# Learning Objectives

- **Identify the importance of 12 Factor Apps**
- **Define each of the 12 factors:**
  - **Codebase**
  - **Dependencies**
  - **Config**
  - **Backing Services**
  - **Build, Release, Run**
  - **Processes**
  - **Port Binding**
  - **Concurrency**
  - **Disposability**
  - **Dev/Prod Parity**
  - **Logs**
  - **Admin Processes**

# What are 12 Factor Apps?

# 12 Factor Apps

In Module 2 we'll be discussing how to get your Django application set up for deployment. It starts with some theory, by discussing 12 factor apps.

## What are 12 factor apps, and why do we need them?

There aren't really any standards for how a web application should be developed and deployed. For example, Django stores settings in `settings.py`, whereas NodeJS application might have them in a `settings.js` or `settings.json` file. Even two Django apps might differ in how they log data – maybe one will log directly to syslog and another will choose to open a specific file and log data to it.

The Twelve-Factor App describes a way of designing web applications to make them easier to deploy, configure, debug and scale. It's not specific to any particular language, so the twelve factors can be applied to Python, Ruby, JavaScript, Java, and other applications. They are more of a guideline, and depending on the size of your application, not all of the factors may be applicable to you.

Let's look at the twelve factors and discuss which ones are important to consider when developing a Django project.

# Factors 1-3

## The Twelve Factors

Note that throughout this section, for brevity, we'll use terms like "you should do…" or "your application should…". What these actually mean are "The Twelve Factors says you should do…" or "The Twelve Factors say your application should…".

### 1. Codebase

You should have a single codebase/repository, from which you can check out the code to different environments. Each deployment can then be configured, based on if it's a development, production, etc. Generally this is the de-facto standard – it's extremely rare to find cases where an application has one Git repository for local development, and another for production.

1. Codebase on 12Factor.net

### 2. Dependencies

Your application shouldn't depend on system-wide packages being installed. When developing with Django, usually we would include a `requirements.txt` file, or use a tool like Pipenv to create a list of dependencies for a our project. We'd also make use of virtual environments to isolate our project's dependencies from others.

With Python, it's not always possible to completely isolate yourself from system packages. Because some of the Python libraries are wrappers around (or otherwise dependent on) lower level code written in C, specific versions of system packages might need to be installed to provide these dynamic libraries. For example, some database connection libraries, machine learning toolkits or image processing libraries (like Pillow) do require system packages to be installed. You'll need to come up with a way of listing these system packages too, depending on the operating system you're deploying to.

2. Dependencies on 12Factor.net

### 3. Config

Your application should read its settings from environment variables. These are settings that are configured by your operating system before your application starts. They can be process-specific, so even on a single machine you can start your Django app with two different sets of environment variables and thus have it behave in two different ways.

A standard Django setup just reads settings from `settings.py`, and there are few different ways to augment how settings are read. For example, you can create a `production_settings.py` file and tell Django to read from that instead of `settings.py`. Or, store settings in JSON or .env files and have a third party library read them. We're going to use the Django Configurations third party library, which reads settings from environment variables. We'll look at setting this up in the next section.

3. Config on 12Factor.net

# Factors 4-6

## 4. Backing Services

When writing code for your application, it shouldn't make a distinction between local, internally hosted or third party services. The main example with Django is databases. Thanks to Django's ORM abstracting database communication, we can easily switch out the backend database simply by updating the setting. This means when our app is deployed to production, we can switch from SQLite to MySQL, for example, without updating our code – it's just a configuration change. Another example is sending email: when developing, use the console backend to just display messages in the local terminal. When you go to production, settings changes will allow you to use a real SMTP server.

When developing in Django, if you have to write custom code to talk to an external service or resource, just keep in mind what changes might have to be made to it when going to production.

4. Backing Services on 12Factor.net

## 5. Build, Release, Run

Your application should be deployable in three distinct stages: build, release, run. This exact process is hard to prescribe as there are so many options out there for doing it, both manually and automatically. We'll look at a manual way of performing this process just to give an example, and outline some guidelines.

**Build Process**

When you're ready to perform a release, you should build a new version. A manual way of doing this might be to create a Git tag of the particular commit that is to be released. If you need to make changes or bug fixes to the code, you can't edit the version, a new one must be created. In this case the build artifact is tracked by Git itself. Some other options might be to create an archive (like a *zip* or *tar* file) of the repository at that point in time.

**Releasing**

When it's time to release to a server, you can check out the code from the Git repository for the new tag. Or, extract the archive file. Once that's done you need to get the system up to date, by doing tasks like performing

database migrations, or restarting the web server. These tasks should only happen once, at release time (i.e. you don't have to run `manage.py migrate` every time you start the Django server.

**Running**

After releasing, the application can be run any number of times. For example, if a web server needed to be restarted, we wouldn't need to build and release again, we could just run the app. A manual way of doing this might be to log into the web server and restart the service that's running Django.

When developing a Django application, we don't really need to keep this process in mind, it's more an after-the-fact decision that doesn't depend on how our application is developed. Django itself already separates out the releasing and running stages into seperate management commands for us.

5. Build, Release, Run on 12factor.net

## 6. Processes

Your app should be stateless, and non persist any of its data in memory. Instead, it should be stored in databases, or on disk. This means that your application can restart and not lose any data, as it's stored outside its own process.

Django is designed this way, and the default method of storing data is in a database via its ORM. With only a few exceptions (such as local-memory caching) Django stores all data outside its own process. Even this example isn't really a problem as the cache is not designed as a permanent store and the data should be backed by a persistent backend. Unless you're going out of your want to persist data in memory, you're probably fine with this factor.

6. Processes on 12factor.net

# Factors 7-9

## 7. Port Binding

Your application should expose its interface by binding to a specific TCP port, instead of injecting itself into some other execution environment. As part of the development process, this is not something you need to worry about.

You can think of this like when you use the `runserver` command, the Django development server listens on port 8000 on your local machine. When you deploy to production, an app server like Gunicorn or uWSGI will perform the same task, exposing your application over TCP, but in a more robust way. A web server like nginx can then load balance your application.

7. Port Binding on 12factor.net

## 8. Concurrency

Your application should be able to be run in multiple processes concurrently. This will allow your deployment to scale horizontally across multiple processors or even multiple servers.

▼ **Horizontal Scaling**
*Horizontal scaling* refers to getting more performance by running the project across multiple CPUs or multiple servers, and balancing requests between them. Compare this to *vertical scaling* which means getting more performance by using a more powerful computer.

In most cases, Django is automatically able to be run concurrently, so it's not something you usually have to worry about when developing. Unless you're manually locking access to files or the database, so that only one process can access them at one time, then you shouldn't have a problem. If you are doing something like this, then keep in mind that it might cause one process to stall while it waits for access to that resource.

8. Concurrency on 12factor.net

## 9. Disposability

Your application process should be disposable, that is, it should start up quickly, stop quickly when requested, and not lose data if it crashes.

As Django developers, this isn't something we have control over, when developing our project. Django processes normally start within a few seconds and so can be quick to spin up and scale horizontally. If your application isn't fast to start up, you would have to investigate to find out the cause.

We have to trust in the app servers (Gunicorn, uWSGI, etc) to shut down gracefully when told to do so, and most of them do. Guaranteeing no data loss during a crash isn't possible. You could make use of manual database transaction management if you have a collection of database operations important to be written at the same time, but that's something to be considered on a case-by-case basis.

9. Disposability on 12factor.net

# Factors 10-12

## 10. Dev/Prod Parity

Your development environment should be kept as close to production as possible. This is another factor that's not necessarily Django or coding related, but more to do with how you set up your system. And like others, its importance depends on the complexity of your application.

For example, in Blango, we're using models with basic fields and relationships, and there's not really any discernible difference in behavior between using SQLite in development and a different database like PostreSQL in production. But if you start using database specific features like stored procedures, then different databases will cause more of an issue.

Tools like Docker can make it easier to get consistent environments. For example, even if you develop on macOS you can run Django in a Linux container, with all the supporting services that you might need. You can even then run under Docker in production.

If you're trying to solve particular problems, there are limits to how close you can get your development to production though. For example, if your application starts having performance problems or bugs once you have a billion rows, it might be hard to replicate on your development laptop. Similarly, trying to replicate a 64-core processor with 512GB of RAM to mirror production can be costly for each developer.

A more useful setup is to have a staging environment that more closely mirrors the resources of production. Releases go to staging first, for load testing and validation. It's even better if you can clone data from production to staging to get a really accurate picture.

10. Dev/Prod Parity on 12factor.net

## 11. Logs

When your application generates logs it should, only write to `stdout`. This means that during development, you can see all the logs that your application is generating as they'll be output to the console. When the application is deployed, the operating system can be configured to route the logs, to a file or a log collection tool (like Logstash).

The alternative would be to configure your application to log to a file that you specify. This is much more difficult for the operating system to route, and you'd need to change the configuration to see the logs when running in

development.

We'll look at configuring Django logs later in the module.

11. Logs on 12factor.net

### 12. Admin Processes

Admin tasks should be run as one-off processes, in the same environment as production. The Django admin task that you'll probably run most often is `migrate`. You should be able to, for example, log into the production server and run `manage.py migrate` using the exact version and configuration of the code that's running and serving requests. Contrast this with connecting your laptop to the production database and running the command. Doing something like that might cause issues due to discrepancies between the two configurations.

Like many of the 12 factors, this isn't something that you need to consider during development, but might influence how your production environment is set up.

12. Admin Processes on 12factor.net

## Conclusion

As we've seen, many of the 12 factors don't necessarily need to be considered during development and won't change your day-to-day workflow. But keep them in mind when developing and in particular, deploying.

We're going to look at how to configure Django to work with factors 3 (configuration) and 9 (logs) in more detail in the next two sections.