-in. Logging in to one will not log you in to another. This is because the Django session cookie is per-domain.

Switching between sites in a local multi-site project

Suppose you have a multi-site project example-stage.eu.aldryn.io whose domains are configured thus:

<table>
<thead>
<tr>
<th>Site</th>
<th>Live</th>
<th>Test</th>
</tr>
</thead>
<tbody>
<tr>
<td>Germany</td>
<td>example.de</td>
<td>de.example-stage.eu.aldryn.io</td>
</tr>
<tr>
<td>Ghana</td>
<td>example.gh</td>
<td>gh.example-stage.eu.aldryn.io</td>
</tr>
<tr>
<td>USA</td>
<td>example.us</td>
<td>us.example-stage.eu.aldryn.io</td>
</tr>
</tbody>
</table>

On both the Test and Live Cloud servers, domain routing will be configured automatically.

Locally, more work is required, as each of the sites will be served at the same address (i.e. localhost). Unlike the Cloud set-up, your local environment has no way to route different requests to different sites - because the request for localhost doesn’t contain the domain information needed to do this.

Instead, you will need to handle multi-site routing manually.

There are two approaches you can take:
Option one: Force the site with an environment variable

This is the simpler method and recommended unless you need to be able to switch often between sites.

Set a local environment variable, SITE_ID, to force serving of a particular site. The site ID you need can be found in the list of Sites in the admin.

After changing the SITE_ID you may need to restart the local server for the change to take effect.

If you set explicitly set the SITE_ID this way, the next method will not work.

Option two: capture the sites’ domains in /etc/hosts

The other method is to point all the domains you need to handle locally to localhost. For example, for each of the domains in the table above, you could add an entry in your /etc/hosts file:

```
localhost example.de
localhost example.gh
localhost example.us
```

These should match the domains as they appear in the admin at Sites > Sites.

Now, each domain will resolve to localhost.

It will help to have your local site served on port 80, so in the docker-compose file, set:

```
ports:
  - "80:80"
```

for the web service.

The local Divio environment server running will now be able to use the information contained in the request for say example.de or example.gh to serve the required site.

Using this method, you can readily switch between sites using the standard site-switching functionality. The site-switching links will refer to the various Live sites, but these addresses will be intercepted by the modified hosts file.

---

**Note:** If the port on which you access local sites is not 80, you will need to append this to the addresses you require.

This method is more convenient if your local development work requires you to switch sites, but you must remember to remove the entries from hosts once you’ve finished.

Note also that if you force the site using the environment variable method, then this method will not work.

### 4.4.5 Django addons

**What are addons, and why use them?**

The *Divio Django addons framework* allows packages to be installed and configured very easily and in some cases fully automatically.
What are Divio addons?

Addons can be thought of as wrappers for Python packages such as Django or django CMS that allow them to take advantage of our addons framework. You’ll see for example addons such as Aldryn Django (a wrapper/installer for Django) and Aldryn Django CMS (a wrapper/installer for django CMS).

Addons are provided as a convenience. If you’d like Django in your project, with all its settings configured for our infrastructure (uWSGI gateway, database, media storage etc), and in such a way that they will work correctly in the Live, Test and local environments, then Aldryn Django will take care of that for you; if it’s installed, all that configuration and wiring will be done automatically.

In the case of Aldryn Django CMS, it will configure settings such as MIDDLEWARE and TEMPLATES automatically. Packages as installed by Divio addons (such as Django or django CMS) are completely standard and unmodified.

Installation and basic configuration of addons is managed via the project’s dashboard in the Control Panel. More advanced configuration can be managed via settings and environment variables.

Using addons is optional

Using the Addons framework on Divio is completely optional.

You don’t need Aldryn Django to install or use Django. Similarly, django CMS can be installed and configured manually, without using Aldryn Django CMS, if you prefer, and the same goes for others.

Why use addons?

Software installed without taking advantage of the addons framework won’t make use of our convenience layers, and it will require a developer to install and configure them, whereas software packaged as an addon can be installed and configured with a few clicks, without any technical knowledge.

Using our addons will make your work as a developer faster and easier, allowing you to concentrate on development instead of configuration.

The addons system exposes packages, along with their versions and configuration options, in our Control Panel - you can apply settings and manage upgrades with our GUI, not even needing to set the repository up locally to edit its code or requirements. This can save a great deal of time, especially when a new version of a package requires different configuration in settings.py.

The Control Panel will alert you when updates are available, or in the case of critical security fixes.

The vast majority of Divio developers users prefer to make use of the framework and the wrapper applications, though some prefer to undertake configuration manually and choose not to use it.

See also:

- How settings are handled in Django addons
- Addon configuration with aldryn_config.py
- The settings.py file

Anatomy of a Divio addon

Basic file structure

For an addon “Susan Example Application”:
addons-dev/
  susan-example-application/
    addon.json
    LICENSE
    MANIFEST.in
    README.rst
    setup.py
    susan_example_application/
      __init__.py

aldryn_config.py

All addons have an `aldryn_config.py` file that takes care of settings, which are then loaded into `settings.py`. This means that any settings you need to apply in a project can’t simply be applied in your `settings.py` if an addon also needs access to them.

For example, nearly every addon will add a package, or sometimes several, to `INSTALLED_APPS`. If you were to assign do `INSTALLED_APPS = [...]` in the usual way, you would overwrite the existing assignments and break the project. That’s why our `settings.py` uses:

```
INSTALLED_APPS.extend([
    # add your project specific apps here
])
```

The same goes for middleware, and other settings.

`aldryn_config.py` is loaded into the Django project at run-time, so any changes are picked up when and reloaded automatically when developing.

`aldryn_config.py` is an ideal place to check for environment variables that should be converted into Django settings.

See Addon configuration with `aldryn_config.py`.

addon.json

A metadata file.

```
{
    "package-name": "susan-example-application",
    "installed-apps": [
        "susan_example_application"
    ]
}
```

setup.py

`setup.py` will be generated by the Control Panel on the basis of the information you provided when you first created it there. The lines highlighted below are those that will be specific to your addon:

```
# -*- coding: utf-8 -*-
from setuptools import setup, find_packages
```
from susan_example_application import __version__

setup(
    name='susan-example-application',
    version=__version__,
    description=open('README.rst').read(),
    author='Susan',
    author_email='susan@example.com',
    packages=find_packages(),
    platforms=['OS Independent'],
    install_requires=['example_application==1.8.3'],
    include_package_data=True,
    zip_safe=False,
)

How settings are handled in Django addons

In Django projects, settings are handled via the settings module (usually, the settings.py file).

In Aldryn addons - those that include an aldryn_config.py file - many of these settings will be automatically managed by the addon itself. This takes place in aldryn_config.py.

All key settings (i.e. settings required for the package to function correctly) as well as many optional settings will be configured. They are then applied to the settings module via the lines:

```python
import aldryn_addons.settings
aldryn_addons.settings.load(locals())
```

From this point in the settings module, those settings that were automatically configured by the addon will be available in the settings.py file.

For example, in a Django project, you will find a file:

```bash
addons/aldryn-django/aldryn_config.py
```

This file adds items to the INSTALLED_APPS, MIDDLEWARE, and applies other settings.

These settings can be controlled and determined in a number of different ways.

Via addon settings in the Control Panel

An addon can expose options for configuration in the Control Panel interface. For example, Aldryn Django has a Remove URL language prefix for default language option. This will apply to all environments of the project.

The value is stored in JSON. You can find the JSON file in the project locally, for example addons/aldryn-django/settings.json.

Via environment variables

Environment variables are suitable for:

- environment-specific settings (e.g. database settings, since each environment should have its own)
- secret settings (e.g. keys for services and APIs)
Environment variables are better than the codebase for such settings. If committed as part of the codebase, they provide the same value in all environments, and they are vulnerable to being accidentally shared.

Via automatically applied environment variables

Some environment variables are provided automatically, and you don’t need to do anything about them at all.

Each project environment has its own variables provided for services such as the database (DEFAULT_DATABASE_DSN), media storage (DEFAULT_STORAGE_DSN) and so on. Locally, the variables are saved in the .env-local file and loaded into the environment via docker compose.

Via user-configured environment variables

Other environment variables can be provided by the user, via the Control Panel’s Env Variables view:

ENVIRONMENT VARIABLES

TEST ENVIRONMENT VARIABLES

SECRET_API_KEY = "aaPfaHloJ5pdqYBc"

If you need the variable in the local development environment as well, add:

SECRET_API_KEY = "aaPfaHloJ5pdqYBc"

to its .env-local.

Manually in settings.py

As mentioned above, all these settings will be applied to the settings file by the aldryn_addons.settings.load(locals()) function. If any of them were written into the file manually before this point, it will overwrite them. Any settings you wish to provide manually should be added after the function to avoid this.

Overwriting automatically-configured settings

Overwriting automatically-configured settings is almost always a bad idea. For example, multiple addons may have added their own requirements to the MIDDLEWARE setting. If you simply do:

MIDDLEWARE = [...

you will obliterate the automatic configuration (or if you place your setting before aldryn_addons.settings.load(locals()), your own setting will be overwritten).

If for example you need to specify additional middleware, the safer and more sophisticated way to do it is by manipulating the list (see How to configure Django settings).

To understand which settings are provided automatically, you can:

- examine the addon’s aldryn_config.py file
- check the reference documentation for Aldryn Django, Aldryn SSO and Adryn Addons, where many important settings are listed
You can list changed settings to see those that have been altered from Django’s own defaults.

**Django addons and templates**

Templates at the project level will override templates at the application level if they are on similar paths. This is standard Django behaviour, allowing application developers to provide templates that can easily be customised.

**On initial project creation**

For your convenience, when you first create a project, any templates in addons are copied to the project level so you have them right at hand (if the addon’s package name and inner application name match.)

For example, templates from Aldryn News & Blog will be copied to templates/aldryn_newsblog/ in your project.

If a template does not exist in the project’s templates directory, Django will simply fall back to the one in the addon itself.

**Subsequent addon updates**

After templates have been copied to the project’s templates directory, they will not be copied again, so as not to overwrite any changes the project developer may have made. However, this does mean that if an addon is subsequently updated and its templates change, those changes will not appear in your project.

In this case:

- if you have made changes to the templates in your project, you will need to obtain any updated templates and merge them with your own versions
- if you have not made any changes, you can simply delete your local versions and Django will use the updated application templates.