



# Understanding Databases:

Deploy High-Performance Database  
Clusters in Modern Applications



**EXTENDED EDITION**

Includes Practical Application Instructions: Efficiently Deploy Django  
Into Production with Linode Managed Databases for MySQL

# **Understanding Databases:** Deploy High-Performance Database Clusters in Modern Applications

*by*

*Justin Mitchel*

*Akamai Technologies*

# Understanding Databases: Deploy High-Performance Database Clusters in Modern Applications

Justin Mitchel

© 2022 Akamai Technologies

All rights reserved. No part of this publication may be reproduced, stored in a retrieval system or transmitted in any form by any means, electronic, mechanical, photocopying, recording or otherwise without the prior permission of the publisher or in accordance with the provision of the Copyright, Design and Patents Act 1988 or under the terms of any license permitting limited copying issued by the Copyright Licensing Agency.

**Published by:**

Akamai Technologies  
249 Arch Street  
Philadelphia, PA 19106

**Typesetting:** Jill McCoach

**Cover design:** Jill McCoach

## Dedication

*To my wife, Emilee: Thank you for being such an amazing partner and mother. You make it possible for me to do the work that I do.*

*To my kids, McKenna, Dakota, & Emerson: Thank you for being you! I cherish your sense of humor, adventure, and curiosity. Thank you for teaching me every day.*

*I love you all and look forward to what our future brings!*

# Part 1

The background is a vertical gradient of green, transitioning from a dark teal at the top to a bright, vibrant green at the bottom. Scattered across the entire background are numerous small white stars of varying sizes and shapes. Some stars are simple dots, while others are four-pointed or six-pointed. Faint, thin white lines connect some of the stars, forming subtle constellations or star patterns. The overall effect is a clean, modern, and celestial-themed design.

# Part 1: Table of Contents

<b>Introduction</b> . . . . .	<b><a href="#">02</a></b>
<b>Section 1: Evaluating the Strengths and Limitations of Database Options</b> . . . . .	<b><a href="#">03</a></b>
Current Database Landscape	<a href="#">04</a>
What is ACID Compliance?	<a href="#">06</a>
What is BASE Model?	<a href="#">06</a>
Overview of Popular DBMSes	<a href="#">07</a>
<b>Section 2: Designing Database Architecture.</b> . . . . .	<b><a href="#">12</a></b>
Reference Architecture Examples	<a href="#">14</a>
Single Server Deployment and Vertical Scaling	<a href="#">15</a>
Horizontal Scaling with MySQL Replication	<a href="#">17</a>
Horizontal Scaling with MySQL NDB Cluster and Database Sharding	<a href="#">21</a>
Incorporating Monitoring	<a href="#">22</a>
Monitoring Options	<a href="#">22</a>
<b>Section 3: Deploying a Database with a Managed Database Services</b> . . . . .	<b><a href="#">23</a></b>
What is a Managed Database?	<a href="#">24</a>
Migrating to a Managed Database	<a href="#">25</a>
<b>Section 4: Managed Databases and the Alternative Cloud.</b> . . . . .	<b><a href="#">26</a></b>
Cloud Provider Considerations for Databases	<a href="#">28</a>
What to Look For When Choosing an Alternative Cloud Provider	<a href="#">29</a>
<b>Section 5: Our Take</b> . . . . .	<b><a href="#">30</a></b>
Our Take	<a href="#">31</a>
<b>Section 6: About</b> . . . . .	<b><a href="#">32</a></b>
About the Instructor	<a href="#">33</a>
About Akamai	<a href="#">33</a>



# Introduction



# Introduction

Every web application requires a database. In your tech stack, the database enables your application to be truly interactive, from storing contact form submissions to powering advanced personalization features. The term “stack” explains how each technology used in your application is an individual component and has its own layer, even if they are installed together right after a server is deployed or as part of an image.

Modern applications constantly change how we think about data and how relationships are created between datasets. This rapidly evolving data landscape requires careful selection of a database management system (DBMS)—or possibly multiple databases—that meets your users’ performance expectations. Here are just some of the factors that shape your choice of a primary or auxiliary database.

- Whether your database needs to be highly available.
- Speed for read operations vs. write operations.
- Read operations fetch requested information from a database.
- Write operations add data to a database or modify existing data in a database.
- Data encryption options to meet security requirements.
- Whether your data can be modeled in a traditional relational schema, or if your data’s structure requires more flexibility.
- Value of open source databases versus proprietary solutions.

In Part 1 of this ebook, you’ll develop a high-level understanding of industry-standard databases, the design of database architectures, and different deployment methods in the cloud. Part 2 contains a practical project application designed by Justin Mitchel of Coding for Entrepreneurs. The project includes step-by-step instructions on how to use Django, Docker, and Linode Managed Databases for MySQL to create a production-ready application from scratch.

VIDEO SERIES AVAILABLE!

## Deploy Django to Linode using Managed Databases for MySQL Video Series

WATCH NOW



Section 1

# **Evaluating the Strengths and Limitations of Database Options**

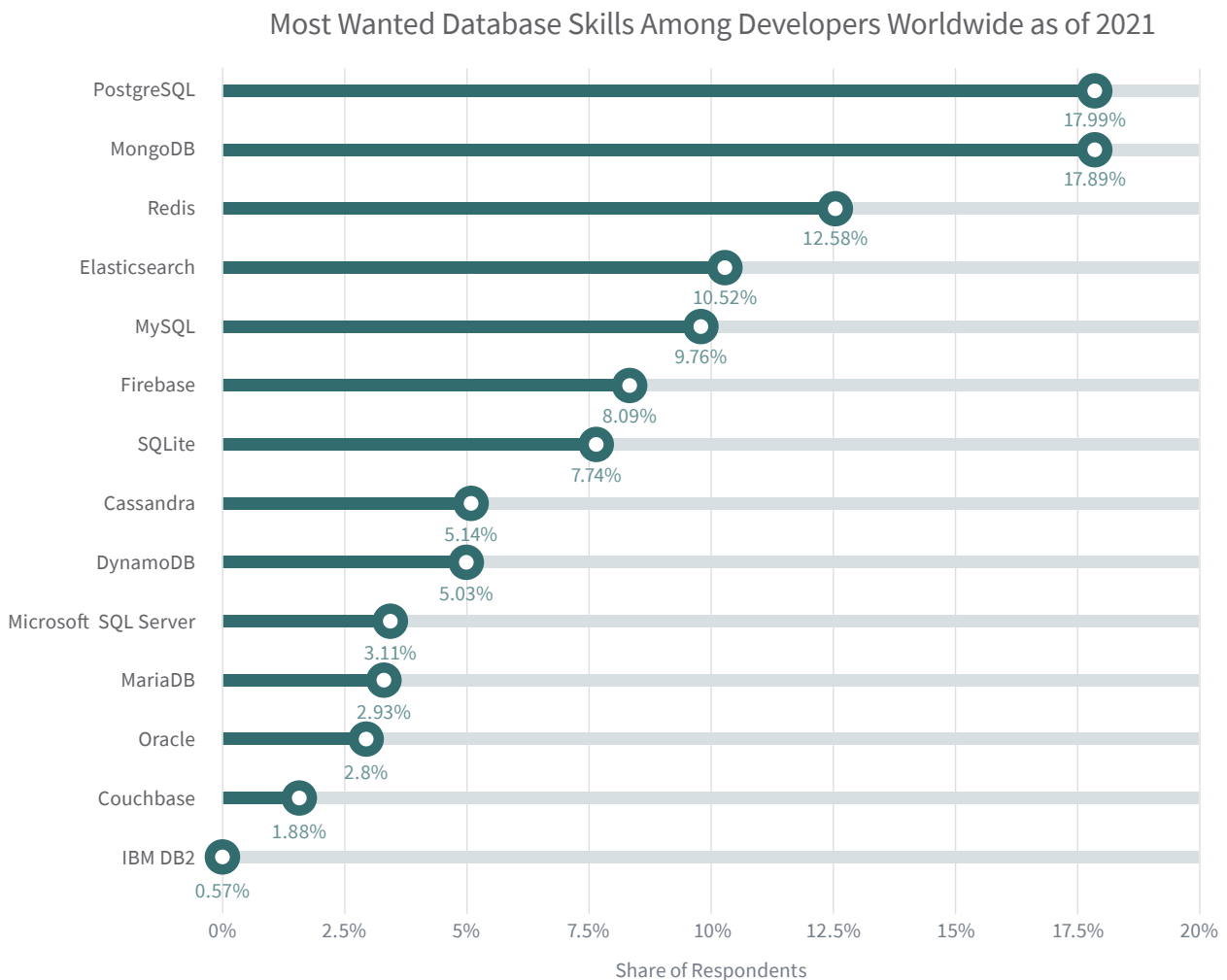
## Section 1

# Evaluating the Strengths and Limitations of Database Operations

Databases are anything but “one size fits all.” From privacy and compliance to supporting specific data types, databases only get more advanced as an application scales. Even if you aren’t a database administrator (DBA), you need to factor in database capabilities and limitations when adding new features or optimizing existing features to your applications.

### Current Database Landscape

The process of scaling applications and improving services can introduce new data types. The need for more flexible data types has launched many new databases in the market in the past decade. As a result, database skills have become increasingly more valued by developers.



According to the survey, just under 18% of respondents identified PostgreSQL as one of their most-wanted database skills. MongoDB ranked almost the same with software developers stating they are not developing with it, but want to.

Source: [Statista](#)

Databases can be split into two types: relational and non-relational.

- **Relational Databases** (also referred to as SQL Databases). Data is organized in tables, where the columns of a table correspond to the data's attributes, including the type of an attribute. For example, an employee table for an HR software suite could contain a column that stores an integer value representing an employee's salary, and another column could store a text value representing their name. The rows of the table represent instances of the data; for example, each row in an employee table would correspond to a different employee. Information between multiple tables is linked via keys. Relational databases use the Structured Query Language (SQL).
- **Non-Relational Databases** (also referred to as NoSQL ("Not Only SQL") Databases). Non-relational databases have more flexible query methods. These query methods vary significantly by DBMS type. There are five types of non-relational databases:
  - **Columnar Data Stores:** Data is organized in a similar row-and-column table structure like in a relational database. However, the data is stored by column, and data from specific columns can be fetched. In comparison, data in a relational database is stored by row, which allows you to fetch specific rows from the table.
  - **Key-Value Stores:** Data is stored as a collection of key-value pairs, where data can be retrieved by specifying a key that is associated with the data.
  - **Document Stores:** Data is stored in documents, like those on your computer's filesystem.
  - **Document Data Stores:** A document store where the files' type is JSON, XML, or some other data-encoding format. The structure of the encoded data is flexible and can allow for complicated queries.
  - **Graph Databases:** Data is stored using a graph structure, where entities correspond to nodes in the graph and the edges between nodes represent relationships between entities. An example of a graph structure would be the connections between users on a social network. Graph databases offer efficient querying for highly-interconnected data.

## Relational



ID	First	Last	Company	Dept	Hire Date	Role	Office
249	Linny	Penguin	Linode	Mktg	06/2003	Mascot	PHL
248	Justin	Mitchel	CfE			Friend	

In this example, we're comparing how data is stored in MySQL (a relational database) versus a document-based non-relational database like MongoDB.

## Non-Relational

dept: **Linode**

```
{
  "_id": 249
  "name": "Linny Penguin",
  "dept": "Linode Mktg",
  "hire_date": "June 2003",
  "office": "PHL",
  "role": "Mascot",
}

{
  "_id": 248
  "name": "Justin M",
  "dept": "Linode",
  "role": "Friend",
}
```

## What is ACID Compliance?

ACID compliance for databases is a set of principles that ensures data integrity while processing a transaction. These principles ensure that the data will not end up in an inconsistent state or be altered as a result of being added to the database, even if the transaction fails. ACID compliance is particularly important for applications that are essential to the financial industry, with millions of transactions processed every second.

### ACID Compliance:

- **Atomic:** Transactions are made up of components, and all components in a transaction must succeed. Otherwise, the transaction will fail and the database will remain unchanged. A transaction must have defined success or fail actions, or unit of work.
- **Consistent:** Successful transactions follow the database's pre-defined parameters and restrictions.
- **Isolated:** Concurrent transactions are isolated from one another. The result of executing several concurrent transactions will be the same as if they were executed in a sequence.
- **Durable:** Once a transaction is written to the database, it will persist, even in the event of a system failure for the server that the database is running on.

Out of the box, NoSQL databases are not ACID compliant, because they are designed to prioritize flexibility and scalability instead of pure consistency.

## What is the BASE Model?

With the increasing use of NoSQL databases, a different transaction model was created to better align with their functionality. Sticking with the chemistry theme, we have the BASE Model.

### BASE Model:

- **Basically Available:** Data is replicated and dispersed across clusters so that failures of part of a cluster should still leave the data available in other locations.
- **Soft State:** Consistency is not immediately enforced across the database cluster after an item in the database is updated. During this time, fetching an updated record from the database may result in different values for different read operations.
- **Eventually Consistent:** Data will eventually reach consistency as updated records are replicated across the nodes in the database cluster.

The clear difference between the ACID and BASE transaction models is seen in the practical use of SQL vs NoSQL databases. NoSQL databases adhering to the BASE model are built for the flexibility and scalability required by massive high-availability deployments. SQL databases adhering to the ACID model value consistency and data integrity above all else.

## Overview of Popular DBMSes

Here's a quick overview of five top databases that serve a range of use cases. This list includes both SQL and NoSQL options.

### SQL



**MySQL** is one of the most widely used database engines. MySQL is a component of the LAMP stack, which serves as the foundation for many content management systems, including WordPress. MySQL is known for its high performance (especially for processing read operations), ease of use, and scalability. Originally founded as an open source project, MySQL was acquired by Oracle in 2010. A free community edition of MySQL is still available, but some features and scalability require a paid enterprise license.

#### Strengths

- **Speed:** Query limits and default read-only permissions make MySQL extremely performant, and there are more options available for memory-optimized tables.
- **Integrations:** Due to its popularity, MySQL has a larger variety of third-party tools and integrations than other database types.
- **Reliability:** MySQL is widely used and vendor-supported.

#### Weaknesses

- **Write Performance:** Without optimizations, performance will decrease for write-heavy applications.
- **Tiers:** Features are split between the free Community edition and the paid Enterprise edition, which limits access to features and potential scalability.

**Use Cases:** Websites (General), Ecommerce, Catalog/Directory



## SQL



**PostgreSQL**, also known as Postgres, is considered to be the most advanced open source database and a more robust SQL database alternative to MySQL. Postgres is also the stronger choice for workloads that are dependent on database write operations and the ability to add custom data types. Postgres is open source and all features are available without a paid license for any size deployment, in contrast to MySQL.

### Strengths

- **True Open Source:** Postgres is maintained by a global community, and a single version of the software is available that contains all features and has no fees.
- **Extensibility:** Postgres supports a wider range of data types and indices than MySQL, and additional feature extensions are provided by the open source community.
- **Data Encryption Options:** Multiple encrypted connection options.

### Weaknesses

- **Server Memory Demands:** Postgres forks a new process for each new client connection. Connections allocate a non-trivial amount of memory (about 10 MB).
- **Increased Complexity over MySQL:** Postgres adheres more strongly to SQL standards so it can be a difficult starter database for standard web applications. For example, querying in Postgres is case-sensitive.

**Use Cases:** High-Scaling Applications, Multi-Database Workloads

## NoSQL



**MongoDB** is a document database that stores data as JSON documents to provide more flexibility for scaling and querying data based as an application evolves. MongoDB is a solution for database requirements that stray away from relational data schemas. This approach allows users to start creating documents without needing to first establish a document structure, which provides developers with more flexibility when updating their applications.

### Strengths

- **Scalability:** It's in the name: Mongo, short for "humongous," is built to store large volumes of data.
- **Search Flexibility:** Supports graph search, geo-search, map-reduce queries, and text search.
- **Flexible Data Options:** More functionality for temporary tables to support complex processes or test new applications/features without needing to switch databases.

### Weaknesses

- **Concurrent Multi-Location Data Write Operations:** Updating a consistent field in several locations (e.g. a commonly used location) can cause lags in performance compared to a relational database
- **Query Formatting:** MongoDB can have a steeper learning curve for developers who are familiar with other databases because queries are written with a JSON syntax, instead of SQL.

**Use Cases:** Personalization Engines, Content Management, Games, IOT Applications

## NoSQL



**Redis** (short for “REmote DIctionary Server”) is an open source in-memory database that is useful for applications that require rapid data updates in real time. Redis stands out from relational databases by using key-value pairs to store data, resulting in faster response times when fetching data. Redis is ideal for projects with a clearly-defined data structure and when speed is more important than scalability.

### Strengths

- **Speed:** Since Redis only stores data in memory, it’s a highly-performant database that rapidly returns results.
- **Benchmarking:** The built-in benchmark tool, `redis-benchmark`, provides insights on the average number of requests your Redis server is able to handle per second.
- **Ease of Use:** Supports a variety of extensions, including `Redisearch`, which provides full text search for a Redis database.

### Weaknesses

- **Memory Restrictions:** Since Redis is an in-memory store, all data must fit within your available memory.
- **Rigidity:** Redis doesn’t have a query language, and entering functions via the available commands can limit your ability to customize results.
- Not ACID compliant out of the box. Additional configuration is required

**Use Cases:** Caching, Real-Time Data Updates

## NoSQL



**Cassandra** is an open source columnar database that supports very large amounts of structured data. Cassandra does not declare a primary server; instead, data is distributed in a cluster of nodes, each of which can process client requests. This provides an “always-on” architecture that is extremely appealing for enterprise applications that cannot experience database downtime. Cassandra was originally an open source project developed by Facebook in 2008 to optimize message inbox search. It was later declared an Apache top-level project in 2010.

### Strengths

- **“Ring” Architecture:** Multiple nodes are organized logically in a “ring.” Each node can accept read and write requests so there is no single point of failure.
- **Ease of Use:** Cassandra’s query language, CQL, has a syntax similar to traditional SQL and has a reduced learning curve for developers switching from a SQL database.
- **Write Speed:** Cassandra is efficient for writing very large amounts of data. When performing a write operation, replicas of a new record are stored across multiple cluster nodes and these replicas are created in parallel. Only a subset of those nodes need to complete a replica update for the write operation to be considered successful, which means that the write operation can finish sooner.

### Weaknesses

- **Read Time:** Records in a Cassandra database are assigned a primary key attribute. The value of the primary key determines which cluster nodes a record is stored on. When querying data by primary key, read performance is fast, because the node that stores the data can be found quickly. However, querying data using attributes other than the primary key is slower.
- Not ACID-compliant out of the box. Instead, Cassandra offers several levels of trade-off between data consistency and availability for developers.

**Use Cases:** Personalization Engines, Content Management, Games, IOT Applications

Section 2

# Designing Database Architecture



## Section 2

# Designing Database Architecture

Many websites and web applications can be operated with a single-server configuration, where your database exists on the same server as all of the application's other software components. This can be appropriate when your server (possibly when paired with external storage, like block storage) has enough compute and storage resources to accommodate your application's data and traffic. This approach is fine for smaller, general use websites that might host some forms and media files. For example, a WordPress blog can run well on a single-server setup.

As your application's features and data requirements grow, your database infrastructure needs additional flexibility and scalability to meet the performance expectations of your users or customers. Necessary changes could include moving the database to its own compute instance(s), or designing an advanced architecture with replicas, read-only instances, sharding, and/or other components. These changes can enhance both the performance and security of your database.

Application performance is a shared responsibility. Even if you or your team doesn't currently prioritize conversion rates, poor application performance will come back to haunt you in other ways, including: decreased user trust, violation of your application's SLA (the guaranteed availability of your services), and an increase in support tickets.



Research into human response indicates applications have roughly 100 milliseconds (ms)—one third of the time it takes to blink—before users feel like they're waiting for a response.

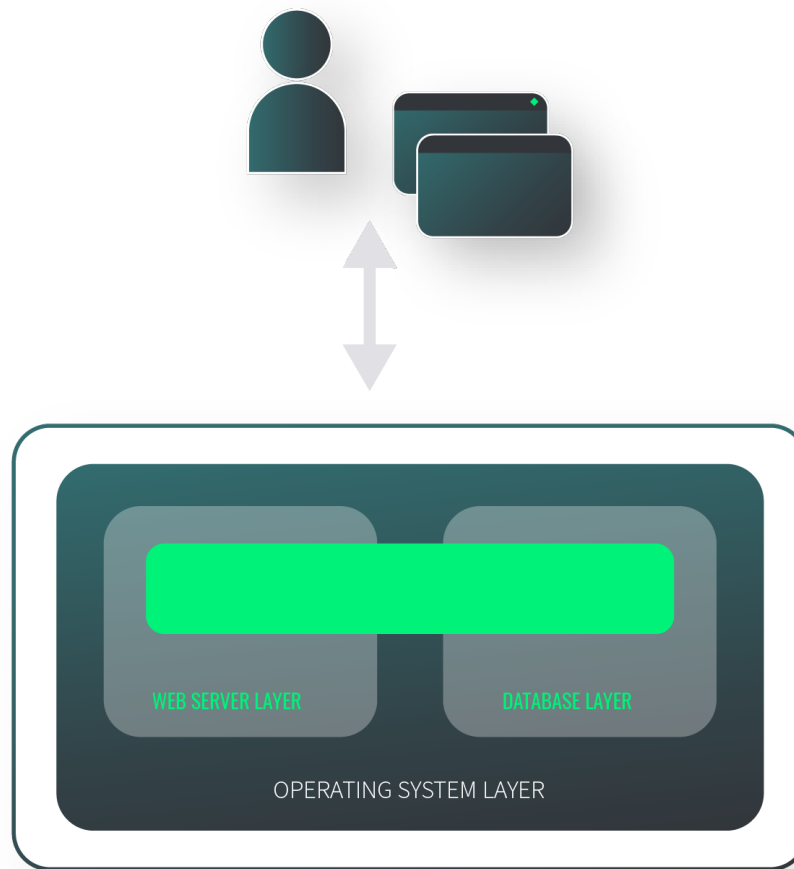
**Lee Atchison**  
Caching at Scale with Redis

To help get you started, here are five examples of database-specific reference architecture that show how these setups differ, and their benefits.

## Reference Architecture Examples

The typical real-world use case for a database is to store and handle structured information as part of an application's software stack. The foundation of a simple web application is the operating system of your server, with a web server, database, and application code layer running on top of it. The OS provides basic networking functionality that the web server relies on to serve content. In the code layer, a programming language and application software framework are used to direct the client-side UI in your users' browser. We can look at a server running a WordPress site to illustrate these concepts:

### Single Node Web Server Architecture



A WordPress site commonly leverages a software stack called *LAMP*, which is composed of the **L**inux operating system, **A**pache HTTP web server, **M**ySQL database, and **P**HP programming language. LAMP is one of the most popular web application software stacks in use. There are also alternatives to the LAMP stack. For example, you can use Python and an associated web application framework (like Django) instead of PHP. Or, you can use the NGINX web server instead of Apache; this is referred to as the *LEMP* stack (the “E” comes from the pronunciation of NGINX, which is “Engine X”). There are many other variations of this concept, but the following scenario focuses on a traditional LAMP stack.

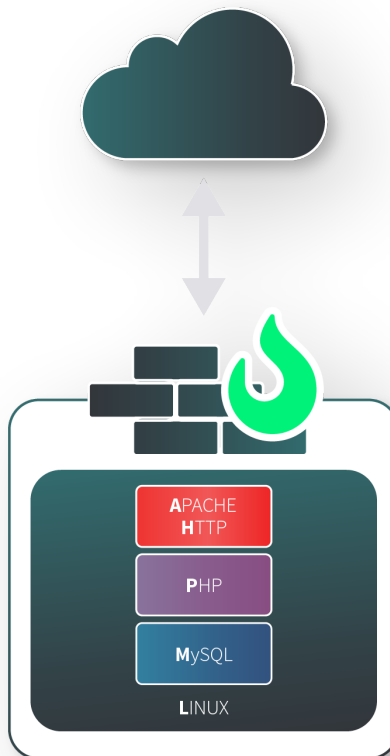
## Single Server Deployment and Vertical Scaling

Provisioning a LAMP stack on a single node can be done in two steps.

1. Deploy a Linux server, then install the other three software packages via your distribution's package manager. Once the stack is provisioned, your web application's code should be installed and configured. This work can often be automated with open source tooling; for example, [WP-CLI](#) can be used to install WordPress on a server.
2. You need to consider the network security and availability of your server. Many cloud providers offer network filtering to prevent or reduce the impact of denial-of-service and other kinds of network attacks. However, it's still important to fine-tune your network security configuration with a firewall.

In this section's example architecture, the firewall is a process that runs on your server. The firewall intercepts network packets when they are first processed by the operating system and only allows traffic on certain network ports to proceed. For a web application, your firewall would allow HTTP (port 80) and HTTPS (port 443) traffic. Linux and other operating systems have built-in firewall options, like [iptables](#), [ufw](#), and [Firewalld](#).

### Single Node LAMP Stack with Cloud Firewall

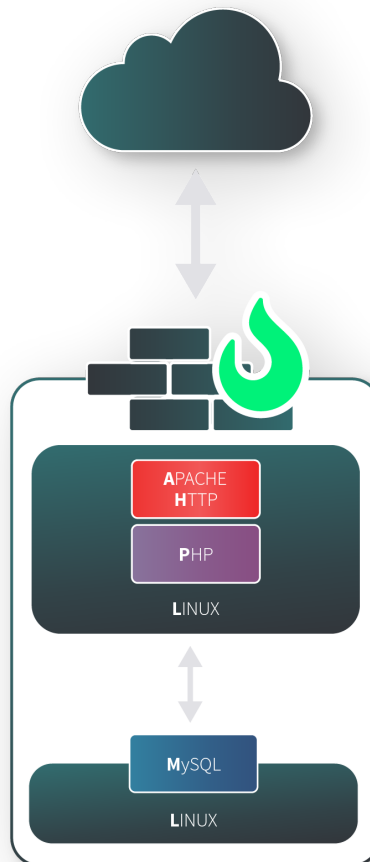


The resulting setup has one server running the LAMP stack where network traffic is filtered by a firewall also running on the server. There are limitations for this setup:

- **Growth and scaling:** With one server, you're confined to that server's memory, compute, and disk space resources. If your website starts seeing more traffic, you might need to increase the memory, compute, and/or disk space of the server to handle the load. This is referred to as *vertical scaling*. This process can vary in difficulty, depending on your cloud provider's tooling, or on hardware availability for on-premise deployments. Vertical scaling can only offer a limited solution for capacity, as there are upper limits on the resources for a single server.
- **Availability:** Operating a single server means that there is a single point of failure for your web application. If the server needs hardware maintenance, or if one of the software components in the stack halts unexpectedly, then your web application will not be available.

To some extent, the issue of traffic growth can be addressed by moving the database to its own server, as in the following diagram:

### Vertical Scaling with Separate Web and Database Nodes



By doing this, the compute, memory, and disk space needs of your web server and database can be adjusted independently. This is still an example of vertical scaling, and it does not resolve availability issues.

## Horizontal Scaling with MySQL Replication

Horizontal scaling can address high-traffic growth and solve availability issues. It addresses these issues by adding additional servers to your infrastructure. Together the servers form a *high availability (HA) cluster*. Adding servers can lead to more complexity in the design of your infrastructure and require changes to your services' configuration and/or application code. Even with these required changes, running your web application on an HA cluster with proper load balancing is the most reliable and efficient way to start building out your backend infrastructure.

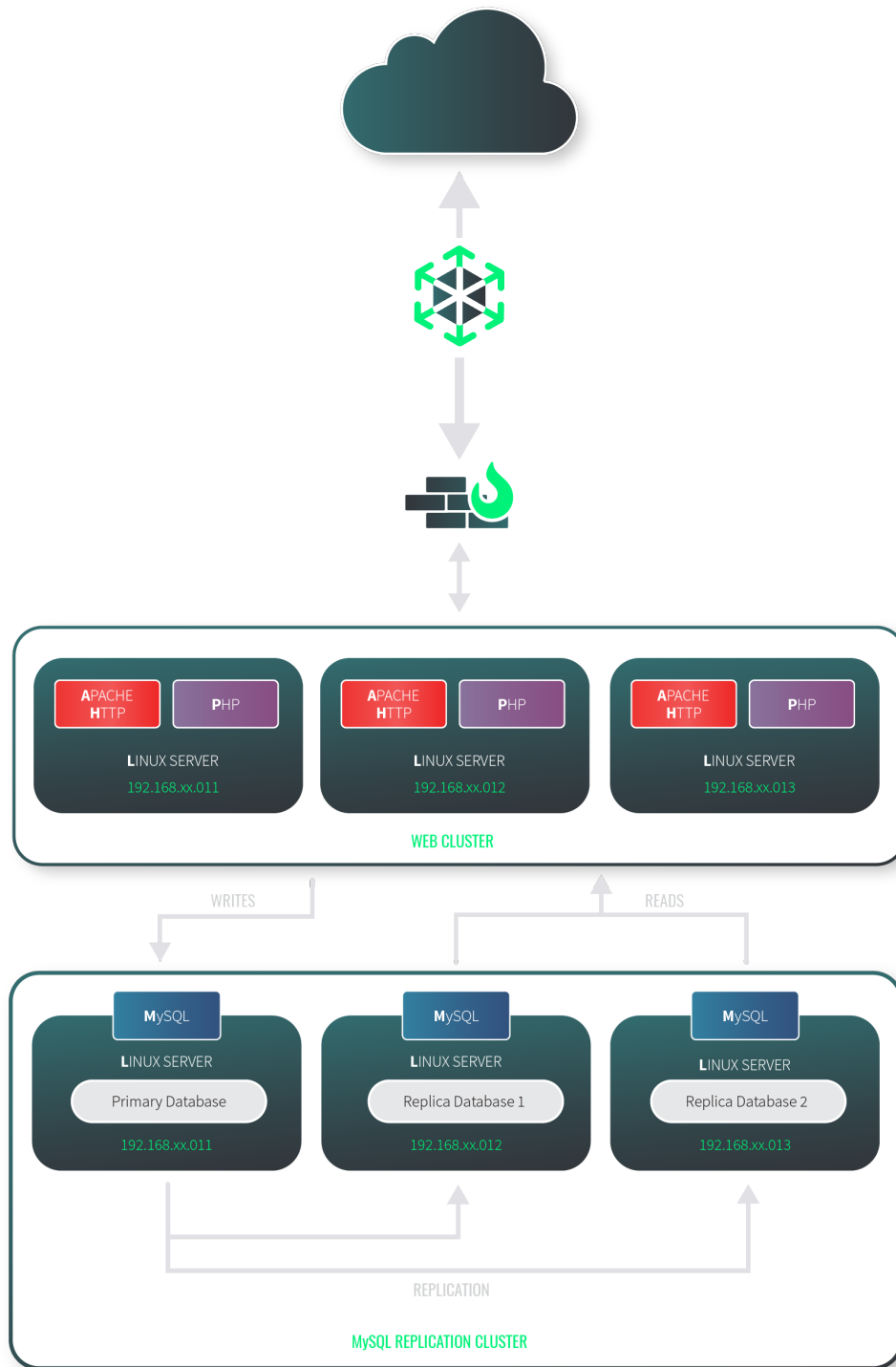
In the previous section, a two-server solution was presented that separated the web server from the database server. While it did not solve vertical scaling's problems, the separation of web servers and database servers is an important step towards creating a horizontal scaling solution.

In a horizontal scaling solution, the single web server is replaced with a high availability cluster of web servers, and the single database server is replaced with another, separate high availability cluster of database servers. By having two separate clusters in your architecture for these services, you can horizontally scale them independently of each other and choose server instances with different hardware that is better suited for each workload.

There are several different ways to architect your database cluster, each with varying degrees of complexity and different trade-offs. This section and the diagram below illustrate how to use *MySQL replicas* to facilitate high availability.



## Multi-Replication for Application High Availability



In the diagram above, a *load balancer* receives traffic from the Internet. A load balancer is a service that forwards and distributes traffic among another set of servers. Cloud providers generally offer load balancing as a service (Linode's NodeBalancer solution is pictured). This service can also be implemented with open source software, including HAProxy, NGINX, and the Apache HTTP Server.

Before the forwarded traffic arrives at your web servers, it is intercepted by a cloud firewall. The cloud firewall lifts the firewalling burden from the operating system to the network level. Many cloud providers offer managed cloud firewalls, but this service can also be implemented with open source software on commodity hardware.

After being filtered, the traffic is distributed among a cluster of web servers. In many cases, these web servers can all be identical clones of each other. You can add or remove web servers in this cluster as needed to handle scaling traffic demand, which is an example of horizontal scalability. If a server fails, the load balancer will re-route traffic to other healthy servers, so the service remains available. Server creation can be automated with configuration management and Infrastructure as Code (IaC) tools like Ansible and Terraform.

The web servers in turn request data from a database cluster. Unlike the web servers, the database server are not all identical in function. In particular, only one database is set to receive write operations from the web application. This server is designated as the primary database. For example, if you operated a news website, the new articles that your writers published would be added to the database on this server.

The other databases in this cluster act as *replicas* of the primary database. These servers receive all data added to the primary database through an asynchronous process. Because this process is asynchronous, write operations on the primary database are still fast to execute. These servers then receive all read operations from the web server cluster.

MySQL offers tools to help add new replicas to the cluster when needed, so your applications' read operations can be horizontally scaled. However, there is only one primary database in this architecture, so write operations cannot also be horizontally scaled. Still, this architecture can offer significant benefits to certain kinds of applications. In the news organization website example, the source of high-traffic demand is from readers of the website, not from writers adding articles. This traffic pattern aligns with the trade-offs in this setup.

There are a few other notable issues with this architecture. First, when a new write operation is received by the primary database, it immediately persists the updated records. The database does not wait for the replication process to complete, because that is asynchronous. This means that in the event of a primary database failure, the replicas might not have the most recently updated records. Also, when the primary database fails, it acts as a single point of failure for write operations. Still, read operations will continue on the replicas in this scenario, so a website can continue to display existing content. If the primary database fails, MySQL provides tools to manually promote one of the replicas to the position of the new primary database, after which write operations can resume.

Enhancements and alternative replication solutions exist to address these issues:

- MySQL replication can be configured to be *semisynchronous* instead of asynchronous. This means that the primary database will wait for at least one replica to process new write operations before they are persisted. Because at least one replica will have the latest data, you can be sure that you will not lose data in the event of a primary database failure. However, semisynchronous replication is slower than asynchronous replication.
- MySQL Group Replication is a replication solution in which the database servers automatically coordinate with each other. For example, in the event of a primary database outage, a replica server is automatically promoted to the new primary position. MySQL Group Replication can be configured with a single primary database, or it can be configured to have multiple primary databases that receive write operations. However, maintaining multiple primary databases involves other trade-offs.
- InnoDB Cluster bundles a MySQL Group Replication cluster with MySQL Router, which facilitates routing traffic from the web servers to the database cluster, and MySQL Shell, which is an advanced administration client for the cluster.
- Galera is a multi-primary database solution where all databases in the cluster can receive write and read operations using completely synchronous replication between them. It is also compatible with forks of MySQL like MariaDB and Percona.

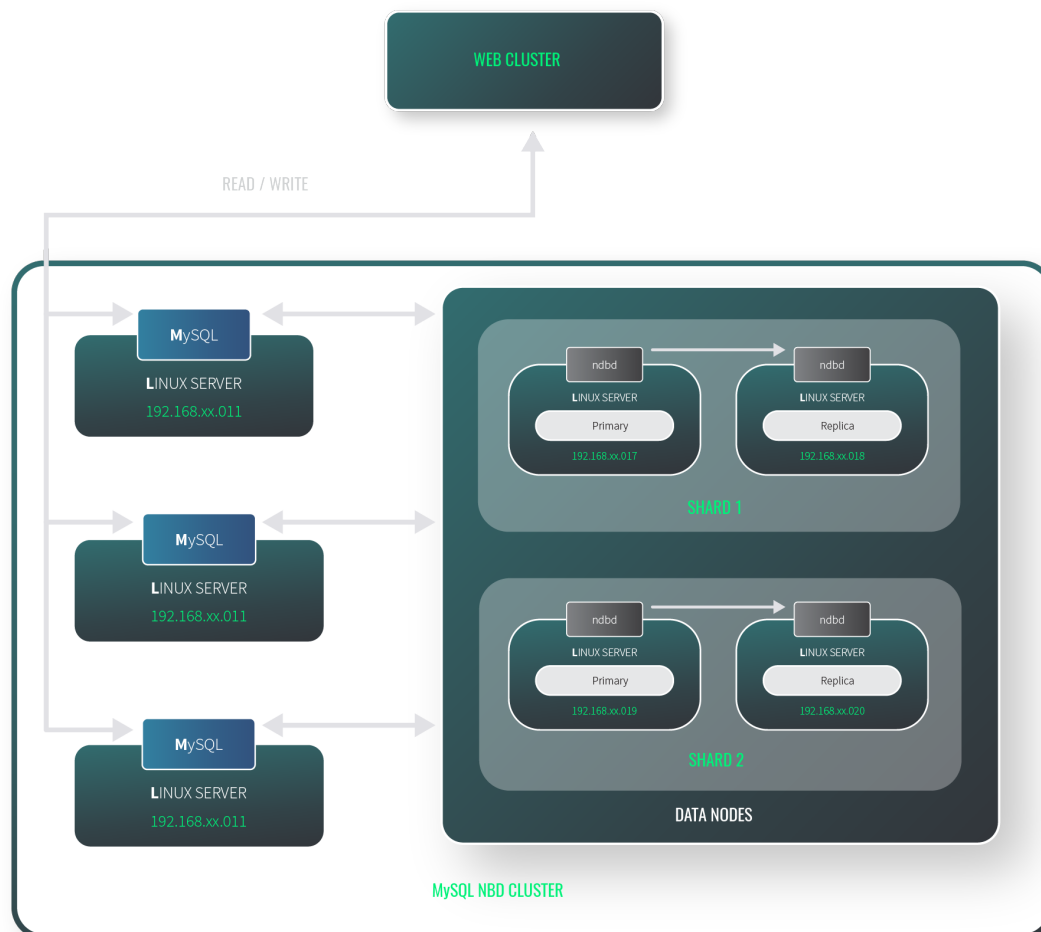
## Horizontal Scaling with MySQL NDB Cluster and Database Sharding

While the replication solutions presented can address horizontal scaling for high volumes of traffic, they do not address the needs of applications that store very large amounts of data. This is because the database replicas each maintain the full set of data for the application, and there are limits to the disk space that can be provisioned to a database server.

To address this issue, an architecture that implements *database sharding* can be used. With sharding, a single, large table is split into multiple smaller tables. The process of splitting a table is referred to as *partitioning*. When stored across multiple servers, these smaller tables are referred to as *shards*. The databases in your cluster each store one of the *shards*. Together, the databases in the cluster constitute your full data set. For example, if a human resources application stored employee records for 1,000 companies, but found that the dataset size was too big for a single table, then the records could be split into two shards representing 500 companies each.

MySQL NDB Cluster is a solution that provides automatic sharding for MySQL. The diagram below shows an example of how a MySQL NDB Cluster would fit into the example web application. This diagram omits the load balancer, cloud firewall, and web server cluster as those remain the same:

### Database Sharding for Large Data Sets



The MySQL NDB Cluster in this diagram provides three MySQL servers that accept read and write SQL commands from the web servers. The number of these servers can be horizontally scaled to meet demand and provide failover. These servers do not store data. Instead, they update and retrieve records from a separate set of data nodes.

The data nodes store shards of the dataset and run the **ndbd** data node daemon process. Each shard can have multiple replica nodes, and you can configure how many replicas per shard there should be. The total number of data nodes is equal to the number of shards multiplied by the number of replicas per shard. In the diagram above, there are two shards, and two replicas per shard (a primary replica and a secondary replica), for a total of four data nodes. Having multiple replicas per shard allows for recovery for failing nodes, and this recovery process is automatic (similar to MySQL Group Replication).

Using sharding can offer very high scaling of dataset size, but it can also make your application logic more complex. In particular, you need to carefully configure how your data is partitioned into multiple shards, because this decision impacts overall database performance. Sharding requires a more in-depth understanding of your database and the underlying infrastructure.

## Incorporating Monitoring

Database monitoring comes in a variety of formats, ranging from high-level reporting of system health to assessing granular operations that could impact application performance.

### Monitoring Options

- **DBMS monitoring extensions:** Extensions or add-ons within the database layer that provide insights on query efficiency, database connections, and more.
- **Cloud provider monitoring tools:** Most cloud providers include free monitoring for your database infrastructure to show metrics like CPU usage and network transfer. Some cloud providers also offer database monitoring as part of their managed database service.
- **External monitoring tools:** Database-specific monitoring tools are designed to provide more insight beyond metrics like your underlying infrastructure's CPU usage or network traffic. There is a wide variety of these tools available that fit different workloads. Here are a few examples of free open source database monitoring tools.
  - **Prometheus & MySQL Exporter:** Built on top of the popular open source infrastructure monitoring tool, Prometheus, MySQL Server Exporter allows you to create a series of collector flags.
  - **VictoriaMetrics:** A time series database and monitoring solution to help process real-time metrics using the PromQL, MetricsQL, and Graphite query languages. Best suited for small to medium size database environments. [Deploy via the Linode Marketplace.](#)
  - **Percona Monitoring & Management:** Optimize database performance and track behavior patterns for MySQL, PostgreSQL, and MongoDB. [Deploy via the Linode Marketplace.](#)

### *When should you separate your database monitoring layer?*

If you just want to keep an eye on CPU usage, relying on more standard infrastructure monitoring solutions will suffice. If you want to be able to observe performance at the query or table level, or check database logs during very specific points in time, an external monitoring tool will give you additional insight that can benefit your application.



Section 3

# Deploying a Database with a Managed Database Service

## Section 3

# Deploying a Database with a Managed Database Service

As shown in the previous section, there are many different ways you can deploy databases in the cloud and choose between automated or manual maintenance. Users and customers have come to expect high-performance applications as the norm, and a managed solution helps make this faster and simpler to achieve. As a result, many cloud providers include a managed database service as part of their core offerings, also referred to as Database as a Service (DBaaS).

### What is a Managed Database?

A managed database service offers the ability to deploy a database with just a few clicks in a cloud provider's UI, via their command line interface, or by an API request. The service shifts some of the database administration and management responsibilities to the cloud provider. Notably, DBMS version updates are performed by the cloud provider, which is important because missing patches or database version upgrades can create security vulnerabilities. As a result, database security is shared between the cloud provider and the developer when using a managed database service.

Easily deploying a highly-available database via a managed service is a significant benefit for all kinds of workloads and applications. A highly-available database is typically a cluster of three nodes, which is composed of a primary database node and two replicas. While undergoing maintenance or in case of node failure, there is another node available to make sure your application doesn't experience downtime.

Across providers, a standard managed database service includes:

- automated major and minor database version upgrades;
- automated operating system patches;
- automated backups that are included as part of the service;
- easy access controls to restrict or open traffic to the database nodes;
- options to create a highly available database cluster; and
- easy database node resizing so you can scale your database resources up or down as needed.

Depending on the cloud provider, a managed database service could include additional tools, including private networking and custom database monitoring alerts.

In general, managed database services allow you to rapidly scale your database with minimal time spent on administration. As a result, you can spend more time tuning and developing your application.

## Migrating to a Managed Database

You can move existing databases from on-prem or another cloud provider to a new cloud provider. This can be fairly seamless with a little due diligence before initiating your migration.

- **Consider dependencies:** Databases are tied to applications, which, in turn, can be tied to other applications in your environment. Consider dependency mapping if you're not moving your entire IT ecosystem in unison.
- **Compute resource compatibility:** Check to make sure the provider's compute plans and additional products (like firewalls or load balancers) are compatible with the managed database service. If they are not compatible, establish alternatives for those services *before* migrating.
- **DBMS version support:** In addition to making sure your preferred DBMS is supported by your cloud provider's DBaaS, check which **versions** it supports. You might need to upgrade your existing database to a provider-supported version before performing the migration. When upgrading a database to a new version, perform testing to make sure that your existing database features are compatible with the new version.
- **Network transfer allotment:** Depending on the size of your current database and the network transfer provided by the new cloud provider, migrating a database can be a hefty expense. Carefully examine how network transfer is calculated, how much is included, and if the cloud provider has any special migration assistance to reduce (or even eliminate) this upfront cost. Make sure to also estimate any outbound network transfer costs from your original on premise deployment or cloud provider.
- **Steps to connect your new database to your application:** If your current database configuration was set up in the past and not documented, make a list of the necessary tasks to point your application's DB connection to your new database deployment. Having this information documented will help mitigate user impact and avoid downtime. These steps will vary depending on the rest of your application's architecture.

Choosing between a cloud provider's managed database service and a self-managed database deployment comes down to application requirements and personal preference. If you don't need to maintain a specific database version or use a specific operating system, paying for a managed service can save you time and effort in the long run. As we reviewed in Section 3, there are many ways to set up your database architecture depending on your application requirements and how you want to prioritize your development time.



Section 4

# **Managed Databases and the Alternative Cloud**

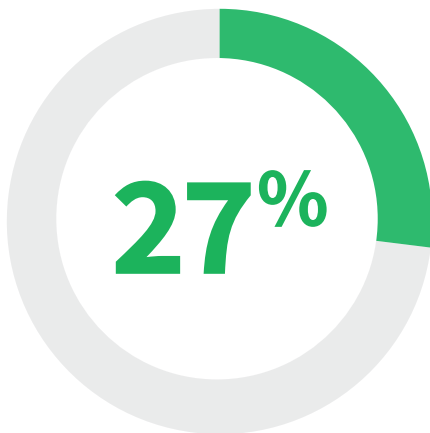
## Section 4

# Managed Databases and the Alternative Cloud

Alternative cloud providers have become credible competition for Amazon Web Services (AWS), Google Cloud Platform (GCP), and Microsoft Azure. These cloud providers have historically maintained an oligopoly over the cloud market. The “Big 3” are now being challenged by smaller alternative cloud providers including Linode (now part of Akamai), DigitalOcean, and Vultr. Alternative cloud providers **now command one-third of all spending on cloud services**, according to Techstrong Research (April 2022).

Alternative providers are comparable in terms of services offered, global availability, and more competitive pricing. The overall market is responding positively to the viability of alternative providers. This is especially true for small businesses and independent developers who don’t want or need the mass amounts of auxiliary and proprietary services offered by the Big 3.

What makes alternative cloud providers a growing threat to the Big 3? They focus on platform simplicity with core cloud offerings, human-centric support, and competitive pricing. According to a recent SlashData survey, usage of alternative cloud providers has nearly **doubled** over the past four years, while usage of the Big 3 (AWS, Azure, GCP) only grew 18%.



“27% of 3,500 developers surveyed by SlashData use an alternative cloud provider like Linode, DigitalOcean, or OVHcloud.”

*SlashData Developer Nation Survey, 2022*

Recent high-profile outages from hyperscale cloud providers have created devastating impacts for production applications. These outages have driven developers and businesses to consider alternative cloud providers and to adopt a multicloud strategy to better protect their applications.

## Cloud Provider Considerations for Databases

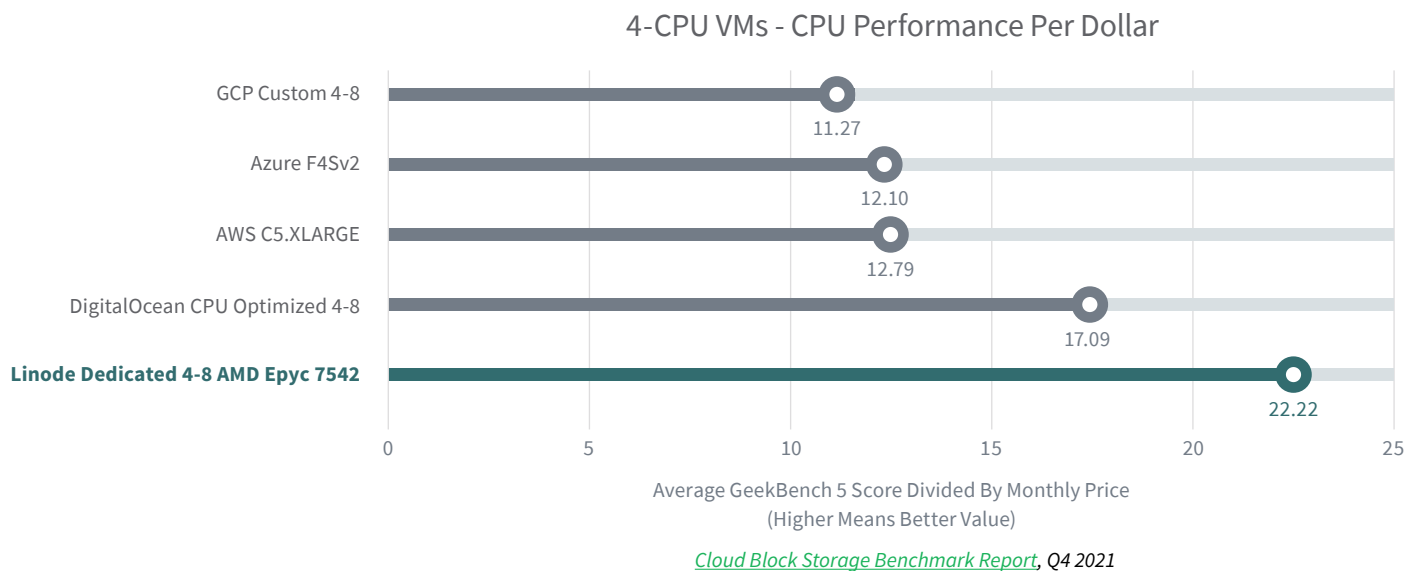
Cloud database performance starts at the hardware level. If you're trying to fine-tune your current database performance or future-proof your architecture for anticipated resource needs, your hardware specification needs to account for more than just memory allocation and deciding between dedicated and shared CPU cores. Evaluating your cloud provider's class of hardware, estimating associated costs, and assessing the risk of proprietary database services are important parts of performance planning.

The Big 3 cloud providers are excellent at finding a pain point for database deployment or management. These pain points are turned into additional paid services or irresistible features for their proprietary (and usually expensive) managed database service. Some workloads definitely benefit from this; for example, a MySQL database deployed using AWS Relational Database and AWS Redshift can provide faster access to business insights. Also, while you can easily find another cloud provider that provides managed MySQL database clusters, AWS provides solutions for more wide-ranging storage architectures, like data warehouses.

If you're looking for a reliable cloud provider to host an industry-standard database as a managed or unmanaged deployment, and your use case doesn't require an enterprise-level tool (like the AWS Redshift example), it's likely that choosing one of the Big 3 won't be worth the premium cost.

You can access the same (or better) hardware and network infrastructure with alternative cloud providers for a fraction of the cost. Independent cloud benchmarking firm Cloud Spectator consistently finds that Linode and other alternative cloud providers outperform the Big 3 in price-performance and, in many cases, overall performance.

Across providers, a standard managed database service includes:



Knowledge is power. Knowing exactly what to shop for as a cloud consumer and how alternative cloud providers compare is the key to building more performant and cost-efficient applications.



## What to Look For When Choosing an Alternative Cloud Provider

- **Core Cloud Offerings:** Strong selection of basic compute and cloud storage plans, plus supported integrations with industry-standard tools like Kubernetes, Terraform, and more.
- **Powerful Hardware:** A reputable alternative provider will have comparable or better CPUs, storage, and GPUs when lined up against hyperscale offerings.
- **High Service Level Agreement:** Look for at least 99.99% guaranteed uptime SLA and a public statement about data rights that's easy to find.
- **Global Footprint:** Look at the alternative provider's current data center locations and expansion roadmap, how their pricing differs for each data center, and how network traffic is routed between data centers.
- **24/7/365 No-Tier Support:** A lack of reliable and accessible technical support is a huge factor for why developers with smaller workloads leave the hyperscale providers. Before making the switch or starting new services, check the provider's support services and search for loopholes.
- **Extensive Documentation:** A provider should have thorough and easy-to-follow documentation on their platform, as well as more general tutorials on your database of choice, Infrastructure as Code tools, and the other tech in your stack.
- **Security and Compliance Information:** Strict security requirements for data centers, networking, and security-focused products are critical checkboxes when considering alternative cloud providers.

A managed database service is a great fit for use cases that require a relatively hands-off approach to database maintenance and uptime. If your application doesn't require a proprietary database service or other tool from a specific cloud provider, looking to the alternative cloud for a managed service is a cost-effective and risk-averse approach to your database deployment.

Section 5

# Our Take

## Section 5

# Our Take

Performant and secure applications are critical to business growth and continuity and must run in tandem with highly-tuned databases. To keep databases running fast and smoothly, every level of your infrastructure must be considered, from hardware specs to consistent patching and maintenance. Fortunately, you no longer need to be a DBA to manage a performant- and highly-available database.

The advancement of cloud services and open source tools makes it possible for any developer to streamline the deployment and maintenance of a database. By eliminating database management overhead, you have more time to develop innovative applications.

When considering different cloud databases, developers can “choose their own adventure” from a range of options tailored to their needs and preferences. Alternative cloud providers make this endeavor more accessible and cost-effective, and Linode empowers developers to focus on their code and build feature-rich applications, instead of server and database administration. Actively expanding your knowledge of databases will help you make the right decisions for your application, ensuring that you meet your users’ expectations for performance, uptime, and security.

## Extended eBook

# Deploy Django to Linode using Managed Databases for MySQL

cfe

dj

MySQL™

**GET YOUR COPY**



## Section 6

# About

# About the Instructor

Justin Mitchel is a father, coder, teacher, YouTuber, best-selling Udemy instructor, and the founder of Coding for Entrepreneurs. Justin also the author and host of the [Try IaC eBook and video-series](#). Connect with Justin on Twitter [@justinmitchel](#).

# About Linode

Akamai's Linode cloud is one of the easiest-to-use and most trusted infrastructure-as-a-service platforms. Built by developers for developers, Linode accelerates innovation by making cloud computing simple, affordable and accessible to all. Learn more at [linode.com](#) or follow Linode on [Twitter](#) and [LinkedIn](#).



# Part 2



# Welcome

This book explores how to sustainably and efficiently deploy Django into production on Linode. Each chapter with step-by-step instructions goes with production-ready code available on our [GitHub](#).

Since this book is about deploying Django into production, we'll limit the amount of *manual* work and opt for as much *automation* as possible. To do this, we'll focus on these core areas:

- CI / CD with Git, GitHub, and GitHub Actions
- Django on Docker & DockerHub (as well as using WatchTower)
- Load Balancing with NGINX
- Production Databases with Managed MySQL by Linode
- Local/Development use of production-like databases
- Terraform to provision infrastructure on Linode
- Ansible to configure infrastructure on Linode in tandem with Terraform
- Django-based file uploads & Django static files on Linode Object Storage

This book focuses on getting Django into production, but most of what we cover can be adapted for many other technologies as well. The reason? Our Django project will run through Docker with Docker containers. Docker is a massive key to putting many, many different applications in production. It's a great tool, and we'll use it as the backbone that Django runs on.

Before all the branded names for the tech stack start to scare you off, I want you to remember this who thing is just a *bunch of documents* and nothing more. Just because it's called **Docker** or **Python** or **Terraform**, they are still all just documents that describe... something.

With that in mind, I want you to think of the above list like this:

- ❑ **GitHub:** A place to store our code, *almost* like a folder on your computer
- ❑ **GitHub Actions:** An app that automatically starts a chain of events related to testing (verifying the document does what the writer intended), building (let's put a stamp on this code and say that this version is done), and running our code in production (find the code that should run the best, is ready, and let's and run that)
- ❑ **Django:** A bunch of python documents to handle the logic of an HTML website, store data, and handle user data
- ❑ **Docker:** Essentially a document that we write to describe, on the operating system level, how our Django app needs to run. Docker itself does more than this, but we mostly work on the document part (a **Dockerfile**)
- ❑ **Load Balancing with NGINX:** A document that we write to tell **nginx** to send incoming website traffic to the different servers running our Docker-based Django app
- ❑ **Managed databases:** a powerful spreadsheet that you never need to open or touch, thanks to Linode and Django
- ❑ **Terraform:** A document describing all of the products and services we need from Linode (like virtual machines) to run our Django project
- ❑ **Ansible:** A few documents we use to tell all of the products and services what software our Django project needs to run
- ❑ **Linode Object Storage:** A file system (sort of) that allows us to store files that don't change very often like images and videos, as well as other documents like CSS and some kinds of JavaScript

If you read the above and thought to yourself, “well that’s overly simplistic” -- **you’re right!**

If you read the above and thought to yourself, “oh no, what am I doing with my life” -- **you’re right!**

The nice thing about what we’re doing here is taking the guesswork out of making this list work, work correctly, and work reliably. Most of the technology we will cover here has been well established and will likely remain unchanged for the foreseeable future.

# Requirements

## Software

Are you new to Python? Watch *any* of the video series on <https://cfe.sh/topics/try-python>

Are you new to Django? Watch *any* of the video series on <https://cfe.sh/topics/try-django>

That's it. You just need some Python experience and some Django experience to do this book. If you're new at both and you do this book anyways, let me know how it goes: <https://twitter.com/justinmitchel>

I'm a big fan of Just In Time learning. That means learning a concept right before you need to use it. My parents literally named me after this principal. **That's obviously not true, or is it?**

## Hardware

- ☐ Access to the internet
- ☐ At least a \$25 Raspberry PI with a keyboard, mouse, and screen. Using a tablet or smartphone *can* be done, but it's a lot more difficult.

It's important to note that almost all of this *can* be done just using GitHub and Linode, although I recommend a computer where you have root access (admin access).

If you have access to the internet, you can use the in-browser coding experience on GitHub. **What's that, you say?** Just press period ( . ) on any GitHub repo, and it will give you the ability to write code inside of an in-browser code editor.

## Where to Get Help

In this book, we use two different repos:

1. <https://github.com/codingforentrepreneurs/cfe-django-blog>
2. <https://github.com/codingforentrepreneurs/deploy-django-linode-mysql>

The first repo is what we will be using as our base Django project. This Django project will evolve outside the context of this book.

The second repo will be the final state as it relates to this book. This repo will only change **if** there are errors or updates needed with the book and/or videos series.

We have all of our projects open-sourced on [codingforentrepreneurs GitHub](#) so you can use that as a resource to learn from as well!

## Asking Questions

The best way to get help is to formulate the question you have to be as clear and non-specific as possible. Let's say you get this error:

```
Traceback (most recent call last):
  File "manage.py", line 12, in main
    from django.core.management import execute_from_command_line
ModuleNotFoundError: No module named 'django'
```

Before you ask for help, consider asking yourself the following:

- ☐ What did I run to get here?
- ☐ Did it work before? If so, did I skip a step to make it work?
- ☐ If someone knew nothing about my project, could they answer what is going on with this error?
- ☐ Did I quit my current session? Restart my computer?
- ☐ Should I just google the error I see `ModuleNotFoundError: No module named 'django'` ?
- ☐ What are 3 or 4 ways that I can ask these questions?

I have found that asking myself the above questions often *solves* the problem for me, but most of the time, I find solutions in two places: Google and StackOverflow.

## Asking Questions (For Real)

Aside from trying to self-diagnose the solution, asking for help often forces you to reframe the question and thus find an answer during that process. I encourage you to ask questions whenever you get stuck. I don't love answering the same question over and over, but when I do, I realize that it's an opportunity for me to explain a given concept or problem better.

This book is the result of me getting a *lot* of questions over the years. These questions are asked both by students I've taught, questions I've asked myself, and questions that other people are asking other people.

Questions are the foundation for finding more questions. Sure, we get answers along the way, but more and better questions are ultimately about how we find excellence and perhaps meaning, in our lives.

Ask a better question, get a better answer.

Let's not forget that the answer to the question in the previous section is simple: either Django is not installed, or you didn't activate the virtual environment, or both! Pat yourself on the back if you knew this, or pat yourself on the back because you now know this. Either way, give yourself a pat on the back.

# Part 2: Table of Contents

<b>Chapter 1: Getting Started . . . . .</b>	<b><u>01</u></b>
<b>Option 1. Clone a Pre-existing Django Project</b>	<b><u>02</u></b>
1. Clone Repo	<u>02</u>
2. Create the Virtual Environment, Activate it, and Install Requirements	<u>03</u>
3. Add Default Environment Variables	<u>03</u>
<b>Option 2. Create a New Django Project</b>	<b><u>05</u></b>
1. Install Python 3.10	<u>05</u>
2. Create a Project Directory	<u>05</u>
3. Create a Virtual Environment	<u>05</u>
4. Activate your Virtual Environment	<u>06</u>
Alternatives to Activating your Virtual Environments	<u>07</u>
5. Initialize Git	<u>08</u>
6. Install Requirements	<u>10</u>
Upgrade <code>pip</code>	<u>11</u>
Install via <code>requirements.txt</code>	<u>11</u>
Verify	<u>11</u>
7. Start a Django project	<u>12</u>
8. Clone Project	<u>14</u>

<b>Chapter 2: Git &amp; GitHub Actions</b> .....	<b>15</b>
Assumptions	17
Remove Cloned Repo	17
Create a New GitHub Repository	17
1. Log in to your GitHub account	17
2. Click <b>+</b> > <b>New Repository</b> or follow this link	17
3. Copy URL as your new remote origin	18
4. Update your local remote	18
5. Push your code	18
6. Automation has already started!	19
Create a GitHub Action Workflow	19
1. GitHub Action Basic Workflow Format	19
2. GitHub Action Triggers	22
3. Test Django Workflow	23
Create (or verify) the <b>.github/workflows</b> folder exists:	23
Replace the <b>test-django-mysql.yaml</b> file	23
Add, commit, and push your code	27
Open GitHub Actions	27
Running Workflows in Workflows	28
1. A workflow for workflows	28
2. The CI/CD Pipeline <b>all.yaml</b> Workflow	28
3. Push <b>all.yaml</b> to GitHub	29
4. Updating <b>all.yaml</b> Workflow	31
Using GitHub Repository Action Secrets	32
Sharing secrets on Shared Workflows	34
Update <b>all.yaml</b> With Shared Secrets	35
Act: run your GitHub actions anywhere	36
 <b>Chapter 3: Containerize Django With Docker</b> .....	 <b>37</b>
Is Docker easy?	39
1. Install Docker Desktop	40
2. Dockerfile for production with Django	40
Remove cloned DOckerfile and Docker Compose File	41
Production-ready Docker File	41
3. Running Django in production: <b>Gunicorn</b>	43
4. Create the entrypoint file ( <b>config/entrypoint.sh</b> )	45
5. <b>.dockerignore</b>	47
6. Build and run container	48
Basic workflow to build a Docker container	50
7. Using Docker Compose	51
Docker Compose for Django development	51
8. Push and host on Docker Hub	59
Sign up for Docker Hub on hub.docker.com/	59
Create a repository here	60
Create a Docker Hub API access token key	61
9. Create GitHub action workflow	62
10. Update <b>all.yaml</b> workflow	64



<b>Chapter 4: Linode Managed Database MySQL for Django . . . . .</b>	<b>65</b>
1. Create a MySQL Cluster on Linode	66
Login to Linode Cloud Manager	66
Navigate to Databases	66
Click <b>Create Database Cluster</b>	66
Once Provisioned, Download <i>Database CA Certificate</i>	67
2. Install MySQL client locally	68
Install MySQL client on MacOS	68
Install MySQL client on Windows	68
Install MySQL client on Linux	69
3. Use MySQL client to create Database & Database User	69
Connect to your Database	69
Create Database & Database User within the MySQL shell	71
Verify user permissions	71
4. Django test database in MySQL	73
5. Required SQL commands reference	75
6. Update <b>.env</b> or Environment Variables for your Django project	76
7. Django commands	78
Run, migrate, and verify Django	78
Load fixtures	78
Run the server	78
Run tests	79
Login to the admin	79
8. Fresh start	80
Create a backup:	80
1. Drop database in MySQL	81
2. Review MySQL	81
3. Re-create database	82
4. Go back to <b>7. Django Commands</b> section and repeat	82
 <b>Chapter 5: Managed MySQL &amp; Django in Production . . . . .</b>	 <b>83</b>
Create a new MySQL cluster	84
Required GitHub action secrets for MySQL database configuration	86
Configure MySQL database with GitHub Actions Workflow	87
Solving errors	92
From one MySQL database to another with GitHub Actions	92
Practical notes on using Terraform for database creation	94
Next steps with MySQL	94

<b>Chapter 6: Linode Object Storage for Django . . . . .</b>	<b>95</b>
Related resources	96
Motivation and background	96
1. Development static file serving	97
2. Production static file serving	98
Install <code>django-storages</code>	98
Environment variables configuration	98
3. Set up Linode Object Storage	99
4. Minimal configuration for <code>django-storages</code>	102
5. Advanced configuration for <code>django-storages</code>	103
Customize the <code>django-storages</code> backends	103
Access control list(s) <code>acl</code> Mixin	105
Configure Linode module	109
Final settings for advanced configuration	110
6. GitHub Actions for Django & Object Storage	111
7. Update <code>all.yaml</code> workflow in GitHub Actions	112
 <b>Chapter 7: Terraform &amp; Provision Infrastructure . . . . .</b>	 <b>113</b>
1. Install Terraform	115
2. Dedicated Terraform and Ansible folders	115
3. Setup Linode Object Storage for Terraform	116
4. Create <code>backend</code> and init project	118
5. Initialize Terraform	119
6. Provision options & <code>terraform.tfvars</code>	120
Create <code>terraform.tfvars</code>	121
Create a Linode Personal Access Token ( <code>linode_pa_token</code> )	122
Create new SSH keys for GitHub Actions and Terraform	124
Generate a root user password for instance(s)	125
Docker-base Django virtual machine count	125
7. Create Terraform files	126
<code>main.tf</code>	128
<code>variables.tf</code>	129
<code>locals.tf</code>	130
<code>linodes.tf</code>	132
<code>outputs.tf</code>	138
Templates	138
8. Run Commands	144
Validate Terraform project files	144
Apply/Update/Create Terraform resources	144
Automatically Apply/Update/Create Terraform resources	144
Destroy Terraform resources	145
Automatically Destroy Terraform resources	145
Using the console	145
9. Custom domain (Optional)	146
10. Part 1: GitHub Actions for Infrastructure	147

<b>Chapter 8: Ansible &amp; Configure Infrastructure. . . . .</b>	<b><a href="#">149</a></b>
0. The Ansible Inventory file	<a href="#">151</a>
Feel Like Skipping Terraform?	<a href="#">152</a>
1. Ansible Basics	<a href="#">153</a>
2. Install Ansible	<a href="#">154</a>
3. Create a Playbook	<a href="#">154</a>
3. Ansible Variables	<a href="#">156</a>
4. Ansible Configuration File	<a href="#">157</a>
5. Ansible Docker Role	<a href="#">158</a>
Local testing	<a href="#">160</a>
Understanding our <code>docker-install</code> role:	<a href="#">162</a>
<code>docker-install</code> handlers:	<a href="#">165</a>
6. Ansible Templates	<a href="#">166</a>
7. Ansible & Docker-based Django App on Docker Hub	<a href="#">169</a>
Create The <code>django-app</code> Role	<a href="#">169</a>
Add <code>django-app</code> Role Tasks	<a href="#">170</a>
Update Django <code>settings.py</code>	<a href="#">171</a>
Add <code>django-app</code> Role Handler	<a href="#">172</a>
Update <code>main.yaml</code>	<a href="#">173</a>
Playbook for force Docker Container rebuild	<a href="#">174</a>
The magic of Watchtower	<a href="#">175</a>
8. Ansible Load Balancer role	<a href="#">176</a>
Create The <code>nginx-lb</code> role	<a href="#">176</a>
Add The <code>nginx-lb</code> tasks	<a href="#">176</a>
Add the <code>nginx-lb</code> handlers	<a href="#">177</a>
Update <code>main.yaml</code>	<a href="#">178</a>
9. Part 2: GitHub Actions for Infrastructure	<a href="#">179</a>
Part 2: Update <code>.github/workflows/all.yaml</code> to support Ansible workflow	<a href="#">183</a>
10. Future considerations	<a href="#">185</a>
Command shortcuts with Make & a Makefile	<a href="#">185</a>
Environment variable updates	<a href="#">185</a>
Making model changes	<a href="#">185</a>
Custom domains with Linode, Certbot, Let's Encrypt	<a href="#">185</a>
 <b>Chapter 9: Final Thoughts . . . . .</b>	 <b><a href="#">189</a></b>
 <b>Appendix. . . . .</b>	 <b><a href="#">191</a></b>
Appendix A: GitHub Actions Secrets Reference	<a href="#">192</a>
Appendix B: Final Project Structure	<a href="#">193</a>
Appendix C: <code>.gitignore</code> and <code>.dockerignore</code> Reference examples	<a href="#">196</a>
Appendix D: Self-Hosted Runners with GitHub Actions	<a href="#">201</a>
Appendix E: Using the GitHub CLI for Setting Action Secrets	<a href="#">203</a>



# Chapter 1

# Getting Started

## Chapter 1

# Getting Started

As with most web applications, I like starting with the application itself. This is certainly the most fun as well as the part you'll modify the most over the lifetime of the project. Applications are what you get to change constantly and evolve to customers' and users' requests. Infrastructure has a keyword in its *structure*, which means it's stable. Or at least stable as in not changing.

Soon enough, we'll be spending a lot of time ensuring our CI/CD pipeline is set so that our infrastructure is set. For now, let's get our Django application in play.

To get started, we have two options:

- *Option 1* Clone a pre-existing Django Project
- *Option 2* Create a fresh Django project

## Option 1. Clone a Pre-existing Django Project

How many of you reading this are actually going to choose *Option 2*? You will probably choose *Option 3* "I'll use my own project" -- normally I am in that camp with you, but I think it would be a good idea for us to be on the same page while putting this Django project into production.

Taking this route assumes that you already how to build a Django project and use a virtual environment. If you don't, skip to *Option 1* further in this chapter.

### 1. Clone Repo

```
git clone https://github.com/codingforentrepreneurs/cfe-django-blog
cd cfe-django-blog
```

## 2. Create the Virtual Environment, Activate It, and Install Requirements:

### Create

```
python3.10 -m venv venv
```

Don't use built-in venv? Know how to create your own virtual environment? Do it. I can't see what you're doing.

### Activate on macOS or Linux

```
source venv/bin/activate
```

### Activate on Windows

```
.\venv\Scripts\activate
```

### Install

```
$(venv) python -m pip install pip --upgrade  
  
$(venv) python -m pip install -r requirements.txt
```

## 3. Add Default Environment Variables:

1. Create `.env` file

```
echo ".env" >> ""
```

## 2. Add environment variables

```
# required keys
DJANGO_DEBUG="1"
DJANGO_SECRET_KEY="gy_1$n9zsaacs^a4a1&-i%e95fe&d3pa+e^@5s*tke*r1b%*cu"
DATABASE_BACKEND="postgres"

# mysql db setup
MYSQL_DATABASE="cfeblog-m-db"
MYSQL_USER="cfeblog-m-user"
MYSQL_PASSWORD="RaSNF5H3ElCbDrGUGpdRSEx-IuDzkeHFL_S_QBuH5tk"
MYSQL_ROOT_PASSWORD="2mLTcmdPzU2LOa0TpAlLPoNf1XtIKsKvNn5WBiszczs"
MYSQL_TCP_PORT=3307
MYSQL_HOST="127.0.0.1"
```

You *should* change these values as needed. `DJANGO_DEBUG` *must not be* `1` in production.

## 3. Test variables are working

```
$(venv) python
```

```
>>> from dotenv import load_dotenv
>>> load_dotenv()
>>> import os
>>> assert os.environ.get("DATABASE_BACKEND") == "postgres"
```

You *should not* see the following:

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AssertionError
```

The `python-dotenv` package is how we're loading the virtual environments.



## Option 2. Create a New Django Project

This is for all you newbies out there that are looking to cross the chasm and get your Python skills to the next level. Let's create a blank Django project so the rest of this book will build on top of that blank project.

If you are new to Django and you're doing this chapter, I applaud you. To me, getting your project into production is just an amazing feeling and one worth celebrating. Give yourself a high five.

Did I mention this book is step-by-step?

### 1. Install Python 3.10

Do you have Python 3.10 installed? If not, go to [python.org](https://python.org) to install.

If Python 3.10 is not available on your machine, you can use Python 3.7 or later although some packages may not be supported in older versions of Python.

### 2. Create a Project Directory

```
mkdir -p ~/dev/cfeproj
```

In this case, my project will live at `/Users/cfe/dev/cfeproj` or `C:\Users\cfe\dev\cfeproj` depending on your system. This directory (folder) is the **root** of your project.

### 3. Create a Virtual Environment

Python virtual environments exist to isolate your python projects from one another so code versions do not cause conflicting issues. For this book, I will be using the built-in `venv` module. If you prefer `pipenv`, `poetry`, `virtualenv`, `virtualenvwrapper`, or any other virtual environment manager, you can use those. This book just assumes you have a virtual environment running and installed for your project.

First, navigate to the **root** of your project (in my case `cd ~/dev/cfeproj` )

**macOS/Linux**

```
python3.10 -m venv venv
```

**Windows**

```
C:\Python310\python.exe -m venv venv
```

`C:\Python310` is not always going to be the default location of Python 3.10 if you install from [python.org](https://python.org).

## 4. Activate your Virtual Environment

Every time you want to work on your project, you *must* activate the virtual environment:

**macOS/Linux**

```
source venv/bin/activate
```

**Windows**

```
.\venv\Scripts\activate
```

After you activate your virtual environment it should resemble:

**macOS/Linux**

```
$(venv)
```

**Windows**

```
(venv)>
```

From here on out, you can use just `python` as your command:

### macOS/Linux

```
$(venv) python -V
```

### Windows

```
(venv)> python -V
```

## Alternatives to activating your virtual environments

To best contradict myself, you absolutely *can* work on your project without activating your virtual environment. The trick is to use the `venv/bin` for the various commands you may have. A few examples would be:

- `venv/bin/python -m pip install pip --upgrade`
- `venv/bin/python manage.py runserver`
- `venv/bin/django-admin startproject cfeproj .`
- `venv/bin/gunicorn cfeproj.wsgi:application`

Do you see a pattern? Activating your virtual environment removes the need to write `venv/bin` every time you need to run a virtual-environment-only command.

## 5. Initialize Git

Version control is an absolute must and `git` is the best way to do it. This book was written using Git.

What is Git? Basically, a ~~nerdy~~ effective way to keep a record of file changes. Here's a rough example of a workflow:

1. Open a file.
2. Type something.
3. Save the file.
4. Tell Git about it.
5. Delete the file.
6. Ask Git for it, the file is back.
7. Open the file. Add stuff.
8. Tell Git about it.
9. Share the file with a friend.
10. Friend ruins file and tells Git about it.
11. You un-friend friend.
12. Ask Git to revert the file back to before you gave it to the friend. Git does.

The key is, that Git tracks changes to files (if you tell it to) and allows you to look at those files at any state in the past. It's neat.

Version control can get very complex especially when you're a large enough team and/or large project. In this book, I will not be touching on how complex **Git** can get. I'll mostly follow the lame example I laid out above.

Ready? Now go install `git` at: <https://git-scm.com/>

It works for macOS, Windows, and Linux.

Assuming you have installed git, let's run:

```
$(venv) git init
```

`git init` creates what's called a "repository" in the folder you call it in. This is often referred to as a `repo`. This repo is what tracks all of our changes over time. If you are super curious, take a look in the `.git` folder inside your project right now.

Now we can track our project pretty simply with the following commands:

1. `git add .`
2. `git commit -m "your message about the changes"`
3. `git push --all`

These three steps mean:

1. I made changes to my current project's code ( `git add .` or `git add --all` or `git add path/to/some/file` or `git add path/to/some/dir/` )
2. I am *sure* I want them tracked by Git and here's what I did "your message about the changes" ( `git commit -m "some msg"` )
3. I want this code pushed to a remote repo like GitHub. This push *often* means pushing this code into production. ( `git push --all` or `git push origin master` or `git push origin main` )

If you know `git` well, you'll know that what I wrote above is pretty simplistic and skips a lot about the power of using `git`. The point of this section is to help beginners get the job done.

If you are a beginner in `git` I recommend using any of the following tools:

- [VS Code](#) has built-in support for managing `git` and file changes. It's a lot more visual than the `git` CLI tool.
- [GitHub Desktop](#) I don't use this one myself but many, many developers do and rely on it to manage their `git` repos.

### Create `.gitignore`

It's important for `git` to ignore some files and some file changes. A `.gitignore` will do that for us.

For example, we do not want to track changes within `venv` or track files that contain passwords (like `.env` ) so we'll do the following:

```
cd ~/dev/cfeproj
echo "venv" >> .gitignore
echo ".env" >> .gitignore
echo "*.py[cod]" >> .gitignore
echo "__pycache__/" >> .gitignore
echo ".DS_Store" >> .gitignore
```

This is a cross-platform friendly way to add items to a file. Here's the resulting file:

**.gitignore**

```
venv
.env
*.py[cod]
__pycache__/
.DS_Store
```

There's a *lot* more we can add to a **.gitignore** file. Consider reviewing [[this GitHub Python .gitignore file](#)]

## 6. Install Requirements:

Okay, now that we have version control setup, we'll need to start adding project dependencies that python calls requirements (since it's often stored in **requirements.txt** ).

Here's what we're going to install:

- **django>=3.2,<4.0** : This gives us the latest long-term support version of Django (known as the most recent LTS)
- **gunicorn** : our production-grade web server
- **python-dotenv** : for loading environment variables in Django
- **black** (optional): for auto-formatting code
- **pillow** : also known as the Python Image Library mainly for the ImageField in Django
- **boto3** : for using Linode Object Storage with Python
- **django-storages** for using Linode Object Storage with Django (depends on **boto3** )
- **mysqlclient** : for connecting Django to a production-grade MySQL database.

To install these packages, we'll create `requirements.txt` in the **root** of our project with the following contents:

```
django>=3.2,<4.0
## our production-grade webserver
gunicorn
## loading in environment variables
python-dotenv
# for formatting code
black
## for image file uploads in Django
pillow
## for leveraging 3rd party static/media file servers
boto3
django-storages
## for mysql databases
mysqlclient
```

## Upgrade `pip` :

Pip likes to be upgraded from time to time. Let's do that now to start on a good note.

```
$(venv) python -m pip install pip --upgrade
```

## Install via `requirements.txt`

```
$(venv) python -m pip install -r requirements.txt
```

## Verify

```
$(venv) python -m pip freeze
```



This should respond with something like:

```
asgiref==3.5.0
black==22.3.0
boto3==1.21.28
botocore==1.24.28
click==8.1.0
Django==3.2.12
django-dotenv==1.4.2
django-storages==1.12.3
gunicorn==20.1.0
jmespath==1.0.0
mypy-extensions==0.4.3
mysqlclient==2.1.0
pathspec==0.9.0
Pillow==9.0.1
platformdirs==2.5.1
python-dateutil==2.8.2
pytz==2022.1
s3transfer==0.5.2
six==1.16.0
sqlparse==0.4.2
tomli==2.0.1
urllib3==1.26.9
```

Please note that some version(s) and packages *will* be different. The `requirements.txt` shows the requirements (aside from version numbers) that *must be* the same.

## 7. Start a Django project

It's time to finally start our Django project. Everything up to this point was for individuals new to this process, that's why the section on [Cloning a Pre-existing Django Project](#) was so short -- once you know how to get to this step, everything prior goes much faster.

Back in the **root** of our project with a virtual environment activated:

```
$(venv) django-admin startproject cfeproj .
```

Let's take a look at what our current directory looks like (without `venv`):

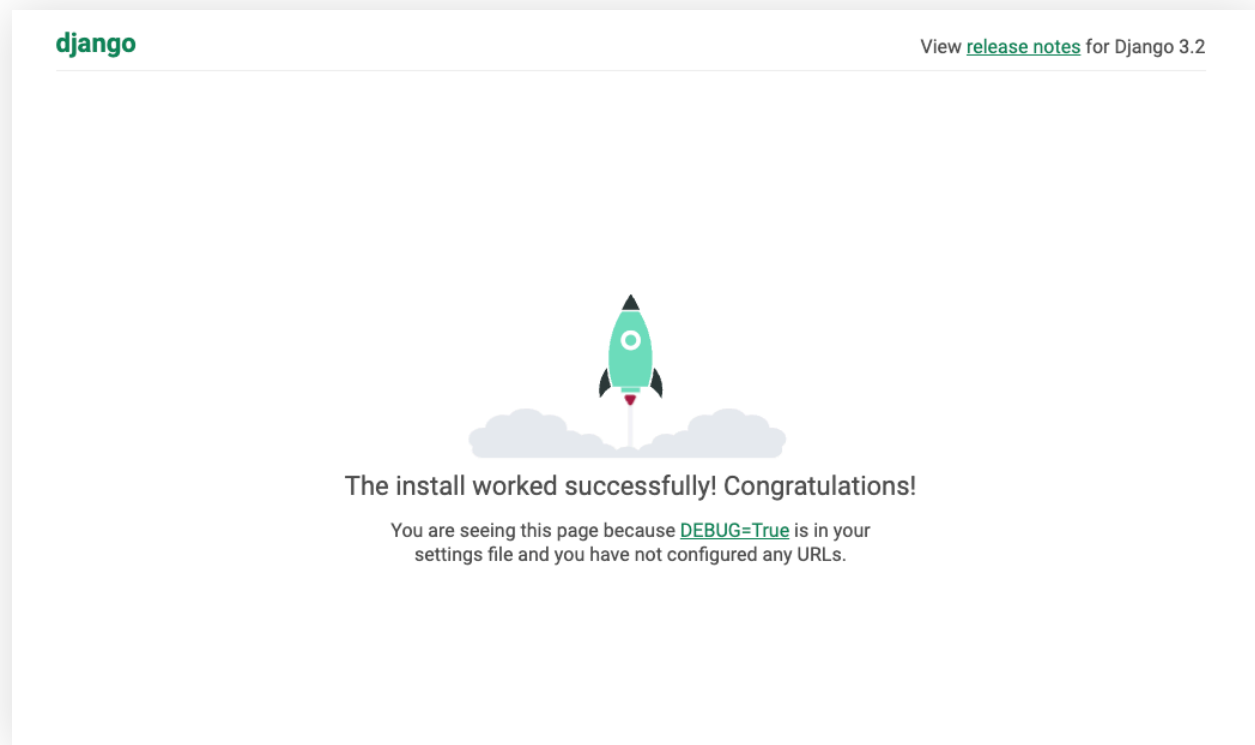
```
.
├── cfeproj
│   ├── __init__.py
│   ├── asgi.py
│   ├── settings.py
│   ├── urls.py
│   └── wsgi.py
├── manage.py
└── requirements.txt
```

The above file layout was created using `tree --gitignore` ( `brew install tree` )

### Boot it up

```
$(venv) python manage.py runserver 8881
```

Now open [localhost:8881](http://localhost:8881) and verify that you see:



## 8. Clone Project

Now that you can create a Django project from scratch, it's time to copy one that I have already created for you. This book isn't going to explore everything about Django, but it will give you a hands-on practical example of bringing a real Django project into production.

If you feel like you could use a bit more time learning the basics of Django (which I highly recommend if you're lost at all at this point) please consider watching any version of my Try Django series on <https://cfe.sh/topics/try-django> or my youtube channel <https://cfe.sh/youtube>.

Now, it's time to go back up to [Option 1: Clone a Pre-existing Django Project](#).

Chapter 2

# Git & GitHub Actions

## Chapter 2

# Git & GitHub Actions

As we write code, it's important to leverage version control, also known as `git`. If you don't have Git installed and set up on your local machine with your Django code, go back to [Getting Started](#) and locate the [initialize Git section](#).

Using version control allows us to move our code around and use services like GitHub, GitLab, Bitbucket, and many others. In this book, we're going to be working around GitHub and GitHub Actions.

GitHub is a place to store code. GitHub Actions is a place to run workflows & pipelines mostly based on this code. GitHub Actions enable us to do what's called Continuous Integration and Continuous Delivery (also known as CI/CD).

CI/CD is the practice of constantly improving your app by automating the stages of development and deployment.

For us, CI/CD means that we'll use GitHub Actions to run the following pipeline (in this order too):

1. Update/Store/Retrieve our Production Secrets & Environment Variables
2. Test our Django code against an Ephemeral Test MySQL Database ( `python manage.py test` )
3. Add our Django Static Files to Linode Object Storage ( `python manage.py collectstatic` )
4. Build our Docker container for our Django Application ( `docker build -t myuser/myapp -f Dockerfile .` )
5. Push our Docker container into DockerHub ( `docker push myuser/myapp --all-tags` )
6. Provision any infrastructure changes via Terraform ( `terraform apply -auto-approve` )
7. Configure our infrastructure via Ansible ( `ansible-playbook main.yaml` )
8. Promote our Docker-based Django app into Production via Ansible ( `ansible-playbook main.yaml` )

In this chapter we're going to focus on the following pieces of the above CI/CD pipeline:

- Creating a GitHub repo (repository)
- Adding/updating GitHub Secrets
- Creating our first GitHub Actions workflow for testing Django

The remainder of the book will be dedicated to building out our CI/CD pipeline in GitHub Actions.

## Assumptions

- You have Django code locally as we did in [this section](#)
- You have Git installed and initialized as we did in [this section](#)
- You already have a `.gitignore` as we did in [this section](#)
- You have a GitHub account (if not, sign up on <https://github.com>)

After we have code, it's important to start working with version control and GitHub. If you're not going to use GitHub my next recommendation would be GitLab. Doing this process in GitLab is similar, but we won't cover it in this book.

This chapter will set up an automated way to **test** our Django code after we push it to GitHub.

## Remove Cloned Repo

If you [cloned the recommended Django project](#), you should already have a remote repository attached to it. The following command will list any attached repositories:

```
$ git remote -v
```

We do not need this repo attached any longer, let's remove it:

```
git remote remove origin
```

## Create a New GitHub Repository

### 1. Log in to your GitHub account

### 2. Click `+` > `New Repository` or Go To [This Link](#)

## Options:

- **Repository name** : Pick a name or use `Deploy Django`
- **Public** or **Private** (I like to pick `private` unless I need to share)
- Leave everything else **blank** such as `README` , `.gitignore` , `license` because our cloned project in [Getting Started](#) already has these things.

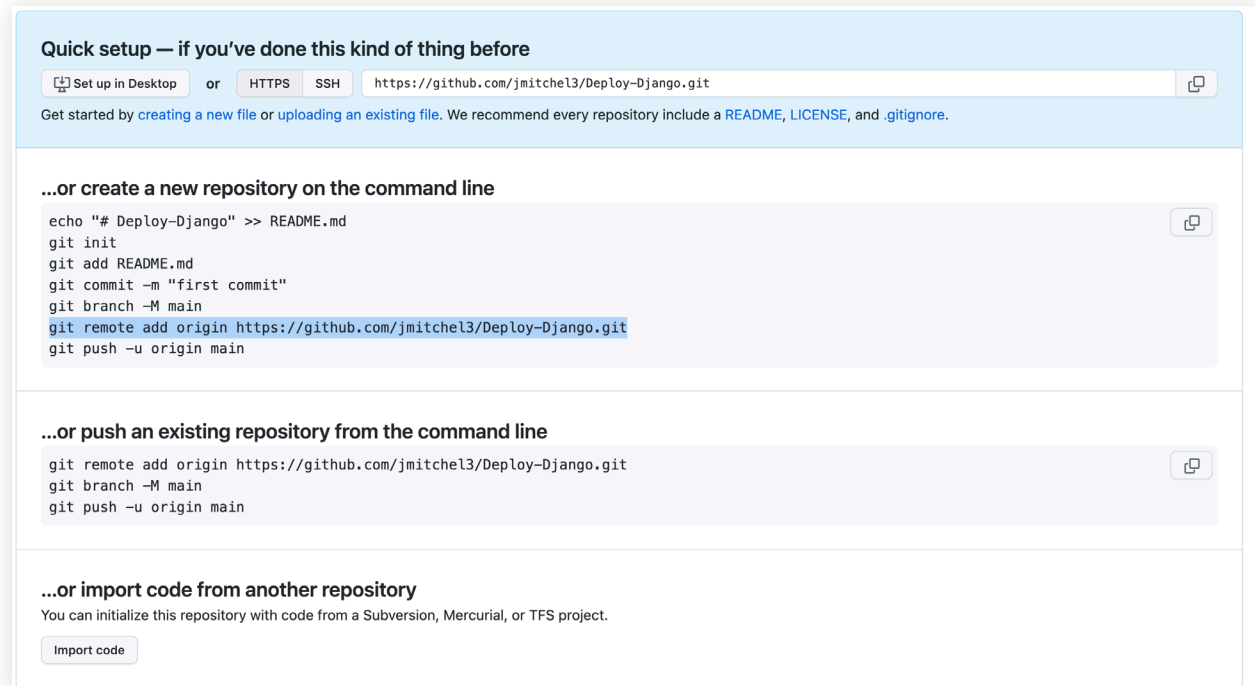
You can make this new repository private or public. I often stick to `private` for my production workflows that don't need to be exposed to the public.

### 3. Copy URL as your new remote origin

The new origin can be either of the following:

- `https://github.com/your-username/your-private-repo`
- `https://github.com/your-username/your-private-repo.git`

Typically when you leave `README`, `.gitignore`, and `license` blank, you should see something like this:



The screenshot shows the GitHub 'Quick setup' interface. At the top, it says 'Quick setup — if you've done this kind of thing before'. Below this, there are three tabs: 'Set up in Desktop', 'HTTPS', and 'SSH'. The 'HTTPS' tab is selected, and the URL 'https://github.com/jmitchel3/Deploy-Django.git' is entered in the text box. Below the text box, it says 'Get started by creating a new file or uploading an existing file. We recommend every repository include a README, LICENSE, and .gitignore.' Below this, there are three sections: '...or create a new repository on the command line', '...or push an existing repository from the command line', and '...or import code from another repository'. Each section has a code block with the necessary Git commands. The '...or create a new repository on the command line' section has a copy icon. The '...or push an existing repository from the command line' section has a copy icon. The '...or import code from another repository' section has an 'Import code' button.

```
Quick setup — if you've done this kind of thing before
```

Set up in Desktop or HTTPS SSH `https://github.com/jmitchel3/Deploy-Django.git`

Get started by creating a new file or uploading an existing file. We recommend every repository include a README, LICENSE, and .gitignore.

...or create a new repository on the command line

```
echo "# Deploy-Django" >> README.md
git init
git add README.md
git commit -m "first commit"
git branch -M main
git remote add origin https://github.com/jmitchel3/Deploy-Django.git
git push -u origin main
```

...or push an existing repository from the command line

```
git remote add origin https://github.com/jmitchel3/Deploy-Django.git
git branch -M main
git push -u origin main
```

...or import code from another repository

You can initialize this repository with code from a Subversion, Mercurial, or TFS project.

Import code

### 4. Update your local remote:

```
git remote add origin https://github.com/your-username/your-private-repo.git
```

### 5. Push your code

```
git push origin main
```

Using `git push --set-upstream origin main` is nice because then you can just run `git push` instead of `git push origin main`



## 6. Automation has already started!

If you go to your repo and then the **Actions** tab (under `https://github.com/your-username/your-private-repo/actions`) you **should** see a workflow already running.

Pretty cool huh? This is the magic of GitHub Actions.

## Create a GitHub Action Workflow

GitHub Actions are an amazing way to automate how we push our code into production. You can do *many, many* other things with GitHub Actions but we'll be focusing on pushing our Django project into production.

What exactly will GitHub Actions do for us? Here's an overview:

1. Test our Django code with a MySQL Database ( `python manage.py test` )
2. Add our static files to Linode Object Storage ( `python manage.py collectstatic` )
3. Build a Docker container for our Django app ( `docker build` )
4. Login & push our container to Docker Hub ( `docker push` )
5. Terraform to apply changes (if any) to infrastructure via our GitHub repo Secrets ( `terraform apply` )
6. We'll use Ansible to configure (or reconfigure) our infrastructure via our GitHub repo Secrets ( `ansible-playbook main.yaml` )

Each step above will be discussed in detail in future chapters so we'll skip a lot of the discussion and get right down to action, GitHub Action (Dad jokes for the win!).

My approach is to create 7 workflows to handle the 6 steps. Each step will have its own workflow with some overlaps as needed.

A workflow is simply a series of steps GitHub Actions needs to take. The format is in `yaml` so it might be a bit strange to you but I promise it will make sense after you use it for some time.

### 1. GitHub Action basic workflow format

Every GitHub Action workflow uses the `yaml` format. It is stored in `.github/workflows` right next to your code and your `.git` folder. The workflow file looks essentially like this:

```
yaml
name: Give me a Name, Any Name
```

## Controls when the workflow will run on

Allows you to call this workflow within another workflow

```
workflow_call:
```

## Allows you to run this workflow manually from the GitHub Actions tab and the GitHub API

```
workflow_dispatch:
```

## Triggered based on the Git event type

```
push:
  branches: [main]
pull_request:
  branches: [main]
```

A workflow run is made up of one or more jobs that can run sequentially or in parallel jobs:

## This workflow contains a single job called "my\_first\_workflow\_job"

```
my_first_workflow_job:
```

## The type of runner that the “my\_first\_workflow\_job” job will run on

```
runs-on: ubuntu-latest
```

## This allows you to change the directory you want this job to run in

```
defaults:
  run:
    working-directory: ./src
```

## Add in environment variables for the entire “my\_first\_workflow\_job” job

```
env:
  HELLO_WORLD: true
```

## Add in a service (like MySQL, Postgres, Redis, etc) you need running during your “my\_first\_workflow\_job” job

```
services:
  mysql:
    image: mysql:8.0.28
```

## Add other configuration items here

```
steps:
```

## Checks out your repository under \$GITHUB\_WORKSPACE, so your job can access it

```
- name: Checkout code
  uses: actions/checkout@v2
- name: Run Hello World
  run: |
    echo "hello world!"
```

Like with everything, I believe in Just-In-Time learning. GitHub Actions is just a tool that follows a `yaml` document's configuration. VS Code has GitHub workflow support. (GitHub workflow is the name of the document that defines a GitHub Action.)

As far as I can tell, you have an *unlimited* amount of workflow files in your GitHub repo. There is a limit to the number of *jobs* that you can run, but the number of files seems to be uncapped.

Understanding every nuance of the GitHub Action workflow file is not a trivial task. The GitHub Actions [documentation](#) is fantastic and worth a look.

## 2. GitHub Action triggers

What is likely going to trip you up is this block right here:

```
yaml
on:
  workflow_call:
  workflow_dispatch:
  push:
    branches: [main]
  pull_request:
    branches: [main]
```

These are all GitHub Action triggers. These signal to GitHub Actions when this trigger should run. There are [many more trigger options](#) but these, I would argue, are by far the most commonly used.

So what does each one mean?

- `on:` this declares the `trigger` block
- `workflow_call:` this means another GitHub action can execute this GitHub action. We'll do this several times with a yet-to-be-created workflow `all.yaml`
- `workflow_dispatch:` this is how you can **run** this workflow via github.com, or with the GitHub API and the GitHub CLI.
- `push` and `pull_request` are directly related to the `git` actions that correspond to them. What's more, we can specify the *git branch* we want this workflow to be run on.

These are the only triggers we'll end up using throughout this book.

### 3. Test Django workflow

The key with this workflow is that we have run `python manage.py test`. If the test, fails, the workflow fails. Once we have everything configured, this means the rest of our CI/CD pipeline *will not* run.

Create (or verify) The `.github/workflows` folder exists:

```
cd path/to/your/project
mkdir -p .github/workflows/
```

Using `-p` with `mkdir` means it will create both folders here. This command should also work in PowerShell on Windows

If you cloned our project in [Getting Started](./02-getting-started.md), then you *may* already have:

- `.github/workflows/test-django-mysql.yaml`
- `.github/workflows/test-django-postgres.yaml`

Replace the `test-django-mysql.yaml` file

We want to replace the existing `.github/workflows/test-django-mysql.yaml` with the following:

yaml

```
name: 1- Test Django & MySQL
on:
  workflow_call:
  workflow_dispatch:
  push:
    branches: [main]
  pull_request:
    branches: [main]
```

## A workflow run is made up of one or more jobs that can run sequentially or in parallel jobs:

This workflow contains a single job called “build”

```
django_mysql:
```

## The type of runner that the job will run on

```
runs-on: ubuntu-latest
```

## Change the default working directory relative to your repo root.

```
defaults:  
  run:  
    working-directory: ./
```

## Add in environment variables for the entire “build” job

```
env:  
  MYSQL_DATABASE: cfeblog_db  
  MYSQL_USER: cfe_blog_user # do *not* use root; cannot re-create the root user  
  MYSQL_ROOT_PASSWORD: 2mLTcmdPzU2LOa0TpALLPoNf1XtIKsKvNn5WBiszczs  
  MYSQL_TCP_PORT: 3306  
  MYSQL_HOST: 127.0.0.1  
  GITHUB_ACTIONS: true  
  DJANGO_SECRET_KEY: test-key-not-good  
  DATABASE_BACKEND: mysql  
services:  
  mysql:  
    image: mysql:8.0.28  
  env:
```

## References the environment variables set at the job-level

```
MYSQL_DATABASE: ${ env.MYSQL_DATABASE }}
MYSQL_HOST: ${ env.MYSQL_HOST }}
MYSQL_USER: ${ env.MYSQL_USER }}
MYSQL_PASSWORD: ${ env.MYSQL_ROOT_PASSWORD }}
MYSQL_ROOT_PASSWORD: ${ env.MYSQL_ROOT_PASSWORD }}
ports:
- 3306:3306
options: --health-cmd="mysqladmin ping" --health-interval=10s
steps:
- name: Checkout code
uses: actions/checkout@v2
```

## When you're using Python, this is a required step

```
- name: Setup Python 3.10
uses: actions/setup-python@v2
with:
```

## Add quotes around version so 3.10 does not become 3.1

```
python-version: "3.10"
- name: Install requirements
run: |
pip install -r requirements.txt
- name: Run Tests
```

## Add additional step-specific environment variables

```
env:
  DEBUG: "0"
  # must use the root user on MySQL to run tests
  MYSQL_USER: "root"
```



## References the environment variables set at the job-level

```

DATABASE_BACKEND: ${ env.DATABASE_BACKEND }
DJANGO_SECRET_KEY: ${ env.DJANGO_SECRET_KEY }
run: |
  python manage.py test

```

So what's going on here?

1. We declare only 1 job to run `django_mysql` in the `jobs` block
2. It runs on `ubuntu-latest` in the current working directory `./` (ie next to `.gitignore`, `manage.py`, etc)
3. It has some environment variables `.env` to the entire job
4. In the `services`, we ensure that MySQL v8.0.28 will run with this job ( `mysql:8.0.28` ) and is initialized with environment variables from the previous step.
5. The actual job steps run after the `services` are done booting
6. It will checkout our code (ie copy it) to the machine running this workflow
7. We use/install `Python 3.10` because of `python-version: "3.10"`
8. We install all Django requirements via our `requirements.txt`
9. We run our tests via our `manage.py`

If any of these above items fail, the workflow will fail and alert us (this includes emailing our GitHub account). These failures are great because they **halt** a push into production with code that doesn't pass tests.

**How amazing is that?**

In this workflow, we did see some django-template-like syntax in this workflow with `${ env.DATABASE_BACKEND }`. This is how we can do string substitution within a workflow. In this case, `${ env.DATABASE_BACKEND }` refers to the `jobs` block, the `django_mysql` job, the `env` block, and finally `DATABASE_BACKEND`. This same sentence I look at it like:

- `jobs:django_mysql:env:DATABASE_BACKEND`
- or
- `jobs["django_mysql"]["env"]["DATABASE_BACKEND"]`

Now, everything in `jobs:django_mysql:env:` is hard-coded to this workflow. In the Action Secrets [section](#), we'll discuss how to abstract these variables away into secrets.

It's important to keep in mind that what you can do with GitHub Actions is nearly unlimited. What I have in this book is simply an approach that has served me very well on dozens of projects and I hope it does the same for you. If you find an approach that works better, please use it and tell me about it!

## Add, commit, & push your code

```
git add .github/workflows/  
git commit -m "Added test workflow for Django"  
git push origin main
```

## Open GitHub Actions

In your browser, go to:

```
https://github.com/your-username/your-private-repo/actions
```

Replace `your-username` and `your-private-repo` with the correct values from when you [created a new repo](#) above.

At this point, you should see the `1- Test Django & MySQL` workflow listed. With this workflow listed you can do the following:

- Run this workflow automatically when you do `git push origin main` (because of the `push:branches: [main]:` configuration)
- Run this workflow manually (one-off) when you click "Run Workflow" in the GitHub Actions tab (because of the `workflow_dispatch:` configuration)
- Run this workflow automatically from another workflow inside your repo (more on this later.) (because of the `workflow_call:` configuration)

If you don't see this workflow listed, here are some possible reasons:

- You didn't save it in the correct location (should be under `.github/workflows` )
- You didn't save in the correct file extension (should be `.yaml` or `.yml` )
- You didn't change the `git remote origin` to your repo correctly
- You didn't commit and/or push your code to GitHub correctly

# Running Workflows in Workflows

## 1. A workflow for workflows

The last workflow we'll create in this chapter is the `.github/workflows/all.yaml` workflow. This one will be the primary entry point to all other workflows. The reason is simple: I want to ensure the workflows run in the order I need them to run in and if any workflow fails, the entire list stops in its tracks.

It will run like this:

1. Run this
2. Then this
3. Then this
4. If this fails, stop for good
5. (Won't be run, 4 failed)
6. (Won't be run, 4 failed)
7. (Won't be run, 4 failed)
8. (Won't be run, 4 failed)
9. (Won't be run, 4 failed)
10. (Won't be run, 4 failed)
11. (Won't be run, 4 failed)
12. (Won't be run, 4 failed)

## 2. The CI/CD pipeline `all.yaml` workflow

Here's what our current `.github/workflows/all.yaml` looks like:

```
yaml
name: 0 - Run Everything
on:
  workflow_dispatch:
  push:
    branches: [main]
  pull_request:
    branches: [main]
jobs:
  test_django:
    uses: ../github/workflows/test-django-mysql.yaml
```

Notice that the job `test_django` uses the `../github/workflows/test-django-mysql.yaml` workflow.

Now, back to the `./.github/workflows/test-django-mysql.yaml` we can remove the following:

```
yaml
push:
  branches: [main]
pull_request:
  branches: [main]
```

Since `all.yaml` will already be doing this for us.

### 3. Push `all.yaml` to GitHub

```
git add .github/workflows/test-django-mysql.yaml
git add .github/workflows/all.yaml
git commit -m "Added a single workflow to trigger others"
git push
```

Now if you go to your GitHub Actions tab on your repo, you should see that `all.yaml` is running and if you click on the running job, you should see something like:

The screenshot shows the GitHub Actions interface for a workflow named "Added a single workflow to trigger others 0 - Run Everything #3". The workflow is triggered via a push to the main branch. The status is "Queued". The total duration is not yet recorded. The artifacts section is empty. The jobs section shows a single job named "test\_django / django\_mysql (3.10)". The job details show that the workflow is triggered on a push to the main branch. The matrix is "test\_django / django\_m...". The job status is "0/1 jobs completed". The "Show all jobs" link is visible.

If you see:

Added a single workflow to trigger others 0 - Run Everything #4

Summary

Jobs

- test\_django / django\_mysql (3.8)
- test\_django / django\_mysql (3.9)
- test\_django / django\_mysql (3.10)

all.yaml

on: push

Matrix: test\_django / django\_m...

0/3 jobs completed

Show all jobs

This is because we have *multiple* versions of python related to these blocks in `./.github/workflows/test-django-mysql.yaml` :

```
yaml
strategy:
  matrix:
    python-version: ["3.8", "3.9", "3.10"]
# Steps represent a sequence of tasks that will be executed as part of the job
steps:
  # Checks-out your repository under $GITHUB_WORKSPACE, so your job can access it
  - name: Checkout code
    uses: actions/checkout@v2
  - name: Setup Python ${ matrix.python-version }
    uses: actions/setup-python@v2
    with:
      python-version: ${ matrix.python-version }
```

In the [Replace the `test-django-mysql.yaml` file](#) section, we do *not* have this code. That's because there is no need to test multiple versions of Python if we have no intention of using them in production.

## 4. Updating `all.yaml` Workflow

In future chapters, we're going to be adding new workflows to this project. Whenever we need that new workflow to run automatically, we need to update `all.yaml` much like this:

```
yaml
name: 0 - Run Everything
on:
  workflow_dispatch:
  push:
    branches: [main]
  pull_request:
    branches: [main]
jobs:
  test_django:
    uses: ../github/workflows/test-django-mysql.yaml
  test_django_again:
    needs: test_django
    uses: ../github/workflows/test-django-mysql.yaml
  test_django_again_again:
    needs: test_django_again
    uses: ../github/workflows/test-django-mysql.yaml
```

Notice that we have 3 jobs (names are arbitrary):

- `test_django`
- `test_django_again`
- `test_django_again_again`

Each job will run the workflow I designate in the `uses` key. In this case, each job runs the same workflow ( `../github/workflows/test-django-mysql.yaml` ) which is fine because this example is here to illustrate the other key: `needs` . Let's define how `needs` will work in this one:

- `test_django` does not have a `needs` key, will run when `all.yaml` runs.
- `test_django_again` needs `test_django` to succeed otherwise this job will not run. That's because `needs: test_django` is prevalent in this job.
- `test_django_again_again` needs `test_django` and `test_django_again` to succeed otherwise this job will not run. That's because `needs: test_django_again` is prevalent in this job AND because `needs: test_django` is prevalent in the `test_django_again` job.

## Using GitHub Repository Action Secrets

Hard-coding variables in a GitHub Action workflow can sometimes be okay; especially when it's test/dummy data. It's *never* okay when they are secret keys that *need* to be hidden from the public. Here are a few examples of items that need to be hidden in this exact project:

- `DJANGO_SECRET_KEY`
- `MYSQL_USER`
- `MYSQL_PASSWORD`
- `MYSQL_ROOT_PASSWORD`
- `MYSQL_HOST`
- `LINODE_BUCKET_ACCESS_KEY`
- `LINODE_BUCKET_SECRET_KEY`
- `DOCKERHUB_USERNAME`
- `DOCKERHUB_TOKEN`
- `LINODE_OBJECT_STORAGE_DEVOPS_ACCESS_KEY`
- `LINODE_OBJECT_STORAGE_DEVOPS_SECRET_KEY`
- `ROOT_USER_PW`
- `SSH_DEVOPS_KEY_PUBLIC`
- `SSH_DEVOPS_KEY_PRIVATE`
- And the many others as outlined in [Appendix A](#)

If done correctly, using GitHub Actions to store secure data in GitHub Action Secrets is generally considered to be a safe option. Just remember that using non-official third-party items in GitHub Actions *can* leave you open to potential security threats. Security is a never-ending battle so be sure to continue to stay vigilant and learn from as many sources as possible on how to better secure your secrets with GitHub Actions.

Aside from what is in this book, further security efforts are encouraged.

Now, let's set up your secrets:

1. Go to your Repo's Settings such as `https://github.com/your-username/your-private-repo/settings` changing `your-username` and `your-private-repo` accordingly.
2. In the sidebar, navigate to `Secrets > Actions`
3. In `Action Secrets`, click the button `New Repository Secret`
4. Name it in the format: `ALL_CAPS_WITH_UNDERSCORES_FOR_SPACES`
5. Value: add a secret here.
6. GitHub Action Secrets are **write only**; after you set a value you will **never** be able to see this secret on GitHub again.



Once you have a key added, you can reference it in a GitHub Action using:

```
${{ secrets.ALL_CAPS_WITH_UNDERSCORES_FOR_SPACES }}
```

Here's a practical example:

1. Navigate to your repo's Action Sections
2. Click **New Repository Secret**
3. Add **DJANGO\_SECRET\_KEY**
4. Value: **much-better-but-probably-\$till-not-gReAT**

In `./.github/workflows/test-django-mysql.yaml` change:

```
yaml
DJANGO_SECRET_KEY: test-key-not-good
```

to:

```
yaml
DJANGO_SECRET_KEY: ${{ secrets.DJANGO_SECRET_KEY }}
```

You can also change `${{ env.DJANGO_SECRET_KEY }}` to `${{ secrets.DJANGO_SECRET_KEY }}` but you don't have to.

## Sharing secrets on shared workflows

First off, secrets will *not* be passed to forked repos that submit pull requests. That means people who are *not collaborators* can't just run your workflows and get your secrets.

But we do have `all.yaml` that will run our `./.github/workflows/test-django-mysql.yaml` workflow. How do we ensure the secrets work on both workflows? We must update `workflow_call` in the workflow that's being called to include our required secrets.

So this:

```
yaml
on:
  workflow_call:
```

Becomes this:

```
yaml
on:
  workflow_call:
    secrets:
      DJANGO_SECRET_KEY:
        required: true
```

Now, in any *other* workflow I can do something like:

```
yaml
jobs:
  test_django_with_secret:
    uses: ./.github/workflows/test-django-mysql.yaml
    secrets:
      DJANGO_SECRET_KEY: ${ secrets.DJANGO_SECRET_KEY }
```

We'll repeat this process several times going forward. This is a super easy step to forget but, luckily for us, GitHub Actions will let us know when we do.

## Update `all.yaml` with shared secrets

In `.github/workflows/all.yaml`, change:

```
yaml
jobs:
  test_django:
    uses: ../github/workflows/test-django-mysql.yaml
```

to:

```
yaml
test_django:
  uses: ../github/workflows/test-django-mysql.yaml
  secrets:
    DJANGO_SECRET_KEY: ${ secrets.DJANGO_SECRET_KEY }
```

Commit and push!

```
git add .github/workflows/test-django-mysql.yaml
git add .github/workflows/all.yaml
git commit -m "Added secrets"
git push origin main
```

## Act: Run Your GitHub Actions Anywhere

If you ever get stuck with GitHub Actions *or* you just want to run these workflows manually, check out the tool [Act](#). It allows us to run these workflows locally (or on our virtual machine). This tool also helps unlock the portability of the GitHub Actions workflows.

Using [Act](#) we can run:

```
act -j test_django
```

And many other commands like it. Learning [Act](#) is outside the scope of this book but it's important to know about it to maintain as much portability as possible!

Chapter 3

# Containerize Django with Docker

## Chapter 3

# Containerize Django with Docker

Let's assume I know nothing about setting up Python and Python virtual environments, but I am decent with using the command line. With this, I ask you to send me your Django project so I can test it out on my local computer. What do you do?

With Docker, you can simply:

- Install Docker (such as Docker Desktop) as it is free and works on macOS, Linux, and Windows (you might need to use Windows Subsystem for Linux).
- Open the command line and run `docker run my-dockerhub-reop/my-django-container`

Without Docker it's:

- Install Python (Wait, which version? From where?)
- Clone code with Git (Is Git installed? Do you know how to use Git?)
- Create a Virtual Environment (What's that?)
- Activate Virtual Environment
- Install Requirements (Wait, is it `pipenv`, `poetry`, or what?)
- Run Django

As you can see, running Django from zero is a lot more complex than Docker. The above scenario is true for almost every application written in just about *any* language. Running a Node.js/Express.js app? Same challenge. Ruby/Ruby On Rails? Same thing. Java/Spring? Same.

Setting up environments to run your code is challenging enough without considering but even more so in production since it *must* run every time you deploy a new version and *all* dependencies *must* be installed otherwise a lot of catastrophic errors can/will occur. As you may know, not all versions of Python (or Node, or Ruby, or Java) will even be readily available on your servers. Generally speaking, if Docker is on the machine, your Docker apps can run.

Docker apps (or Docker containers) are isolated environments at the OS level instead of just the Python level like Virtual Environments. This means that Docker can run nearly any version of Python (yes, probably even Python 2.1) right next to any other version of Python (like Python 3.11) on the exact same server while the server is none the wiser.

As a Django developer, I wish I adopted Docker sooner.

## Is Docker Easy?

At first, Docker seems incredibly hard because there's so much that Docker can do. Then you start hearing about Docker Compose, Docker Swarm, Kubernetes, Serverless, and other new terms. You hear about them *because* Docker is fundamental to them. You *unlock* a world of opportunity once you start running your apps through Docker.

I think of Docker as just another document with a list of logical instructions. Here's a simple **Dockerfile** (the Docker instructions document) that will work with your Django app in your local development environment:

### **dockerfile**

```
FROM python:3.10.4-slim
COPY . /app
WORKDIR /app
RUN python3 -m venv /opt/venv
RUN /opt/venv/bin/pip install pip --upgrade && \
    /opt/venv/bin/pip install -r requirements.txt
CMD /opt/venv/bin/gunicorn cfeblog.wsgi:application --bind "0.0.0.0:8000"
```

This example is a **non-production Dockerfile**, but it's not far off. What I did here is almost exactly what I described before about how sharing a Django app without Docker works:

- Install Python: **FROM python:3.10.4-slim**
- Clone code: **COPY . /app**
- Create a Virtual Environment: **RUN python3 -m venv /opt/venv**
- Activate Virtual Environment: not needed because I just use **/opt/venv/bin** instead
- Review [this section](#) if this part is unclear
- Install Requirements: **RUN /opt/venv/bin/pip install pip --upgrade && /opt/venv/bin/pip install -r requirements.txt**
- Run Django: **/opt/venv/bin/gunicorn cfeblog.wsgi:application --bind "0.0.0.0:8000"**
- **gunicorn** is a production-ready way to run Django; **python manage.py runserver** is not.

Before I used Docker regularly, I would look at a **Dockerfile** like this and think it adds yet another layer of complexity on top of a potentially already complex Django project. Now, I have concluded that it might *feel* very complex initially, but in the long run, it makes our lives much *less* complex. Especially when you need *portability* and *repeatability*.

But why do you need *portability*?

There are many reasons for *portability* and Docker. Here are just a few good ones:

- Switch your hosting provider within hours instead of days, weeks, or months.
- Share with team members and/or other businesses without touching production or requiring specific knowledge other than `docker run`.
- Testing production code on non-production machines/hosts.
- Horizontally scale your virtual machines to meet demand with ease.
- Move to a managed serverless architecture that runs on Docker containers & Kubernetes.
- Create a custom Kubernetes cluster to host *all* of your Django / Docker projects.
- Faster recovery when servers and/or data center running your code go down and cannot recover.
- No more using Dave's custom bash spaghetti scripts when Dave left the team 4 years ago.
- Your Platform as a Service (PaaS) leaks all credentials in their service and you need to migrate real quick.

Those are just a few examples of when you might want to use Docker. I'm sure there are more.

If you need more convincing (or if you're already convinced), let's write some actual code to see how simple it is and why using Docker is the right call for production.

## 1. Install Docker Desktop

Your local machine just needs Docker Desktop. To install,

1. Go to [Docker Desktop](#)
2. Download for your operating system (macOS, Windows, Windows Subsystem for Linux (WSL), or Linux)
3. Complete the installation process for your machine

## 2. Dockerfile for Production with Django

Docker containers have ephemeral runtimes. That means that they run like any other app on your machine and they are *made* to be destroyed. While there are exceptions to this, by default, data within a container does not persist when the container is no longer available.

Think of Docker containers much like you think of smartphone apps -- you can easily delete them, install again, delete again, install again, delete again, as often as you like. You can repeat this for nearly any app and for nearly any number of times all while your data for the app/service remains intact (thanks to managed Databases and Object Storage -- these are covered later in this book).

How this works with *Django*, *FastAPI*, *Flask*, *Express.js*, and *Ruby on Rails*, is essentially the same:

1. Write your code
2. Update or create the `Dockerfile` (if necessary)
3. Build your Docker container image
4. Run your Docker container image
5. Repeat steps 1-4



The programming language we use does not matter to Docker at all, as long as it runs without causing major errors on the system. This is exactly like how your macOS or Windows systems can always run the exact same apps. (Can you read the sarcasm?)

What about *PostgreSQL*, *MySQL*, *Redis*, *MongoDB*, and *Cassandra* -- should we use Docker for those? Even with Docker containers being ephemeral, you can still run databases with Docker containers by attaching persistent volumes (non-ephemeral data storage volumes) but we're going to skip that layer of complexity in this book. For our persistent data, we'll use managed Linode services for databases ([Chapter 5](#)) and object storage ([Chapter 7](#)).

## Remove cloned Dockerfile and Docker Compose file

If you cloned our project in the [Getting Started Chapter](#), you will already have the following docker files:

- `Dockerfile`
- `docker-compose.yaml`

You should now **remove** these files. We are going to *replace* these files with the contents of this book. If, for some reason, you're having issues with the code within the book use [this repo](#) as your primary reference for the *absolute final* code from this book.

## Production-ready Docker file

Let's have a look at our *production-ready* `Dockerfile` for **Django**:

```
dockerfile
FROM python:3.10.3-slim
```

```
# copy your local files to your
# docker container
COPY . /app

# update your environment to work
# within the folder you copied your
# files above into
WORKDIR /app

# os requirements to ensure this
# Django project runs with mysql
# along with a few other deps
RUN apt-get update && \
    apt-get install -y \
    locales \
```

```
libmemcached-dev \  
default-libmysqlclient-dev \  
libjpeg-dev \  
zlib1g-dev \  
build-essential \  
python3-dev \  
python3-setuptools \  
gcc \  
make && \  
apt-get clean && \  
rm -rf /var/lib/apt/lists/*  
  
# libpq-dev is a PostgreSQL client install, change as needed  
# default-libmysqlclient-dev is a MySQL client, this is our current default  
  
ENV PYTHON_VERSION=3.10  
ENV DEBIAN_FRONTEND noninteractive  
  
# Locale Setup  
RUN locale-gen en_US.UTF-8  
ENV LANG en_US.UTF-8  
ENV LANGUAGE en_US:en  
ENV LC_ALL en_US.UTF-8\  
RUN sed -i -e 's/# en_US.UTF-8 UTF-8/en_US.UTF-8 UTF-8/' /etc/locale.gen \  
    && locale-gen  
RUN dpkg-reconfigure locales  
  
# Create a Python 3.10 virtual environment in /opt.  
# /opt: is the default location for additional software packages.  
RUN python3.10 -m venv /opt/venv  
  
# Install requirements to new virtual environment  
# requirements.txt must have gunicorn & django  
RUN /opt/venv/bin/pip install pip --upgrade && \  
    /opt/venv/bin/pip install -r requirements.txt && \  
    chmod +x config/entrypoint.sh  
  
# entrypoint.sh to run our gunicorn instance  
CMD [ "/app/config/entrypoint.sh" ]
```

This file is a bit more complex than our earlier example but that's only because we added a few os-level configuration items to ensure the stability of this app and Python 3.10.

If this `dockerfile` is failing you, please review the code that we have hosted on our GitHub repo for this book at [Deploy Django on Linode with MySQL](#) and **not the repo** for the code we cloned in [this section](#) (the book code and cloned repo will differ slightly).

### 3. Running Django in production: Gunicorn

`gunicorn` ([docs](#)) is the defacto standard for running Python web applications in production. You can use it with Django, FastAPI (with a `uvicorn` worker), Flask, and many others.

In development you're probably used to running:

```
python manage.py runserver
```

This command, as you likely know, is not suitable for production. Instead, we want to use `gunicorn` since it's a production-ready Python Web Server Gateway Interface (WSGI). `gunicorn` makes it easy for `Django`, `Flask`, `FastAPI`, or any other Python web application to understand web requests. In other words, nearly every Python web application can use `gunicorn` in production; `gunicorn` can even run with a single Python file that has no web framework and still be able to handle HTTP requests!

To replace our *development* command with our production-ready command, it will be:

```
gunicorn mydjango.wsgi:application
```

**Windows users:** At the time of this writing, `gunicorn` does not work natively on Windows but will work in Docker containers.

The `mydjango.wsgi:application` is going to be custom per project, per framework. Let's take a look at the `wsgi.py` file in our Django project:

python

```
"""
WSGI config for mydjango project.

It exposes the WSGI callable as a module-level variable named ``application``.

For more information on this file, see
https://docs.djangoproject.com/en/3.2/howto/deployment/wsgi/
"""

import os

from django.core.wsgi import get_wsgi_application

os.environ.setdefault('DJANGO_SETTINGS_MODULE', 'mydjango.settings')

application = get_wsgi_application()
```

If you go into your Django project, next to `settings.py` you'll see both `urls.py` and `wsgi.py`. Your `wsgi.py` will resemble the above code replacing `mydjango` with your project's name.

For some environments, we'll need to bind `gunicorn` to a specific port which we can do with:

```
gunicorn mydjango.wsgi:application --bind "0.0.0.0:8000"
```

### Async Python Web Applications & Gunicorn

It's important to mention that `gunicorn` can run asynchronous Python web apps like `FastAPI` and async `Django`. With `Django`, our project has an `asgi.py` file that's almost identical to the `wsgi.py` file but just requires the following command to work:

```
gunicorn mydjangoapp.asgi:application -k uvicorn.workers.UvicornWorker --bind
"0.0.0.0:8000"
```

A few things to note:

- `asgi.py` is for async
- `-k uvicorn.workers.UvicornWorker` requires `uvicorn` installed with `pip install uvicorn`
- If you're using `FastAPI`, you can run this much like `gunicorn main:app -k uvicorn.workers.UvicornWorker`

## 4. Create the Entrypoint File ( `config/entrypoint.sh` )

In our above `Dockerfile`, we saw the line that started with `CMD`. `CMD` is the *default* command to run this docker container. You use a different command at runtime if you need (more on this later).

The key to `CMD` is it's centered around *runtime* and not *build time*. This is important because *build time* will likely *not* have access to our Database and related services.

In the **non-production** `Dockerfile` we had the `CMD`:

```
CMD /opt/venv/bin/gunicorn cfeblog.wsgi:application --bind "0.0.0.0:8000"
```

Now we have:

```
CMD [ "/app/config/entrypoint.sh" ]
```

You can select either method that works for you but I prefer using a custom `bash` script so I can provide additional details as needed.

Here's my `entrypoint.sh` :

`config/entrypoint.sh`

sh

```
#!/bin/bash
APP_PORT=${PORT:-8000}
cd /app/
/opt/venv/bin/gunicorn --worker-tmp-dir /dev/shm cfeblog.wsgi:application --bind
"0.0.0.0:
  ${APP_PORT}"
```

`APP_PORT=${PORT:-8000}` If `PORT` is sent in the Environment Variables `APP_PORT` will be that value. If it's not set, `APP_PORT` will use the fallback value of `8000` ensuring we have a `PORT` for gunicorn to bind to.

But what if we wanted to make sure we ran `python manage.py migrate` every time I ran this container? We **cannot** effectively run this command during a container *build time* -- even if it's possible. `python manage.py migrate` relies on a connection to a database which likely relies on environment variables so `python manage.py migrate` *should fail* if we are building this container correctly.

Here are two `Dockerfile` `RUN` commands we should **never** see:

dockerfile

```
RUN /opt/venv/bin/python manage.py migrate
RUN /opt/venv/bin/python manage.py collectstatic --no-input
```

Both of these seem like great ideas, but there are two key problems:

- We don't have a production database attached to this container yet (ie it's **build time** not **runtime**)
- We don't have *environment variables* (secrets) attached to this container yet
- `python manage.py migrate` *sometimes* seems to work during **build time** because you never removed the `sqlite` configuration in Django

We have to remember that the `Dockerfile` builds the container. `CMD` and `config/entrypoint.sh` are for when we *run* the Docker container app.

Here's the final `entrypoint.sh` file I'll use (for now):

`config/entrypoint.sh`

sh

```
#!/bin/bash
APP_PORT=${PORT:-8000}
cd /app/
/opt/venv/bin/python manage.py migrate
/opt/venv/bin/gunicorn --worker-tmp-dir /dev/shm cfeblog.wsgi:application --bind
"0.0.0.0:
${APP_PORT}"
```

- `/opt/venv/bin/python manage.py migrate` running this has essentially zero effect if migrations are already done, but has a major effect if they are not done.
- `python manage.py collectstatic` will be saved for another process
- `python manage.py createsuperuser` *could* be ran in `config/entrypoint.sh` as well

## 5. `.dockerignore`

Docker containers need *as little* of your code as possible because the building process will

- Install all dependencies
- Become much larger if unnecessary files are added

Using a `.dockerignore` is the best way to ensure we do *not* accidentally add too many files to our docker containers. It works exactly like a `.gitignore` file but is used when you do stuff like:

dockerfile

```
COPY . /app
```

Here is the minimum `.dockerignore` I recommend you use for this project:

```
staticfiles/  
mediafiles/  
.DS_Store  
venv  
.env  
*.py[cod]  
__pycache__/
```

You should also just add [the GitHub Python .gitignore file](#) contains to the bottom of this `.dockerignore`

## 6. Build & Run Container

From here on out, if we make changes to our Django project and want to run those changes within a Docker container. We must *build* that container.

Here's how that's done:

```
docker build -t cfe-django-blog -f Dockerfile .
```

Then to run it, we'd do:

```
docker run -e DJANGO_SECRET_KEY=something -p 8000:8000 cfe-django-blog
```



Let's break these two commands down:

#### **docker build**

- At its core, this command tells docker we're going to build a new Docker container image
- **-t** is the flag to "tag" this container. This is a name you get to pick and if you plan to use Docker Hub (like we do later in this chapter), you'll have to include your username so it's **-t codingforentrepreneurs/myproject** instead of just **-t myproject**
- **-f Dockerfile** this is telling Docker which Dockerfile to use to build this container. You can have as many Dockerfiles as you'd like.
- **.** The final period ( **.** ) tells docker **where** to build this container. As you may recall, our **Dockerfile** does **COPY . /app** . The **COPY . /app** will correspond directly to this final period.

#### **docker run**

- At its core, this command tells docker that we want to run an image
- **-e** is a flag for passing an environment variable to this run command. You can have many of these. You can also use **--env-file** like: **docker run --env-file .env ...** . The syntax for **-e** is **KEY=VALUE** so **-e KEYA=value\_a -e KEYB=value\_b** and so on.
- **-p** This allows your Docker container to run on a port that the rest of your system can access. You know how when you run **python manage.py runserver** and you can access Django at port 8000 (by default) in your browser? This is the same concept, but for Docker. Without **-p** , your browser (and the rest of your computer) has no way to connect to what's running within the container. Remember how I said Docker containers are isolated? This is one way to "expose" it to your computer (or external world)
- The **cfe-django-blog** in **docker run ... cfe-django-blog** is the tag for this container. It is *identical* to what you used in the **docker build -t** call. What's more, you can run *official* Docker images (like Nginx, MySQL, Python, and others) using this same run command. Basically **docker run nginx** -- this will even *download* the image for you if you don't have it. Neat!

## Basic Workflow to Build a Docker Container

Before we go any further, let's take a look at the GitHub Action to just build our Docker container. It's as simple as:

`.github/workflows/container.yaml`

yaml

```
name: 2 -Build App Container Image

on:
  workflow_call:
  workflow_dispatch:

jobs:
  docker:
    runs-on: ubuntu-latest
    steps:
      - name: Checkout
        uses: actions/checkout@v2
      - name: Set up QEMU
        uses: docker/setup-qemu-action@v1
      - name: Set up Docker Buildx
        uses: docker/setup-buildx-action@v1
      - name: Build container image
        run: |
          docker build -f Dockerfile -t myproject .
```

The goal of seeing this workflow is to show you how simple it's going to be to build our container in GitHub Actions. It's also important to know that this workflow is *not* passing *any* environment variables to the build command. The only time our Docker Container Image should need environment variables is during *runtime*.

And yes, in a few sections we'll change `docker build -f Dockerfile -t myproject .` to include a tag of a container we need to use.

## 7. Using Docker Compose

Docker Compose was invented to simplify running and building multiple containers at once.

So instead of doing:

```
docker build -t my_django project
docker build -t my_mysql_local_db
docker build -t my_redis_local
docker build -t my_worker

docker run my_django project
docker run my_mysql_local_db
docker run my_redis_local
docker run my_worker
```

We can just do:

```
docker compose up
```

**Sound intriguing to you? Let's do this.**

### Docker Compose for Django development

During development, I *rarely* have my Django application running through Docker. You *can* run it through docker but I really don't want to rebuild my Django container *every time* I make a change.

So, what do I use Docker Compose for in Development? *Databases*.

In addition to running Docker containers when we need, Docker Compose can also:

- Start containers on a system reboot
- Run containers in the background
- Change ports for common technologies (MySQL is typically running on 3306)
- Allow every local project to have its own running database (or datastore) instead of a shared one for the entire machine
- Build our images when needed
- Run different containers for different needs (like profiles)

Let's look at a `docker-compose.yml` file just for our Django app:

yaml

```
version: "3.9"

services:
  app:
    image: cfe-django-blog
    build:
      context: .
      dockerfile: Dockerfile
    restart: unless-stopped
    env_file: .env
    ports:
      - "8000:8000"
```

Now we can just run:

```
docker compose up --build
```

If `docker` works on your machine but the command `docker compose` doesn't, then you'll need to install the pip package `docker-compose` with `python3 -m pip install docker-compose`. After you do that, you can run `docker-compose` instead of `docker compose`. We'll see this again in the Ansible chapter when we use Docker compose in production.

This will:

- Build our app based on `app:build:context` and `app:build:dockerfile:`
- Run our app with local port `8000` mapped to the container port `8000`
- Automatically restart our app unless we manually stop it ( `restart: unless-stopped` )
- Use our local `.env` file to run this container

Running this as a `docker run` command would require you to do this:

```
docker build -f Dockerfile -t cfe-django-blog .
docker run --env-file .env -p 8000:8000 --restart unless-stopped cfe-django-blog
```

Now, you try to remember how to run these two lines or just opt for Docker compose. You probably already know what I recommend.

Let's add a local MySQL Database to our Docker compose file:

yaml

```
version: "3.9"

services:
  app:
    image: cfe-django-blog
    build:
      context: .
      dockerfile: Dockerfile
    restart: unless-stopped
    env_file: .env
    ports:
      - "8000:8000"
    profiles:
      - app_too
    depends_on:
      - mysql_db
  mysql_db:
    image: mysql:8.0.26
    restart: always
    env_file: .env
    ports:
      - "3307:3307"
    expose:
      - 3307
    volumes:
      - mysql-volume:/var/lib/mysql
    profiles:
      - main
      - app_too

volumes:
  mysql-volume:
```

Let's break down all the changes:

- In `services:app`, we added `profiles` and `depends_on`.
- `depends_on` allows a service (or container) to *wait* until another container is ready and done
- `profiles` allows us to run only *some* services (or containers) at a time.
- `mysql_db` references a public Docker MySQL image found [here](#) with a tag of `8.0.26` (found [here](#)). This tag happens to be the actual version of MySQL this image is running.
- `restart:always` means this database will *always* restart especially when we run in detach mode and restart the machine (more on this in a moment).
- Both `app` and `mysql_db` services use the same `.env` file but you can easily reference another file or multiple
- `volumes:mysql-volume` is an automated way to ensure our MySQL data is *not ephemeral* like a default container. In other words, we're automatically attaching a persistent volume here.

Let's see how to run these:

```
docker compose --profile main up -d
```

This will run:

- the `main` profile
- in `-d` detached mode -- this just means to run these services as background services (they are non-blocking on the command line)

**Did it fail?** If you see `Bind for 0.0.0.0:3307 failed: port is already allocated`; this means that the port `3007` is in use elsewhere. You can either stop that service or choose a new port. If you choose a new port be sure to update it in `.env` and wherever else it's declared.

We do not need to build the image ( `image: mysql:8.0.26` ) for the `mysql_db` service *because* it's already built and stored on Docker Hub (as mentioned above).

To take this down, we can run:

```
docker compose --profile main down
```

- If you change service names or add/remove services, you might need to run `docker compose --profile main down --remove-orphans`
- To remove all volumes and delete all database data, run `docker compose --profile main down -v`

Let's say we wanted to add *Redis* to our local environment, here's what you need to update your `docker-compose.yaml` to:

yaml

```
version: "3.9"

services:
  app:
    image: cfe-django-blog
    build:
      context: .
      dockerfile: Dockerfile
    restart: unless-stopped
    env_file: .env
    ports:
      - "8000:8000"
    profiles:
      - app_too
    depends_on:
      - mysql_db
  mysql_db:
    image: mysql:8.0.26
    restart: always
    env_file: .env
    ports:
      - "3307:3307"
    expose:
      - 3307
    volumes:
      - mysql-volume:/var/lib/mysql
    profiles:
      - main
      - app_too
```

```
redis_db:
  image: redis
  restart: always
  expose:
    - 6388
  ports:
    - "6388:6388"
  volumes:
    - redis_data:/data
  entrypoint: redis-server --appendonly yes --port 6388
  profiles:
    - main
    - app_too

volumes:
  mysql-volume:
  redis-volume:
```



In the [Ansible Chapter](#), we will use a version of Docker compose. Here's a preview of what it will be:

`docker-compose.prod.yaml`

yaml

```
version: "3.9"

services:
  watchtower:
    image: index.docker.io/containrrr/watchtower:latest
    restart: always
    volumes:
      - /var/run/docker.sock:/var/run/docker.sock
      - /root/.docker/config.json:/config.json
    command: --interval 30
    profiles:
      - app
  app:
    image: index.docker.io/codingforentrepreneurs/cfe-django-blog.com:latest
    restart: always
    env_file: ./env
    container_name: prod_django_app
    environment:
      - PORT=8080
    ports:
      - "80:8080"
    expose:
      - 80
    volumes:
      - ./certs:/app/certs
    profiles:
      - app
```

```
redis:
  image: redis
  restart: always
  ports:
    - "6379:6379"
  expose:
    - 6379
  volumes:
    - redis_data:/data
  entrypoint: redis-server --appendonly yes
  profiles:
    - redis

volumes:
  redis_data:
```

I'll explain in more detail how this works later but for now, a high-level overview is:

- **watchtower** : this small container will run next to our App container. It will enable graceful *updates* to our running App container. This means that when we push our container image into Docker Hub, **watchtower** will automatically fetch and restart it for us! How it's configured above also works with private repositories! (Assuming the host machine is authenticated with Docker Hub)
- **services:app:expose** -- this app is now going to run on port 80. This is intentional so our IP address is what connects directly to the services running within our Docker container.
- **services:redis** -- this is a quick and easy way to get a Redis instance running in production. A managed instance is preferred but that often comes at a higher cost. Heavy workloads should *not* use this method in my opinion.

## 8. Push & Host on Docker Hub

Okay so we have GitHub and Docker Hub... what's the difference? It's all about the use case.

- GitHub is for code
- Docker Hub is for containers

They each have their own kind of version control: GitHub with Git and Docker Hub with container tags.

Containers are often huge (5-20 GB), whereas code is often small (under 10 MB; if not under 2 MB).

Containers are not stored as one big file but rather broken up into smaller chunks called layers. Docker Hub is a registry designed to manage these containers and their respective layers. Docker has open-sourced the Registry process as you can read about [here](#). This means you can deploy your *own* Docker Hub if you want (you'd say that you're deploying your own Docker Registry).

The nice thing about both GitHub and Docker Hub is that they provide:

- Public repos
- Private repos
- Easily discovered repos
- Ways to automatically build containers
- Ways to share containers with hosting platforms

To that end, let's push our built code onto Docker Hub.

**Sign up for Docker Hub on [hub.docker.com/](https://hub.docker.com/)**

Before we create a repo, I want to remind you that we use GitHub Actions to build our containers. Once we have a Docker Hub account and an API Token, we will never have to manually create a repo again; GitHub Actions will create the repo for us except it defaults to a Public repo. You'll just need to log in to Docker Hub and change it to be private.

## Create a repository [here](#)

Docker Hub allows for *1 Private Repository* for every free account and *Unlimited Public Repositories*. Given the sheer size of Docker containers, the limitations of the free account are justified. I gladly pay \$60/year for Docker Pro just for unlimited private repositories.

Here are the options I suggest:

- **Name:** *add your own unique name or* `cfe-django-blog`
- **Description:** (optional)
- **Visibility:** *Private* (if you can)
- **Build Settings:** **None**
- Hit Create

In **Build Settings**, you might be tempted to connect to GitHub but *\_don't do it\_*. GitHub Actions will do *all* of the container building we need. It seems that free accounts might not have this ability any longer.

There are several advantages to using GitHub Actions (as opposed to Docker Hub) to automate our container building including:

- Flexibility to change our Docker Registry (for example, choosing not to use Docker Hub)
- Build and or Push errors are in one place (GitHub Actions)
- Secrets are all in one place (GitHub Actions)
- Speed; I've found that GitHub Actions is *dramatically* faster than Docker Hub when building containers (especially on the Docker Hub free account)
- Ease; Building containers is easy no matter where you do it; automating the build with GitHub Actions was [super easy remember?](#)
- Building vs CI/CD; A free account on Docker Hub *just* builds containers, it doesn't perform CI/CD
- Automated Notifications; Do you want to send a slack message when your container is built? How about an SMS? A Tweet? Within a GitHub Workflow, you can do that along with some more code you'd have to write. Docker Hub has webhooks but who's making an ingestion server just to relay build notifications.

I want to make it clear that if you use Gitlab CI/CD instead of GitHub Actions, everything above is still likely to be true.

## Create a Docker Hub API Access Token Key

1. Navigate to [this link](#)
2. Add a token description to *match* such as: `GITHUB_ACTIONS_TOKEN_CFE_DJANGO_DEPLOY`
3. Click **Generate**

You should see something like this:

### Copy Access Token

When logging in from your Docker CLI client, use this token as a password. [Learn more](#)

ACCESS TOKEN DESCRIPTION

GITHUB\_ACTIONS\_TOKEN\_CFE\_DJANGO\_DEPLOY

ACCESS PERMISSIONS

Read, Write, Delete

**To use the access token from your Docker CLI client:**

1. Run `docker login -u codingforentrepreneurs`
2. At the password prompt, enter the personal access token.

`e90485e4-9477-4b55-8b9f-3c54a65a522a`



**WARNING:** This access token will only be displayed once. It will not be stored and cannot be retrieved. Please be sure to save it now.

**Copy and Close**

4. Copy your token: something like `e90485e4-9477-4b55-8b9f-3c54a65a522a`
5. Go to GitHub Actions Secrets. Add the following:

KEY: `DOCKERHUB_USERNAME`

Value: `Your username`

KEY: `DOCKERHUB_TOKEN`

Value: `e90485e4-9477-4b55-8b9f-3c54a65a522a` (your just created Access Token)

KEY: `DOCKERHUB_APP_NAME`

Value: `add your own unique name or cfe-django-blog`

## 9. Create GitHub Action Workflow

In `.github/workflows/container.yaml` add the following:

yaml

```
name: 2 -Build App Container Image

on:
  workflow_call:
    secrets:
      DOCKERHUB_USERNAME:
        required: true
      DOCKERHUB_TOKEN:
        required: true
      DOCKERHUB_APP_NAME:
        required: true
    workflow_dispatch:

jobs:
  docker:
    runs-on: ubuntu-latest
    steps:
      - name: Checkout
        uses: actions/checkout@v2
      - name: Set up QEMU
        uses: docker/setup-qemu-action@v1
      - name: Set up Docker Buildx
        uses: docker/setup-buildx-action@v1
      - name: Login to DockerHub
        uses: docker/login-action@v1
        with:
          username: ${ secrets.DOCKERHUB_USERNAME }
          password: ${ secrets.DOCKERHUB_TOKEN }
      - name: Build container image
        run: |
          docker build -f Dockerfile \
            -t ${ secrets.DOCKERHUB_USERNAME }/${ secrets.DOCKERHUB_APP_NAME }:latest \
            -t ${ secrets.DOCKERHUB_USERNAME }/${ secrets.DOCKERHUB_APP_NAME }:
            ${GITHUB_SHA::7}-${GITHUB_RUN_ID::5} \
            .
      - name: Push image
        run: |
          docker push ${ secrets.DOCKERHUB_USERNAME }/${ secrets.DOCKERHUB_APP_NAME }
          --all-tags
```

Let's go over the keys:

- **docker/setup-buildx-action** Ensures docker is available on this workflow
- **docker/login-action** allows us to login via our Docker Hub Username nad API Access Key Token.
- **-t \${ secrets.DOCKERHUB\_USERNAME }}/\${ secrets.DOCKERHUB\_APP\_NAME }}:latest**

For every build, I add the **:latest** tag so I know for sure that's the most recent build. This line will be something like **-t codingforentrepreneurs/cfe-django-blog:latest** .

Before we go any further, the rationale behind **codingforentrepreneurs/cfe-django-blog:latest** is so we can do either of the following:

As a command-line command:

```
docker run codingforentrepreneurs/cfe-django-blog:latest
```

or in a Dockerfile:

**docker**

```
FROM codingforentrepreneurs/cfe-django-blog:latest
```

Since this repository is private, the only way the above two commands work is if we **log in to Docker** on the machine attempting to use this Container image. We'll revisit this again when we implement [Ansible](#).

You can have multiple tags per Docker container image. It's nice because then it allows us to have the latest version like what we did with **:latest** as well as a historical record of versions especially if we need to roll back our deployments.

Here's a Docker tag approach for tagging I like using on GitHub Actions:

```
-t ${ secrets.DOCKERHUB_USERNAME }/${ secrets.DOCKERHUB_APP_NAME }:${GITHUB_SHA::7}-${GITHUB_RUN_ID::5}
```

`GITHUB_SHA` and `GITHUB_RUN_ID` are both examples of GitHub Actions Environment Variables [there are many](#).

`GITHUB_SHA` is the specific `git` commit cut down to the first 7 characters `::7`. `GITHUB_RUN_ID` is the GitHub workflow run id cut down to the first 5 characters `::5`.

`docker push ${ secrets.DOCKERHUB_USERNAME }/${ secrets.DOCKERHUB_APP_NAME } --all-tags` will push `:latest` as well as `:${GITHUB_SHA::7}-${GITHUB_RUN_ID::5}` and any other tags you add to your container.

## 10. Update `all.yaml` Workflow

In `./.github/workflows/all.yaml`, update to the following:

yaml

```
name: 0 - Run Everything

on:
  workflow_dispatch:
  push:
    branches: [main]
  pull_request:
    branches: [main]

jobs:
  test_django:
    uses: ./.github/workflows/test-django-mysql.yaml
  build_container:
    needs: test_django
    uses: ./.github/workflows/container.yaml
    secrets:
      DOCKERHUB_USERNAME: ${ secrets.DOCKERHUB_USERNAME }
      DOCKERHUB_TOKEN: ${ secrets.DOCKERHUB_TOKEN }
      DOCKERHUB_APP_NAME: ${ secrets.DOCKERHUB_APP_NAME }
```



Chapter 4

# Linode Managed Databases: Using MySQL for Django

## Chapter 4

# Linode Managed Databases: Using MySQL for Django

The goal of this chapter is to set up a managed database cluster on Linode, prepare it for your project, and integrate it with Django.

We're going to be doing this *exact same process* for both **development** and **production**. We do this to ensure our development environment matches our production environment as much as possible.

I use `docker compose` to run my development database (like we did in [the previous chapter](#)) with the *same database version* as my production environment. I'll leave it up to you to choose.

Here's the setup process:

1. Create a MySQL Database cluster on Linode's console
2. Install MySQL Client (non-python)
3. Use the MySQL Client to set up your project in MySQL. This will include a new non-root user with permissions to a new database.
4. Install a Python MySQL Client for Django
5. Configure Django with database user, database name, database port, and database host.
6. Run Django Management commands (such as `migrate` , `loaddata` , `createsuperuser` )

## 1. Create a MySQL Cluster on Linode

Login to The [Linode Console](#)

Navigate to [Databases](#)

Click `Create Database Cluster`

For development, we'll use the following configurations:

**Cluster Label:** `cfe-db-dev`

**Database Engine:** `MySQL 8 (currently v8.0.26)`

**Region:** (Select one near you)

**Shared CPU:** `Nanode 1 GB`

**Plan:** `DBaaS MySQL - Nanode 1GB`

**Number of Nodes:** `1`

Under **Add Access Controls** add:

```
0.0.0.0/0
```

**IMPORTANT:** this value will allow *all* connections from anywhere. The database we're using in this step is *only* for development.

It's also important to note that this step can take up to *20 minutes* to complete.

For production, I recommend using at least a 2 node cluster using at minimum the shared **Linode 2 GB** instance. More on this in the next chapter.

## Once provisioned, download *Database CA Certificate*

1. Go to [Databases](#)
2. Select your database **cfe-db-dev**
3. Under **Connection Details**, click *Download CA Certificate*
4. At the root of your project, create a directory called **certs**
5. Update **.gitignore** with **certs/ : echo 'certs/' >> .gitignore**
6. Move your download certificate to **certs/** and rename it to **db.crt** like:

```
mv ~/downloads/django-db-dev-ca-certificate.crt ~/dev/django-project/certs/db.crt
```

Once we create a production cluster, the **db.crt** contents will be stored within GitHub Action Secrets. (More on this in the next chapter). Renaming our *CA Certificate* makes switching database clusters convenient.

## 2. Install MySQL Client Locally

On our local machine, we will have to run a number of commands to configure our database. These commands require a MySQL client to connect to our MySQL database.

A MySQL client is typically much smaller in size than a MySQL server installation since all we're doing is connecting to an already running MySQL server.

### Install MySQL client on MacOS

Using [homebrew](#)

```
brew install mysql-client
```

You can also use:

- MySQL Shell (can use JavaScript, Python, or SQL commands): `brew install mysql-shell` or via the [docs](#)
- MySQL Server Community Edition: `brew install mysql` or via the [docs](#)

### Install MySQL client on Windows

Using [chocolatey](#)

```
choco install mysql-cli
```

Alternatives that should work:

- MySQL Shell via the [docs](#)
- MySQL Server Community Edition: `choco install mysql` or via the [docs](#)

## Install MySQL client on Linux

There are many different Linux distributions so I recommend you check the official MySQL docs for this. If you're using Ubuntu (like we did with our Python-based Dockerfile), you can:

```
sudo apt-update && sudo apt-get install -y mysql-client
```

Alternatives:

- MySQL Client (docker): `sudo apt-update && sudo apt-get install -y default-libmysqlclient-dev`
- MySQL Shell `sudo apt-update && sudo apt-get install mysql-shell` or via the [docs](#)
- `sudo apt-update && sudo apt-get -y install mysql-server`

## 3. Use MySQL Client to Create Database & Database User

We need to create a database for our Django project in our MySQL cluster; this database will be created by using the `linroot` user set for our Linode Database Cluster. To ensure better security, we will create a database user specifically for our Django project during development.

It's important that you *do not* use the root user on your database for security reasons.

### Connect to Your Database

Login to your database via MySQL Client:

Assuming these settings:

- *host*: `lin-cfe-mysql-primary.servers.linodeb.net`
- *user*: `linroot`
- *password*: `not-a-real-one`

```
mysql -h lin-cfe-mysql-primary.servers.linodeb.net -u linroot -p --ssl-mode=required
```

After you enter this command you should see:

```
Enter password:
```

Enter your password, typing will be hidden.

If the password is correct, you should see:

```
Your MySQL connection id is 65929
Server version: 8.0.26 MySQL Community Server - GPL

Copyright (c) 2000, 2022, Oracle and/or its affiliates.

Oracle is a registered trademark of Oracle Corporation and/or its
affiliates. Other names may be trademarks of their respective
owners.

Type 'help;' or '\h' for help. Type '\c' to clear the current input statement.
mysql>
```

For brevity in the book, I will just show the following for each MySQL command:

mysql

```
mysql>
```

If you're new to SQL or MySQL, a few notes you should know:

- CAPITALIZATION in commands matters, when in doubt, try to use ALL CAPS
- Quoting is funny, when in doubt use a single quote ( `'` ) or a tick ( `\`` )
- End all commands with a semicolon ( `;` )
- The commands you'll see in this book will be thoroughly tested and re-tested so you don't have to become an expert in MySQL

## Create database & database user within the MySQL shell

Now we're going to create our database and database user for django. I'll use the following settings:

- **name:** `cfeblog_dev_db`
- **user:** `cfeblog_dev_db_user`
- **user password:** `J-10TDQ8hky-vdKDcksKgsDfrTgj0y0kbeKUfGwgX8w` (generated with `python -c "import secrets;print(secrets.token_urlsafe(32))"`)
- **timezone:** `+00:00` -- This is also the `UTC` timezone and the default timezone in Django.

mysql

```
mysql> CREATE DATABASE IF NOT EXISTS cfeblog_dev_db CHARACTER SET utf8mb4 COLLATE
utf8mb4_0900_ai_ci;
mysql> CREATE DATABASE IF NOT EXISTS cfeblog_dev_db_user IDENTIFIED BY
'J-10TDQ8hky-vdKDcksKgsDfrTgj0y0kbeKUfGwgX8w';
mysql> SET GLOBAL time_zone = "+00:00";
mysql> GRANT ALL PRIVILEGES ON cfeblog_dev_db.* TO cfeblog_dev_db_user;
```

At this point, if everything above was successful, I would normally update my `.env` to match these values. In this guide, I will do that at a later step to provide additional context.

## Verify user permissions

Now let's verify this all worked at least to a degree:

```
mysql -h lin-cfe-mysql-primary.servers.linodeb.net -u cfeblog_dev_db_user -p
--ssl-mode=required
```

You should see:

```
Enter password:
```

After you enter your new password:

```
Welcome to the MySQL monitor.  Commands end with ; or \g.  
Your MySQL connection id is 66038  
Server version: 8.0.26 MySQL Community Server - GPL  
  
Copyright (c) 2000, 2022, Oracle and/or its affiliates.  
  
Oracle is a registered trademark of Oracle Corporation and/or its  
affiliates. Other names may be trademarks of their respective  
owners.  
  
Type 'help;' or '\h' for help. Type '\c' to clear the current input statement.  
  
mysql>
```

Let's try a command we should *not* have permission to do with `cfefblog_dev_db_user` :

SQL

```
mysql> CREATE DATABASE please_dont_work;
```

We should see:

```
ERROR 1044 (42000): Access denied for user 'cfefblog_dev_db_user'@'%' to database  
'please_dont_work'
```

How about a command that works:

SQL

```
mysql> USE cfefblog_dev_db;
```



Now we can look for tables within this database with:

```
SHOW TABLES;
```

At this point, we have no tables in this database and we should receive *no errors* with the last two commands. You *could* experiment with my SQL now and create tables and insert data. We're not going to do that. Instead, we'll opt to let Django manage creating our tables (via `python manage.py migrate`) and insert our data when we save model instances.

## 4. Django Test Database in MySQL

When we run `python manage.py test` Django, by default, will create and destroy a new database specifically for testing. We need to enable this permission for our MySQL user.

**RECOMMENDED** To give specific permissions to CREATE/DROP a test database:

sql

```
mysql> GRANT ALL PRIVILEGES ON `test_cfelog\_%`.* TO cfelog_dev_db_user;
```

Notice that I used `test_cfelog\_%`, why is this? This syntax will match *any database* that starts with `test_cfelog`. Django, by default, will prepend `test_` to our database name when we run `python manage.py test` so `test_cfelog` will solve that.

We *could* use `test\_%` instead of `test_cfelog\_%` but then that would give this user permission to *other* Django projects we might use with this test database, and I don't want that.

**NOT RECOMMENDED** To give unlimited permissions to a user, you can run:

sql

```
mysql> GRANT ALL PRIVILEGES ON *.* TO cfelog_dev_db_user;
```

In general, granting unlimited permissions is something we want to avoid, even at the development level.

### Revoking Permissions (Reference)

To revoke *all permissions*:

sql

```
mysql> REVOKE ALL PRIVILEGES ON *.* FROM cfeblog_dev_db_user;
```

If you revoke all permissions, you will have to re-grant permissions for our primary database ( `cfeblog_dev_db` from the previous step):

sql

```
mysql> GRANT ALL PRIVILEGES ON cfeblog_dev_db.* TO cfeblog_dev_db_user;
```

To revoke just the *test database permissions*:

sql

```
mysql> REVOKE ALL PRIVILEGES ON `test_cfeblog`.* FROM cfeblog_dev_db_user;
```

### Review User Permissions (Reference)

Here's a simple command to see what GRANTS (permissions) a user has in a MySQL database:

sql

```
mysql> SHOW GRANTS FOR cfeblog_dev_db_user;
```

## 5. Required SQL Commands Reference

sql

```
CREATE DATABASE IF NOT EXISTS cfeblog_dev_db CHARACTER SET utf8mb4 COLLATE
utf8mb4_0900_ai_ci;
CREATE USER IF NOT EXISTS cfeblog_dev_db_user IDENTIFIED BY
'J-10TDQ8hky~vdKDcksKgsDfrTgj0y0kbeKUfGwgX8w';
SET GLOBAL time_zone = "+00:00";
GRANT ALL PRIVILEGES ON cfeblog_dev_db.* TO cfeblog_dev_db_user;
GRANT ALL PRIVILEGES ON `test_cfeblog\_%`.* TO cfeblog_dev_db_user;
```

You can store this data as a `.sql` file. To do so, use the folder `db-init/` and the file `init.sql` and execute it with `mysql` at any time with:

```
mysql < db-init/init.sql
```

Or, a more full example:

```
mysql -h lin-cfe-mysql-primary.servers.linodeb.net -u linroot -p
--ssl-mode=required < db-init/init.sql
```

Even better, we can do create a file called `mysql-config.cfg` in `db-init/` :

```
[client]
user=linroot
password=your-database-password
host=lin-cfe-mysql-primary.servers.linodeb.net
port=3306
```

And then do:

```
mysql --defaults-extra-file=db-init/mysql-config.cfg --ssl-mode=required <
db-init/init.sql
```

The `--defaults-extra-file` must be the first argument.

Be sure to add `db-init/` to `.gitignore` if you decide to use this method. We'll be building off this idea in the next chapter.

## 6. Update `.env` or Environment Variables for Your Django Project

According to everything that was provisioned above, I will update (or add) the following values in `.env`:

```
DATABASE_BACKEND=mysql
MYSQL_DATABASE=cfeblog_dev_db
MYSQL_USER=cfeblog_dev_db_user
MYSQL_PASSWORD=J-10TDQ8hky-vdKDcksKgsDfrTgj0y0kbeKUfGwgX8w
MYSQL_ROOT_PASSWORD=J-10TDQ8hky-vdKDcksKgsDfrTgj0y0kbeKUfGwgX8w
MYSQL_TCP_PORT=3306
MYSQL_HOST=lin-cfe-mysql-primary.servers.linodeb.net
MYSQL_DB_REQUIRE_SSL=true
```

`.env` is loaded into our Django project using a third-party package called [python-dotenv](#). It's a simple way to manage your environment variables on your local machine through Django.

These values correspond to the following Django database settings (found in `settings.py`). We have our `mysql` database settings for MySQL in `cfeblog/settings/db/mysql.py` based on what we did in [Getting Started](#) and after we cloned [the cfe-django-blog repo](#).

python

```
import os

from django.conf import settings

BASE_DIR = settings.BASE_DIR
DEBUG = settings.DEBUG

MYSQL_USER = os.environ.get("MYSQL_USER")
MYSQL_PASSWORD = os.environ.get(
    "MYSQL_ROOT_PASSWORD"
) # using the ROOT User Password for Local Tests
MYSQL_DATABASE = os.environ.get("MYSQL_DATABASE")
MYSQL_HOST = os.environ.get("MYSQL_HOST")
MYSQL_TCP_PORT = os.environ.get("MYSQL_TCP_PORT")
MYSQL_DB_IS_AVAIL = all(
    [MYSQL_USER, MYSQL_PASSWORD, MYSQL_DATABASE, MYSQL_HOST, MYSQL_TCP_PORT]
)
MYSQL_DB_CERT = BASE_DIR / "certs" / "db.crt"

if MYSQL_DB_IS_AVAIL:
    DATABASES = {
        "default": {
            "ENGINE": "django.db.backends.mysql",
            "NAME": MYSQL_DATABASE,
            "USER": MYSQL_USER,
            "PASSWORD": MYSQL_PASSWORD,
            "HOST": MYSQL_HOST,
            "PORT": MYSQL_TCP_PORT,
        }
    }
}

if MYSQL_DB_CERT.exists() and not DEBUG:
    DATABASES["default"]["OPTIONS"] = {"ssl": {"ca": f"{MYSQL_DB_CERT}"}}
```

If you decide to put these in `settings.py` feel free to remove:

python

```
from django.conf import settings

BASE_DIR = settings.BASE_DIR
DEBUG = settings.DEBUG
```

## 7. Django Commands

In our [cloned Django project repo](#), we have backup data under the directory `fixtures/`. (In Django, this backup data is called fixtures.) In this portion, we're going to load these fixtures (backup data) into our database cluster:

### Run, Migrate, & Verify Django

```
python manage.py migrate
```

This will create all the tables needed in our database ( `cfeblog_dev_db` ) assuming we granted the correct permissions in the previous steps.

### Load Fixtures

```
python manage.py loaddata fixtures/auth fixtures/articles
```

### Run the Server

```
python manage.py runserver
```

Now open <http://127.0.0.1:8000>, do you see something that resembles a blog?

## Run Tests

```
python manage.py test
```

Was it successful?

## Login to The Admin

```
python manage.py runserver
```

Now open <http://127.0.0.1:8000/admin/> with the following:

- *username:* **admin**
- *password:* **thisisdjango**

These credentials come from the fixtures that we loaded a few steps ago. If they fail, just create a new user with:

```
python manage.py createsuperuser
```

## 8. Fresh Start

Every once and a while you need to destroy your database and start again. When I am doing tests, I do this constantly.

### Create a Backup:

First and foremost, Linode automatically backs up your database clusters for you which, of course, is one of the many benefits of using a managed service.

Every once in a while you might want a one-off backup of your own -- even if it's just for learning purposes.

First, let's create a backups folder that we will omit from git:

```
mkdir -p backups
echo "\nbackups/" >> .gitignore
```

The `\n` in front of `backups/` is to ensure a newline is created.

#### Option 1: Inline

```
mkdir -p backups

mysqldump -h lin-cfe-mysql-primary.servers.linodeb.net -u cfeblog_dev_db_user -p
--ssl-mode=required cfeblog_dev_db > backups/mybackupdb.sql echo "\nbackups/" >>
.gitignore
```

This should prompt you for the password that we set above. After this command completes, you will have a new `mybackupdb.sql` file containing your Django database backup.

You can also use `mysql-config.cfg` if you set that up too:

#### Option 2: Using Config File

```
mysqldump --defaults-extra-file=db-init/mysql-config.cfg --ssl-mode=required cfeblog_
dev_db > backups/mybackupdb.sql
```

Either of these options will only backup the database this user has access to and/or the one we specified: `cfeblog_dev_db`.



To use this backup (reload or restore), you can review the [GitHub Action for migrating a database](#) in the [next chapter](#).

## 1. Drop database in MySQL

We should log in to the root user to drop our database (depending on the permissions you gave the non-root user above):

```
mysql -h lin-cfe-mysql-primary.servers.linodeb.net -u linroot -p --ssl-mode=required
```

SQL

```
mysql> DROP DATABASE cfeblog_dev_db;
```

That's it. The database is gone. Django can no longer log in and store data.

## 2. Review MySQL

Let's verify `cfeblog_dev_db` is no longer available:

### Review Databases

SQL

```
mysql> SHOW DATABASES;
```

### Review user grants (permissions)

SQL

```
mysql> SHOW GRANTS FOR cfeblog_dev_db_user;
```

Do you see:

SQL

```
mysql> GRANT ALL PRIVILEGES ON `cfeblog_dev_db`.* TO `cfeblog_dev_db_user`@`%`
```

Do you see:

SQL

```
mysql> GRANT ALL PRIVILEGES ON `cfefblog_dev_db`. * TO `cfefblog_dev_db_user`@`%`
```

We only deleted the database, we did *not* delete the permission to manage the database. Pretty neat right?

### 3. Re-create Database

SQL

```
mysql> CREATE DATABASE IF NOT EXISTS cfefblog_dev_db CHARACTER SET utf8mb4 COLLATE  
utf8mb4_0900_ai_ci;  
Query OK, 1 row affected (0.04 sec)
```

We should not have to change the timezone since we did a global change in previous sections. Just verify the UTC time with:

sql

```
mysql> SELECT UTC_TIMESTAMP();
```

And compare to now:

sql

```
mysql> SELECT now();
```

Looking good? Now exit:

```
mysql> exit;  
Bye
```

## 4. Go Back to Section 7. Django Commands Section And Repeat

Chapter 5

# Managed MySQL & Django in Production

## Chapter 5

# Managed MySQL & Django in Production

In the last chapter, we set up a managed MySQL database cluster to work with our Local Development environment. In this chapter, we'll set it up to work in our production environment.

This is the shortest chapter of the book for several reasons:

1. MySQL can be complicated, but configuring a database with one user is not
2. We did a lot of heavy lifting for MySQL & our Django project in the last chapter
3. GitHub Actions helps ensure what we do in this chapter works well *every time*

## Create a New MySQL Cluster

Now that we're preparing our project for production, we'll use Linode's recommended 3 Node cluster.

1. Log in to Linode
2. Create MySQL Cluster:

Now, we'll create a cluster for our production workflow.

**Cluster Label:** `cfe-db-prod`

**Database Engine:** `MySQL 8 (currently v8.0.26)`

**Region:** (Select one near you)

**Shared CPU:** `Linode 2 GB`

**Plan:** `DBaaS MySQL - Linode 2GB`

**Number of Nodes:** `3`

Under **Add Access Controls** add:

```
0.0.0.0/0
```

To configure this database cluster using the GitHub Action below, we'll use `0.0.0.0/0` and after we complete the MySQL configuration you should **remove** `0.0.0.0/0` as it allows *any IP Address* to attempt to connect to this cluster. And yes, you need to update the **Access Control** list to include the IP Addresses of your Instances.

3. Click Create Cluster.
4. Wait 15-30 minutes (it can take this long for the cluster to provision)
5. Download Certificate

We'll delete this file shortly because we do not need it to remain on our local machine. If you decide that you need it again, you can always re-download it.

Just remember that if you do decide to keep this file locally, *never commit it with Git*.

6. Copy Contents of Certificate into GitHub Actions

**Key:** `MYSQL_DB_CERT`

**Value:** Enter something that resembles:

```
-----BEGIN CERTIFICATE-----
MIIDQzCCAiugAwIBAgIJALBU6zR0XJzzMA0GCSqGSIb3DQEBCwUAMDcxExARBgNV
BAMMCmxpbm9kZS5jb20xEzARBgNVBAoMA0GCSqGSIb3DQEBCwUAMDcxExARBgNVD
MCAXDTIyMA0GCSqGSIb3DQEBCwUAMDcxExARBgNVDEQ1WjA3MRMwEQYDVQDDAps
aW5vZGUuY29tMRMwEQYDVMA0GCSqGSIb3DQEBCwUAMDcxExARBgNVEDSDzCCASIw
DQYJKoZIhvcNAQEBBQADggEPADCCAQoCggEBALjvCV+UqcXFQqpCr5vRBqMksXbc
AayBjgiS+CLQUVPLzAkn805vTHfZmeE3Y7i07wRrGVS22DUB1/Y66jJkMaX6dxmE
B7zp/BCTKEtmd3+J2TS645N23htI6s+3zYg4m8Z6SuRZyWZU1foOT1D4/9puNeVZ
Ccqp3+ZdLanMA0GCSqGSIb3DQEBCwUAMDcxExARBgNVDASDFkMA0GCSqGSIb3DQ
FMrSAdVEixh/rxOUdumHJ8aYITZBMAwGA1UdEwQFMAMBAf8wDQYJKoZIhvcNAQEL
BQADggEBABWfqZ+teQdjnf99mXyHMxGJf5kvPwyp29dxArOL1HUxpqPjY960dIRg
f+1G0+BkwCMA0GCSqGSIb3DQEBCwUAMDcxExARBgNVDEMOiFemheb0eSZp19jp5U
hEKahLBjMA0GCSqGSIb3DQEBCwUAMDcxExARBgNV323MA0GCSqGSIb3DQEBCwUAM
Wh2mgycTtvSxrMiSFC8JDey+Hu7pzH0RDiU1X5/0Wx31XFauEPLhaA1U08/QzU3a
uPb8M+FHkaSzXUNT9b7hiaehlg0XAZBcSbynLfGIkNdMsRGWqUCsWS2wfU17sYim
J1PR3bF5QMfxidf0SiGVoimPLVN074c=
-----END CERTIFICATE-----
```

Be sure to copy the entire contents, including any trailing spaces.

## Required GitHub Action Secrets for MySQL Database Configuration

First off, we need to add all of the other credentials to GitHub Actions for our automated database configuration to work.

Here are the keys we're going to set:

- `MYSQL_DB_ROOT_USER`
- `MYSQL_DB_ROOT_PASSWORD`
- `MYSQL_DB_HOST`
- `MYSQL_DB_PORT`
- `MYSQL_DATABASE`
- `MYSQL_USER`
- `MYSQL_PASSWORD`

**Key:** `MYSQL_DB_ROOT_USER`

**Value:** `linroot`

You'll find the root user in the Linode console.

**Key:** `MYSQL_DB_ROOT_PASSWORD`

**Value** *some-val-in-console*

You'll find the root user password in the Linode console.

**Key:** `MYSQL_DB_HOST`

**Value** *(from linode console)*

This will resemble something like: `lin-3032-3038-mysql-primary-private.servers.linodeb.net`

**Key:** `MYSQL_DB_PORT`

**Value** `3306` *(but verify in linode console)*

Port `3306` is the standard MySQL port.

**Key:** `MYSQL_DATABASE`

**Value** `cfe_django_blog_db` *(you pick a valid SQL database name)*

The easiest way to write valid SQL database name is to use basic Latin letters, digits 0-9, dollar, underscore ( `[0-9,a-z,A-Z$_]` ). To read more about this topic, [visit here](#).

**Key:** `MYSQL_USER`

**Value** `cfe_django_blog_user` *you pick a valid SQL user name*

A valid SQL user name consists of alphanumeric characters and underscores. You can read more about it [here](#).

**Key:** `MYSQL_PASSWORD`

**Value** *create one*

Create a strong password. When in doubt, use the SQL functions [here](#) to validate your password within MySQL. Here's how I create a strong password:

### macOS/Linux

```
python3 -c "import secrets;print(secrets.token_urlsafe(32))"
```

### Windows

Assuming your Python3.10 is stored under `C:\`, you can run:

```
C:\Python310\python.exe -c "import secrets;print(secrets.token_urlsafe(32))"
```

This will yield a new secret over and over. Something like:

- `s8W2w_NLg0K_46BWr4gXlBr9NvR52NbFBU8uT_MSwaA`
- `3Fnju743wDCb0gsp2FQD-57Fobknn1z9bdrKptsKBwk`
- `sDu9Nd8be14h1KQ_f7l-BXTJYmB4S1iSunIiJhjSUEU`

## Configure MySQL Database with GitHub Actions Workflow

Now we'll implement an automated version of [this section](#) from the last chapter. We're essentially going to run a `.sql` script to create our Django database.

We *should* only have to run this one time, so we do not need to integrate it into `all.yaml`. This is also why it only responds to a `workflow_dispatch`.

Also notice that we have zero code being checked out here; that's because this is an easily portable workflow that you can use again and again.

And now the workflow, let's create the workflow to put the previous keys into action.

```
.github/workflows/mysql-init.yaml
```

yaml

```
name: MySQL Client to Configure Linode Database

on:
  workflow_dispatch:

jobs:
  mysql_client:
    runs-on: ubuntu-latest
    steps:
      - name: Update packages
        run: sudo apt-get update
      - name: Install mysql client
        run: sudo apt-get install mysql-client -y
      - name: Mysql Version
        run: echo $(mysql --version)
      - name: Config Create
        run: |
          cat << EOF > db-config
          [client]
          user=${{ secrets.MYSQL_DB_ROOT_USER }}
          password=${{ secrets.MYSQL_DB_ROOT_PASSWORD }}
          host=${{ secrets.MYSQL_DB_HOST }}
          port=${{ secrets.MYSQL_DB_PORT }}
          EOF
      - name: Add SSL Cert
        run: |
          cat << EOF > db.crt
          ${{ secrets.MYSQL_DB_CERT }}
          EOF
```



```
- name: SQL Init Script
run: |
  cat << EOF > db-init.sql
  CREATE DATABASE IF NOT EXISTS ${ secrets.MYSQL_DATABASE }} CHARACTER SET
    utf8mb4 COLLATE utf8mb4_0900_ai_ci;
  CREATE USER IF NOT EXISTS ${ secrets.MYSQL_USER }} IDENTIFIED BY
    '${ secrets.MYSQL_PASSWORD }}';
  SET GLOBAL time_zone = "+00:00";
  GRANT ALL PRIVILEGES ON ${ secrets.MYSQL_DATABASE }}.* TO ${
    secrets.MYSQL_USER }};
  GRANT ALL PRIVILEGES ON \`test_${ secrets.MYSQL_DATABASE }}\_%\`.* TO
    ${ secrets.MYSQL_USER }};
  EOF
- name: Run Command
run: mysql --defaults-extra-file=db-config --ssl-ca=db.crt < db-init.sql
```

As mentioned in the last chapter, the `--defaults-extra-file` must be the first argument.

After you create this file locally, perform the following steps

1. `git add .github/workflows/mysql-init.yaml`
2. `git commit -m "Added MySQL Init Workflow"`
3. Log in to your repo
4. Navigate to Actions, manually `MySQL Client to Configure Linode Database`
5. Verify your run shows either of the following:

## Successful Initial Run

```
mysql_client
succeeded 42 seconds ago in 11s

> ✓ Set up job 1s
> ✓ Update packages 8s
v ✓ Install mysql client 0s
  1 ▶ Run sudo apt-get install mysql-client -y
  4 Reading package lists...
  5 Building dependency tree...
  6 Reading state information...
  7 mysql-client is already the newest version (8.0.29-0ubuntu0.20.04.3).
  8 0 upgraded, 0 newly installed, 0 to remove and 48 not upgraded.
v ✓ Mysql Version 0s
  1 ▶ Run echo $(mysql --version)
  4 mysql Ver 8.0.29-0ubuntu0.20.04.3 for Linux on x86_64 ((Ubuntu))
v ✓ Config Create 0s
  1 ▶ Run cat << EOF > db-config
v ✓ Add SSL Cert 0s
  1 ▶ Run cat << EOF > db.crt
v ✓ SQL Init Script 0s
  1 ▶ Run cat << EOF > db-init.sql
v ✓ Run Command 0s
  1 ▶ Run mysql --defaults-extra-file=db-config --ssl-ca=db.crt < db-init.sql
> ✓ Complete job 0s
```

## Future Runs

```
mysql_client
succeeded 23 seconds ago in 11s

> ✔ Set up job 0s
> ✔ Update packages 8s
▼ ✔ Install mysql client 0s
  1 ▶ Run sudo apt-get install mysql-client -y
  4 Reading package lists...
  5 Building dependency tree...
  6 Reading state information...
  7 mysql-client is already the newest version (8.0.29-0ubuntu0.20.04.3).
  8 0 upgraded, 0 newly installed, 0 to remove and 48 not upgraded.
▼ ✔ Mysql Version 0s
  1 ▶ Run echo $(mysql --version)
  4 mysql Ver 8.0.29-0ubuntu0.20.04.3 for Linux on x86_64 ((Ubuntu))
▼ ✔ Config Create 0s
  1 ▶ Run cat << EOF > db-config
▼ ✔ Add SSL Cert 0s
  1 ▶ Run cat << EOF > db.crt
▼ ✔ SQL Init Script 0s
  1 ▶ Run cat << EOF > db-init.sql
▼ ✔ Run Command 0s
  1 ▶ Run mysql --defaults-extra-file=db-config --ssl-ca=db.crt < db-init.sql
> ✔ Complete job 0s
```

## Solving Errors

If everything is configured correctly, the above workflow should work every time. That said, errors can almost certainly still occur. Here are a few ideas on how to solve errors with the `mysql_init.yaml` workflow:

- If you're getting a connection error, check that you're using the correct MySQL cluster `host` and not the `private network host` from the Linode console.
- If you see `Unable to get private key` in the errors, be sure to check you're using the flag `--ssl-ca` and not `--ssl-cert` with your `mysql` commands.
- Manually configure the database and database user for Django with what we did in [the previous chapter](#).
- Use [nektos/act](#) to test workflow locally.
- Use the MySQL Docker container's bash (command line) to perform everything manually.

If you continue to have errors, please submit an issue to the official final project GitHub repo at: <https://github.com/codingforentrepreneurs/deploy-django-linode-mysql>

## From One MySQL Database to Another with GitHub Actions

An underlying theme of this entire book is portability. This section will show you how to build on what we did above and migrate a MySQL Database to another. Database migration, of course, is a key to portability so we'll create a GitHub Action to help us with this migration.

Databases (and a database dump), can be pretty large. Keep in mind that GitHub Actions is currently restricted to 14GB of SSD disk space, [read more about it here](#). If you want to use more than 14GB, consider using a [self-hosted runner](#) which we discuss in [Appendix D](#).

yaml

```

name: Migrate from one MySQL Database Cluster to Another

on:
  workflow_dispatch:

jobs:
  mysql_client:
    runs-on: ubuntu-latest
    steps:
      - name: Update packages
        run: sudo apt-get update
      - name: Install mysql client
        run: sudo apt-get install mysql-client -y
      - name: Mysql Version
        run: echo $(mysql --version)
      - name: Current Host Config
        run: |
          cat << EOF > db-config-current
          [client]
          user=${{ secrets.MYSQL_DB_ROOT_USER }}
          password=${{ secrets.MYSQL_DB_ROOT_PASSWORD }}
          host=${{ secrets.MYSQL_DB_HOST }}
          port=${{ secrets.MYSQL_DB_PORT }}
          EOF
      - name: New Host Config
        run: |
          cat << EOF > db-config-new
          [client]
          user=${{ secrets.MYSQL_DB_ROOT_USER }}
          password=${{ secrets.MYSQL_DB_ROOT_PASSWORD }}
          host=${{ secrets.MYSQL_DB_HOST }}
          port=${{ secrets.MYSQL_DB_PORT }}
          EOF
      - name: Current Host SSL Cert
        run: |
          cat << EOF > db-current.crt
          ${ secrets.MYSQL_DB_CERT }
          EOF

```

```
- name: New Host SSL Cert
  run: |
    cat << EOF > db-new.crt
    ${ secrets.MYSQL_DB_CERT }
    EOF
- name: Run Command
  run: mysqldump -defaults-extra-file=./db-config-current --ssl-cert=db-current.
    crt --all-databases - --result-file=dump.sql
- name: Load Dump into New Database
  run: mysql --defaults-extra-file=./db-config-new --ssl-cert=db-new.crt
    < dump.sql
```

You can use this workflow as a reference whenever you need to upgrade your database cluster *or* migrate it to a Linode Managed Database.

In some migration cases, you will *not* need a certificate to perform this migration.

## Practical Notes on Using Terraform for Database Creation

In Chapter 8, we'll cover how to use Terraform to automate creating *most* of our infrastructure with Linode. The part that's missing from that chapter? MySQL.

You absolutely can use Terraform to provision your MySQL database, but then we have another challenge: updating our GitHub Action secrets with the contents of our database certificate. While this challenge is pretty easy to solve with the GitHub CLI and [Appendix E](#), it is outside the scope of this book at this time due to unnecessarily added complexity.

## Next Steps with MySQL

In our [Ansible Chapter](#), we will be updating a GitHub Action Workflow to ensure that our database is connected correctly to our Django project. Assuming our [Configure MySQL Database with GitHub Actions workflow](#) section went well, you should be done and ready for the next chapter.

Chapter 6

# Linode Object Storage for Django

## Chapter 6

# Linode Object Storage for Django

In this section, we're going to implement Linode Object Storage as our Static File server for *any* Django project.

## Related Resources

- How to manage static files ([Django Docs](#))
- How to deploy static files ([Django Docs](#))
- Django Storages [Docs](#)

## Motivation & Background

Django uses an MVT Architecture, which consists of simple Python data models (how Django syncs with a Database), views (how Django responds to URL requests), and templates (how Django renders out the response to a URL request).

Any file that is *not* required for Django's MVT to work but is necessary for the Web App to work, we'll consider a static file. These files include:

- Images
- JavaScript
- Cascading Style Sheets (CSS)
- CSV files (for download)
- PDF files (for download)
- Videos

Make no mistake, Django can *produce* these files with the right package but *serving* these files is something Django should not do. As the docs state:

**[Using Django static views] is grossly inefficient and probably insecure, so it is unsuitable for production.**

Instead of using Django, you'll want to use a dedicated server and/or an external service for serving these types of files. In this section, we'll explore setting up Linode Object Storage to serve these files on our behalf. Object Storage is an S3-compatible storage solution.

Managing a dedicated server for our static files is possible and can be very interesting to implement, but it adds unnecessary complications and costs since storage solutions like Linode Object Storage exist.



# 1. Development Static File Serving

There are two primary states for serving static files:

- Development
- Production

During development, Django can serve the static files using Django static views.

`cfeproj/urls.py`

python

```
from django.conf import settings
from django.conf.urls.static import static

urlpatterns = [
    # ... the rest of your URLconf goes here ...
]

if settings.DEBUG:
    urlpatterns += static(settings.STATIC_URL, document_root=settings.STATIC_ROOT)
    urlpatterns += static(settings.MEDIA_URL, document_root=settings.MEDIA_ROOT)
```

A bit more context on the above code:

- `cfeproj` is just an arbitrary name for a Django project (created with `django-admin startproject cfeproj` )
- `STATIC_URL` and `STATIC_ROOT` are set in your main settings file ( `cfeproj/settings.py` )
- `MEDIA_URL` and `MEDIA_ROOT` are set in your main settings file ( `cfeproj/settings.py` )
- `STATIC_ROOT` are static files uploaded via `python manage.py collectstatic` (automatically uploaded)
- `MEDIA_ROOT` are static files uploaded via `FileField` or `ImageField` (user uploaded)
- `cfeproj/urls.py` is your primary URLs file (in this case, `ROOT_URLCONF = "cfeproj.urls"` )

As the [Django docs state](#), this method is *grossly inefficient* and *probably insecure*, so it is **unsuitable for production**. The only time I use this method in development is when my static files change state constantly (i.e I'm using a React.js fronted with my Django backend). I typically opt for the production method (next section) even while in development. In some cases, I have a production bucket on Linode Object Storage as well as a development bucket to help isolate the two environments.

## 2. Production Static File Serving

Here we'll setup basic configuration for [django-storages](#) in preparation for Linode Object Storage.

### Install `django-storages`

We need to install two packages:

- `boto3` ([docs](#)): This allows any Python project to connect Linode Object Storage.
- `django-storages` ([docs](#)): This configures Django specifically to use Object Storage and `boto3`.

```
$(venv) python install pip install django-storages boto3
```

A few things to note:

- If this command is unfamiliar/new to you, please review [1 - Getting Started](#)
- Unlike many Django packages, you should not need to add `django-storages` or `boto3` to `INSTALLED_APPS` (in `settings.py`).

### Environment Variables Configuration

Here are the environment variables we will configure soon:

```
DJANGO_STORAGE_SERVICE=linode
LINODE_BUCKET=
LINODE_BUCKET_REGION=
LINODE_BUCKET_ACCESS_KEY=
LINODE_BUCKET_SECRET_KEY=
```

Let's break these down:

- `DJANGO_STORAGE_SERVICE` is a simple flag I use so I can change Object Storage service(s) in any project; it gives me the flexibility to choose.
- `LINODE_BUCKET` is the name of the Linode Object Storage bucket we'll create later in this chapter.
- `LINODE_BUCKET_REGION` is the region of our Linode Object Storage bucket (the physical location of the servers that hold our static files).
- `LINODE_BUCKET_ACCESS_KEY` is a public API key for accessing our bucket (we'll create this after we create our bucket).
- `LINODE_BUCKET_SECRET_KEY` is a private API key for accessing our bucket (we'll create this after we create our bucket).

We use environment variables for these items for a few reasons:

- Security: we don't want to expose these values to the world -- just to our current server/runtime.
- Flexibility: we want to be able to turn these on/off at any time as well as change these values at anytime.
- Reusability: creating Python code that leverages environment variables over hard-coded values improves reusability.

## 3. Setup Linode Object Storage

Now we're going to create our Linode Object Storage Bucket. Remember, think of a storage bucket as an *external hard drive in the cloud* that you can use to store almost anything. The only way to connect to buckets is by using an **Access Key** and **Secret Key** along with a client like Python, Terraform, or many other S3-Compatible tools.

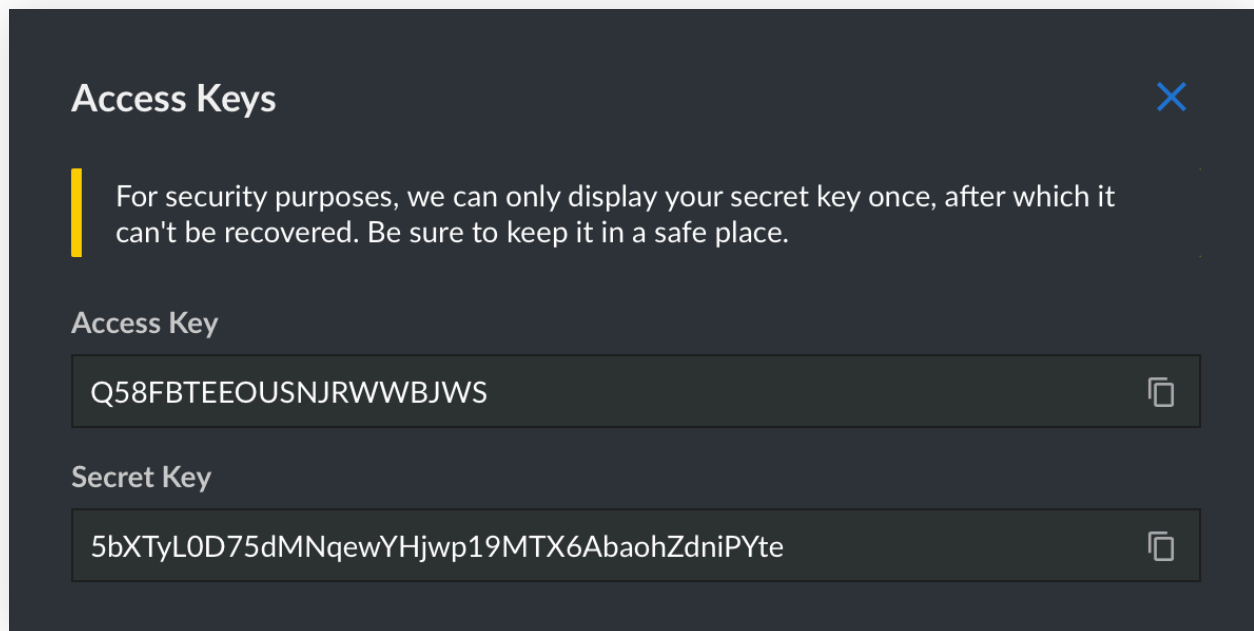
At the end of this project, we'll have 2 separate buckets in Linode Object Storage. In this section, we'll set up a bucket specifically for our Django project. In the next chapter, we will set up yet another bucket for Terraform.

Here's the process for setting up a bucket:

1. Login to [Linode](#)
2. Navigate to [Object Storage](#)
3. Click **Create Bucket**
  - a. Add a label, such as: **lets-deploy-django** (change as this value must be unique)
  - b. Add a region, such as **Atlanta, GA** -- ideally the location is nearest to you, or your end-users
  - c. Click **Create Bucket**
4. Create an API Key

Now that we have `lets-deploy-django` in the `us-southeast-1` region ( `Atlanta, GA` ), let's create an API Key.

1. Go to [Object Storage](#)
2. Select the tab for `Access Keys`
3. Click `Create Access Key`
4. Add a label, such as `My Production Key for Django Deployment`
5. Select `limited access` (To be more secure, always limit your access keys if you can)
6. Locate your bucket and check `Read/Write` access
7. Click `Create Access Key` under the list of permissions
8. After a moment, copy the two keys available `Access Key` and `Secret Key` . It will look something like:



1. Update your Development `.env` file and add the following

```
DJANGO_STORAGE_SERVICE=linode
LINODE_BUCKET=Bucket from Linode
LINODE_BUCKET_REGION=Region from Linode
LINODE_BUCKET_ACCESS_KEY=Access Key From Linode
LINODE_BUCKET_SECRET_KEY=Secret Key from Linode
```

My keys from the above example look like this:

```
DJANGO_STORAGE_SERVICE=linode
LINODE_BUCKET=lets-deploy-django
LINODE_BUCKET_REGION=us-southeast-1
LINODE_BUCKET_ACCESS_KEY=Q58FBTEE0USNJRWWBJWS
LINODE_BUCKET_SECRET_KEY=5bXTyL0D75dMNqewYHjwp19MTX6AbaohZdniPYte
```

2. Navigate to your GitHub repo
3. Navigate to Action Secrets  
( <https://github.com/your-username/your-repo/settings/secrets/actions> )
4. Add the keys from above one by one

## 4. Minimal Configuration for `django-storages`

In this section, I'll show you how to configure `django-storages` with a minimal configuration. For how I configure `django-storages`, see the next section.

Here's the Linode Object Storage configuration:

`cfeproj/settings.py`

python

```
import os

DJANGO_STORAGE_SERVICE = os.environ.get("DJANGO_STORAGE_SERVICE")
LINODE_BUCKET = os.environ.get("LINODE_BUCKET")
LINODE_BUCKET_REGION = os.environ.get("LINODE_BUCKET_REGION")
LINODE_BUCKET_ACCESS_KEY = os.environ.get("LINODE_BUCKET_ACCESS_KEY")
LINODE_BUCKET_SECRET_KEY = os.environ.get("LINODE_BUCKET_SECRET_KEY")
LINODE_OBJECT_STORAGE_READY = all([LINODE_BUCKET, LINODE_BUCKET_REGION,
    LINODE_BUCKET_ACCESS_KEY, LINODE_BUCKET_SECRET_KEY])

if DJANGO_STORAGE_SERVICE == "linode" and LINODE_OBJECT_STORAGE_READY:
    AWS_S3_ENDPOINT_URL = f"https://{LINODE_BUCKET_REGION}.linodeobjects.com"
    AWS_ACCESS_KEY_ID = LINODE_BUCKET_ACCESS_KEY
    AWS_SECRET_ACCESS_KEY = LINODE_BUCKET_SECRET_KEY
    AWS_S3_REGION_NAME = LINODE_BUCKET_REGION
    AWS_S3_USE_SSL = True
    AWS_STORAGE_BUCKET_NAME = LINODE_BUCKET
    AWS_DEFAULT_ACL="public-read"

# USER Uploaded Media/Static Files like MEDIA_ROOT
# ie FileField and ImageField
DEFAULT_FILE_STORAGE = "storages.backends.s3boto3.S3Boto3Storage"

# DJANGO Uploaded Media/Static Files like STATIC_ROOT
# ie python manage.py collectstatic
STATICFILES_STORAGE = "storages.backends.s3boto3.S3Boto3Storage"
```

A few things to note with this code:

- You see `AWS_` (like `AWS_S3_ENDPOINT_URL` and `AWS_STORAGE_BUCKET_NAME`) because Linode Object Storage is S3 compatible. `boto3` (and `boto`) is maintained by Amazon Web Services (AWS) and since `django-storages` leverages `boto3` (in this case), we must use the `boto3` configuration.
- `AWS_DEFAULT_ACL="public-read"` means that all uploaded files (both `STATIC_ROOT` and `MEDIA_ROOT`) will be public. If you want to see ACL options check the [How I do it](#) section below
- `storages.backends.s3boto3.S3Boto3Storage` comes directly from `django-storages`

## 5. Advanced Configuration for `django-storages`

In this section, I'll show you an advanced configuration and how I use `django-storages` across all my projects. This complexity is not always necessary but it gives me flexibility since sometimes I use Linode Object Storage, other times I serve via NGINX, and sometimes I use AWS S3. This section, along with `django-storages`, makes it easy to switch cloud providers.

First, let's create our own `storages` module:

```
mkdir -p cfeproj/storages/services/linode/
echo "" > cfeproj/storages/__init__.py
echo "" > cfeproj/storages/services/__init__.py
```

- `cfeproj` is my main Django configuration folder (where `settings.py` is)
- `cfeproj/storages` will contain my `django-storages` config
- `cfeproj/storages/services` will contain all of the S3-compatible services I want support for (Linode, AWS, and others)

### Customize the `django-storage` Backends

In the minimal example, we have the following settings:

- `DEFAULT_FILE_STORAGE = "storages.backends.s3boto3.S3Boto3Storage"`
- `STATICFILES_STORAGE = "storages.backends.s3boto3.S3Boto3Storage"`

Both of these settings will upload files to the *root* of our Object Storage bucket. This is not ideal for a few reasons:

- You may override files on accident
- You may confuse user-generated content with developer-generated content
- Archiving or moving content is much more complex
- Implementing a proper Content Delivery Network (CDN) can be more complex

What we want is:

- All Django static files in 1 Location ( `static/` )
- All default media files (uploaded via a FileField/Image field) in 1 Location ( `media/` )
- One-off media file storage (such as `private/` or `reports/` etc)

To do this, we need to subclass `S3Boto3Storage` like so:

`cfeproj/storages/backends.py`

python

```
from storages.backends.s3boto3 import S3Boto3Storage

class PublicS3Boto3Storage(S3Boto3Storage):
    location = 'static'

class MediaS3Boto3Storage(S3Boto3Storage):
    location = 'media'

class PrivateS3Boto3Storage(S3Boto3Storage):
    location = 'private'
```

Now we have 3 storage options that map to:

- `static/` → `https://{LINODE_BUCKET_REGION}.linodeobjects.com/static/`  
→ because of `PublicS3Boto3Storage`
- `media/` → `https://{LINODE_BUCKET_REGION}.linodeobjects.com/media/`  
→ because of `MediaS3Boto3Storage`
- `private/` → `https://{LINODE_BUCKET_REGION}.linodeobjects.com/private/`  
→ because of `PrivateS3Boto3Storage`



Before we change the Django project defaults, we can use each of these storage options in a FileField/ImageField like the following example:

python

```
# exampleapp/models.py
from django.conf import settings
from cfproject.storages.backends import PrivateS3BotoStorage

User = settings.AUTH_USER_MODEL

def handle_private_file(instance, filename):
    return f"{instance.user}/{filename}"

class ExampleModel(models.Model):
    user = models.ForeignKey(User, on_delete=models.CASCADE)
    private = models.FileField(upload_to=handle_private_file,
                              storage=PrivateS3BotoStorage())
    public_image = models.ImageField(storage=PublicS3Boto3Storage())
    media_image = models.ImageField(storage=PrivateS3BotoStorage())
```

## Access Control List(s) `acl` Mixin

In the minimal example, we have the following setting:

- `AWS_DEFAULT_ACL="public-read"`

This means that `_all_` of our storage backends are `public-read` (not-private) which means we have:

- `PublicS3Boto3Storage = public-read`
- `MediaS3BotoStorage = public-read`
- `PrivateS3BotoStorage = public-read`
- `storages.backends.s3boto3.S3Boto3Storage = public-read`

When we want:

- `PublicS3Boto3Storage = public-read`
- `MediaS3BotoStorage = private`
- `PrivateS3BotoStorage = private`
- `storages.backends.s3boto3.S3Boto3Storage = public-read`

Access Control List ( **ACL** ) is how we grant permission to various objects in Object Storage. The available options are listed [here](#) but the ones I use most are:

- **private** : requires a signed key to access
- **public-read** : anyone can access

Now, we're going to create a mixin so we can do the following:

python

```
class PublicS3Boto3Storage(S3Boto3Storage):
    location = 'static'
    default_acl = 'public-read'

class MediaS3Boto3Storage(S3Boto3Storage):
    location = 'media'
    default_acl = 'private'

class PrivateS3Boto3Storage(S3Boto3Storage):
    location = 'private'
    default_acl = 'private'
```

At the time of this writing, **django-storages** does not support this feature natively.

Below is the mixin that we can use for the above settings:

`cfeproject/storages/mixins.py`

python

```
class DefaultACLMixin():
    """
    Adds the ability to change default ACL for objects
    within a 'S3Boto3Storage' class.

    Useful for having
    static files being public-read by default while
    user-uploaded files being private by default.

    # CANNED ACL Options come from
    # https://docs.aws.amazon.com/AmazonS3/latest/userguide/acl-overview.html#canned-acl
    """

    CANNED_ACL_OPTIONS = [
        'private',
        'public-read',
        'public-read-write',
        'aws-exec-read',
        'authenticated-read',
        'bucket-owner-read',
        'bucket-owner-full-control'
    ]

    def get_default_settings(self):
        _settings = super().get_default_settings()
        _settings['default_acl'] = self.get_default_acl()
        return _settings

    def get_default_acl(self):
        _acl = self.default_acl or None
        if _acl is not None:
            if _acl not in self.CANNED_ACL_OPTIONS:
                acl_options = "\n\t".join(self.CANNED_ACL_OPTIONS)
                raise Exception(f"The default_acl of \"{_acl}\" is invalid. Please use one
                    of the following:\n{acl_options}")
        return _acl
```

Now let's update our backends:

`cfeproj/storages/backends.py`

python

```
from storages.backends.s3boto3 import S3Boto3Storage

from . import mixins

class PublicS3Boto3Storage(mixins.DefaultACLMixin, S3Boto3Storage):
    location = 'static'
    default_acl = 'public-read'

class MediaS3Boto3Storage(mixins.DefaultACLMixin, S3Boto3Storage):
    location = 'media'
    default_acl = 'private'

class PrivateS3Boto3Storage(mixins.DefaultACLMixin, S3Boto3Storage):
    location = 'private'
    default_acl = 'private'
```

You can now have a per-storage `acl` setting that overrides the `AWS_DEFAULT_ACL="public-read"` configuration.

## Configure Linode Module

cfeproj/storages/services/linode.py

python

```
import os

LINODE_BUCKET = os.environ.get("LINODE_BUCKET")
LINODE_BUCKET_REGION = os.environ.get("LINODE_BUCKET_REGION")
LINODE_BUCKET_ACCESS_KEY = os.environ.get("LINODE_BUCKET_ACCESS_KEY")
LINODE_BUCKET_SECRET_KEY = os.environ.get("LINODE_BUCKET_SECRET_KEY")
LINODE_OBJECT_STORAGE_READY = all([LINODE_BUCKET, LINODE_BUCKET_REGION,
    LINODE_BUCKET_ACCESS_KEY, LINODE_BUCKET_SECRET_KEY])

if LINODE_OBJECT_STORAGE_READY:
    AWS_S3_ENDPOINT_URL = f"https://{LINODE_BUCKET_REGION}.linodeobjects.com"
    AWS_ACCESS_KEY_ID = LINODE_BUCKET_ACCESS_KEY
    AWS_SECRET_ACCESS_KEY = LINODE_BUCKET_SECRET_KEY
    AWS_S3_REGION_NAME = LINODE_BUCKET_REGION
    AWS_S3_USE_SSL = True
    AWS_STORAGE_BUCKET_NAME = LINODE_BUCKET
    AWS_DEFAULT_ACL="private" #private is the default

    DEFAULT_FILE_STORAGE = "cfeproj.storages.backends.MediaS3BotoStorage"
    STATICFILES_STORAGE = "cfeproj.storages.backends.PublicS3Boto3Storage"
```

The key part is that `django-storages` only works if `LINODE_OBJECT_STORAGE_READY` is `True`. `LINODE_OBJECT_STORAGE_READY` will only be `True` if all variables are actually available in environment variables. (`django-storages` will throw errors if any configuration fails)

## Final Settings for Advanced Configuration

`cfeproj/storages/conf.py`

python

```
import os

DJANGO_STORAGE_SERVICE = os.environ.get("DJANGO_STORAGE_SERVICE")

if DJANGO_STORAGE_SERVICE is not None:
    """
    Set default options from django-storages
    if DJANGO_STORAGE_SERVICE key exists
    """
    # USER UPLOADED MEDIA
    DEFAULT_FILE_STORAGE = "storages.backends.s3boto3.S3Boto3Storage"
    # Staticfiles
    STATICFILES_STORAGE = "storages.backends.s3boto3.S3Boto3Storage"

if DJANGO_STORAGE_SERVICE == 'linode':
    from .services.linode import * # noqa
```

Now update `cfeproj/settings.py` :

python

```
STATIC_URL = "/static/"
STATIC_DIRS = [BASE_DIR / "staticfiles"]
STATIC_ROOT = BASE_DIR / "staticroot"
MEDIA_URL = "/media/"
MEDIA_ROOT = BASE_DIR / "mediafiles"

# use django-storages to serve & host
from .storages.conf import * # noqa
```

Be sure to keep `STATIC_ROOT` and `MEDIA_ROOT` above `from .storages.conf import *` so we have a fallback for `python manage.py collectstatic`

## 6. GitHub Actions for Django & Object Storage

Let's create a GitHub Action for automatically collecting our static files to the Linode Object storage:

In `./.github/workflows/staticfiles.yaml` add:

yaml

```
name: 3 - Static Files for Django

# Controls when the workflow will run
on:
  # Allows you to call this workflow within another workflow
  workflow_call:
    secrets:
      LINODE_BUCKET:
        required: true
      LINODE_BUCKET_REGION:
        required: true
      LINODE_BUCKET_ACCESS_KEY:
        required: true
      LINODE_BUCKET_SECRET_KEY:
        required: true
    workflow_dispatch:

# A workflow run is made up of one or more jobs that can run sequentially or in
parallel
jobs:
  # This workflow contains a single job called "build"
  django_staticfiles:
    # The type of runner that the job will run on
    runs-on: ubuntu-latest
    # Add in environment variables for the entire "build" job
    env:
      GITHUB_ACTIONS: true
      DATABASE_BACKEND: mysql
      DJANGO_SECRET_KEY: just-a-test-database
      DJANGO_STORAGE_SERVICE: linode
      LINODE_BUCKET: ${ secrets.LINODE_BUCKET }
      LINODE_BUCKET_REGION: ${ secrets.LINODE_BUCKET_REGION }
      LINODE_BUCKET_ACCESS_KEY: ${ secrets.LINODE_BUCKET_ACCESS_KEY }
      LINODE_BUCKET_SECRET_KEY: ${ secrets.LINODE_BUCKET_SECRET_KEY }
```

```

steps:
  # Checks-out your repository under $GITHUB_WORKSPACE, so your job can access it
  - name: Checkout code
    uses: actions/checkout@v2
  - name: Setup Python 3.10
    uses: actions/setup-python@v2
    with:
      python-version: "3.10"
  - name: Install requirements
    run: |
      pip install -r requirements.txt
  - name: Collect Static
    run: |
      python manage.py collectstatic --noinput

```

If you see an error for:

```

if not VALID_BUCKET.search(bucket) and not VALID_S3_ARN.search(bucket):
TypeError: expected string or bytes-like object

```

There's a good chance your `LINODE_BUCKET` was not set correctly.

## 7. Update `all.yaml` Workflow in GitHub Actions

In `./.github/workflows/all.yaml`, add the following to the `jobs` block just after the `build_container` job.

yaml

```

collectstatic:
  needs: test_django
  uses: ./.github/workflows/staticfiles.yaml
  secrets:
    LINODE_BUCKET: ${ secrets.LINODE_BUCKET }
    LINODE_BUCKET_REGION: ${ secrets.LINODE_BUCKET_REGION }
    LINODE_BUCKET_ACCESS_KEY: ${ secrets.LINODE_BUCKET_ACCESS_KEY }
    LINODE_BUCKET_SECRET_KEY: ${ secrets.LINODE_BUCKET_SECRET_KEY }

```



Chapter 7

# Using Terraform to Provision Infrastructure

## Chapter 7

# Using Terraform to Provision Infrastructure

Now we need the actual hardware to run our Django project and we'll use Linode Instances/Virtual Machines. Turning on a Linode instance is simple:

1. Log in to Linode
2. Click **Create Linode**
3. Add your configuration
4. You're done

You probably know there's a bit more to it than just these 4 steps, but that's the *gist* of it. The nuance of this *gist* is why tools like Terraform exist -- no more guessing what exactly was spun up, when, and by whom. If we need to scale resources up or down, it can be as simple as changing a single variable in our GitHub Actions secrets.

Terraform will help us with:

- Setting the kind of virtual machine to provision (Ubuntu 18.04, Ubuntu 20.04, CentOS, or other Linux operating systems)
- The number of instances to run
- What region to run in
- How much disk space does each instance need
- Adding in the ssh keys and root user password

All of the above steps are important to also *track* over time through version control ( **git** ) and update automatically when changes happen (through **GitHub Actions** ).

In this chapter, we're going to set up our minimal infrastructure -- mostly just our virtual machines -- using Terraform so that we're nearly ready to start running our Django project. After we use Terraform to spin things up, we'll use Ansible to configure the Linode Instances to run the software we to run. We'll cover Ansible in the next chapter and I cover both in more detail in my series *Try Infrastructure as Code*.

# 1. Install Terraform

All official installation options are [here](#).

**macOS** via [Homebrew](#)

**bash**

```
brew tap hashicorp/tap
brew install hashicorp/tap/terraform
```

**Windows** via [Chocolatey](#)

**powershell**

```
choco install terraform
```

**Linux** (Ubuntu / Debian)

**bash**

```
sudo apt-get update && sudo apt-get install -y gnupg software-properties-common curl
curl -fsSL https://apt.releases.hashicorp.com/gpg | sudo apt-key add -
sudo apt-add-repository "deb [arch=amd64] https://apt.releases.hashicorp.com
$(lsb_release -cs) main"
sudo apt-get update && sudo apt-get install terraform
```

## 2. Dedicated Terraform and Ansible Folders

In the root of your Django project (the repo we cloned in [getting started](#)) create the following folders:

- `devops/tf`
- `devops/ansible`

We'll use these folders over the next two chapters. The `devops/tf` folder needs to exist in this way so we can initialize our Terraform project. Now, let's talk about storing Terraform state.

Once we are all complete, `.gitignore` will include the following:

- `keys/`
- `devops/tf/terraform.tfvars`
- `devops/tf/backend`

You can either add these now or when we do it later in this chapter.

### 3. Set Up Linode Object Storage for Terraform

Since Terraform is an open-source IaC tool, we have to configure it to work with the many providers it supports. In our case, Terraform is not directly connected to everything inside of our Linode account. We need to give it permission, and we need a cloud-based way to store the Terraform state.

Terraform monitors the state of the infrastructure that we have spun up through Terraform. This means that Terraform keeps a record of what should be running based on what our Terraform files ( `.tf` ) declare. You can read more about Terraform state [here](#).

We'll use Linode Object Storage as our cloud-based storage location for the Terraform State -- this is important because it allows us to leverage GitHub Actions to handle all things we need Terraform to do. Without a cloud-based storage location for the Terraform state, automating our entire CI/CD pipeline would be incredibly difficult.

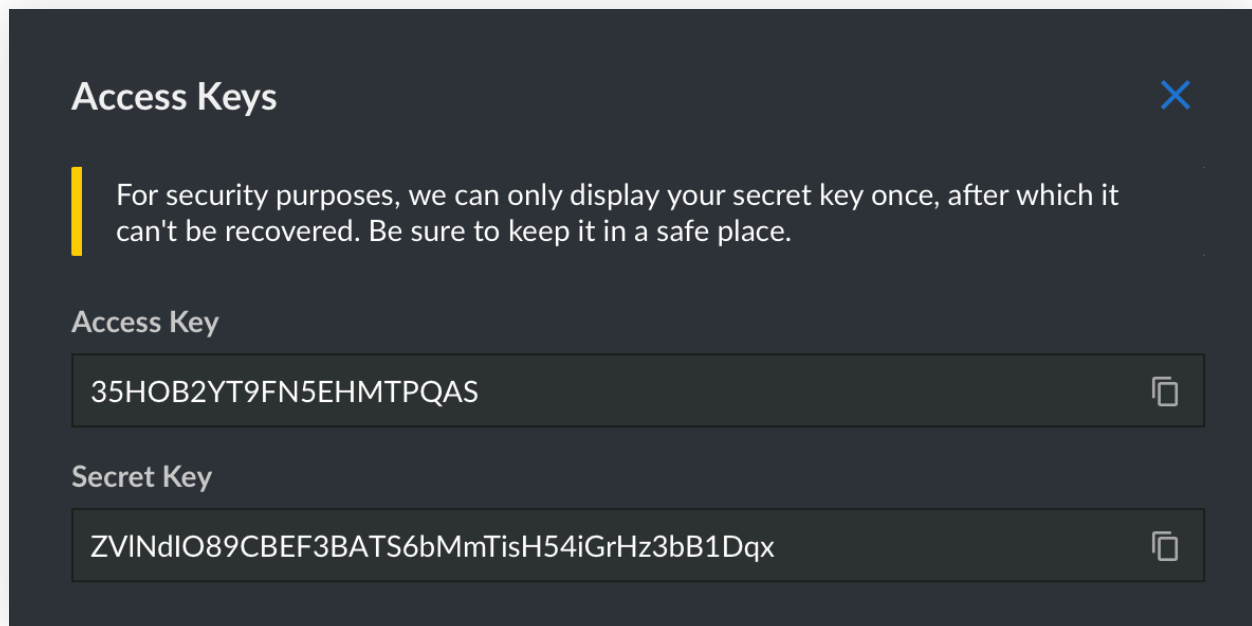
Creating a Linode bucket for Terraform is the *same* process in the chapter [Object Storage for Django](#) for [setting up Linode Object Storage](#).

Here's what we'll do this time around (change configuration):

1. Log in to [Linode](#)
2. Navigate to [Object Storage](#)
3. Click **Create Bucket**
4. Add a label, such as: `lets-store-infra` -- change as this value must be unique and this *must* be different than the Django bucket
5. Add a region, such as `Atlanta, GA` -- ideally the location is near your or your end-users; this location *can* be different than the Django bucket
6. Click **Create Bucket**
7. Create an API Key

Now that we have `lets-store-infra` in the `us-southeast-1` region ( `Atlanta, GA` ), let's create an API Key:

1. Go to [Object Storage](#)
2. Select the tab for `Access Keys`
3. Click `Create Access Key`
4. Add a label, such as `My Production Key for Terraform and Django CI/CD`
5. Select `limited access` (To be more secure, always limit your access keys if you can)
6. Locate your bucket, and check `Read/Write` access.
7. Click `Create Access Key` (under the list of permissions)
8. After a moment, copy the two keys available `Access Key` and `Secret Key` . It will look something like:



As a side note, we *can* use Terraform to manage our Linode Buckets but I prefer not to, primarily so I do not accidentally delete *a lot* of stored content by running 1 Terraform command.

1. Update your Development `.env` file and add the following:

```
LINODE_OBJECT_STORAGE_DEVOPS_BUCKET=Bucket from Linode
LINODE_OBJECT_STORAGE_DEVOPS_BUCKET_ENDPOINT="Region from Linode".linodeobjects.com
LINODE_OBJECT_STORAGE_DEVOPS_ACCESS_KEY=Access Key From Linode
LINODE_OBJECT_STORAGE_DEVOPS_SECRET_KEY=Secret Key from Linode
```

My keys from the above example look like this:

```
LINODE_OBJECT_STORAGE_DEVOPS_BUCKET=lets-store-infra
LINODE_OBJECT_STORAGE_DEVOPS_BUCKET_ENDPOINT=us-southeast-1.linodeobjects.com
LINODE_OBJECT_STORAGE_DEVOPS_ACCESS_KEY=35H0B2YT9FN5EHMTPQAS
LINODE_OBJECT_STORAGE_DEVOPS_SECRET_KEY=ZVlNdI089CBEF3BATS6bMmTisH54iGrHz3bB1Dqx
```

2. Navigate your GitHub repo
3. Navigate to Action secrets  
( <https://github.com/your-username/your-repo/settings/secrets/actions> )
4. Add the keys from above one by one
5. Also be sure to add:

**Key:** `LINODE_OBJECT_STORAGE_DEVOPS_TF_KEY`

**Value:** `your_project_name.tfstate` (or what we set below as `django-infra.tfstate` )

## 4. Create backend and Init Project

Now that we have a bucket, we need to tell Terraform that we want to store our state in this bucket. We do this before creating our Terraform project *because* it's the easiest way to do so.

Now, create the file `backend` in `devops/tf/backed` with the following:

```
skip_credentials_validation = true
skip_region_validation = true
bucket="your_bucket"
key="your_project_name.tfstate"
region="us-southeast-1"
endpoint="us-southeast-1.linodeobjects.com"
access_key="your_object_storage_public_key"
secret_key="your_object_storage_secret_key"
```

Here's an example using the bucket created in the last section:

```
skip_credentials_validation = true
skip_region_validation = true
bucket="lets-store-infra"
key="django-infra.tfstate"
region="us-southeast-1"
endpoint="us-southeast-1.linodeobjects.com"
access_key="35HOB2YT9FN5EHMTPQAS"
secret_key="ZV1NdIO89CBEF3BATS6bMmTisH54iGrHz3bB1Dqx"
```

Right now, update `.gitignore` to include `devops/tf/backend` because we have several secrets that we must keep secret.

```
echo "devops/tf/backend" >> .gitignore
```

I hope you see this step and wonder "How am I going to add this to GitHub Actions?" If you're starting to think in this way, that's great! If you're not, that's okay, I think you just need more practice. A core theme behind this book is to constantly think "how do I automate this step" which often translates to "how do I put this in GitHub Actions" -- it took me a while to truly start thinking this way but once I did, I never looked back. It's also why this book exists.

## 5. Initialize Terraform

The standard command is `terraform init` but for simplicity, we're keeping our Terraform code alongside our Django code. In this case, we'll use the following command:

```
terraform -chdir=devops/tf/ init -backend-config=backend
```

This command will:

- Initialize a new Terraform project
- Use `devops/tf` as the root of our Terraform project
- use `devops/tf/backend` as the backend config (yes `-backend-config=backend` is correct because of `-chdir=devops/tf` )

## 6. Provision Options & `terraform.tfvars`

In this project, we're going to employ basic horizontal scaling coupled with an NGINX load balancer. Horizontal scaling, in this case, is simply the process of adding more virtual machines running more versions of Django. Docker is perfectly suited for horizontal scaling by its ephemeral nature. Don't get me wrong, we can horizontally scale to help with the load too, but we're going to start with the minimum provision options to get this project into production.

Here's what we're going to set up:

- (2) 2GB or (3) 2GB Linode Instances running our Docker-based Django App
- (1) 4GB or (1) 8GB Linode Instance running NGINX for Load Balancing (horizontal scale as needed)
- (1) 2GB Linode Instance to run our Docker-based Redis Instance
- (1) 2GB Linode Instance to run our Docker-based Django & Celery Instance

Here's what Terraform cares about:

- Group A: (2) 2GB or (3) 2GB Linode Instances
- Group B: (1) 4GB or (1) 8GB Linode Instance
- Group C: (1) 2GB Linode Instance
- Group D: (1) 2GB Linode Instance

Here's what Ansible (next chapter) cares about:

- Group A: Docker-based Django App
- Group B: NGINX for Load Balancing
- Group C: Docker-based Redis
- Group D: Docker-based Django & Celery

I think this illustrates how Ansible and Terraform work in harmony together -- Terraform turns things on/off and Ansible configures them to work in a specific way.

These groups (names, what resources they need, and what software) have been decided specifically for this book but they can be virtually anything and in virtually any configuration you may need. You can also easily add or remove services as needed.



## Create `terraform.tfvars`

Now, let's create the `devops/tf/terraform.tfvars` file and get all of the related content. This is a resource file that contains all of the variable values your Terraform project may need (including secrets). I tend to think of these as input arguments to all of the `.tf` files.

Location: `devops/tf/terraform.tfvars`

```
linode_pa_token="your_linode_persona_access_token"
authorized_key="your_public_ssh_key"
root_user_pw="your_root_user_password"
app_instance_vm_count=1
linode_image="linode/ubuntu20.04"
```

Right now, update `.gitignore` to include `devops/tf/terraform.tfvars` because we have several secrets that we must keep secret.

```
echo "devops/tf/terraform.tfvars" >> .gitignore
```

In GitHub Action secrets, add:

Key: `LINODE_IMAGE`

Value: `linode/ubuntu20.04`

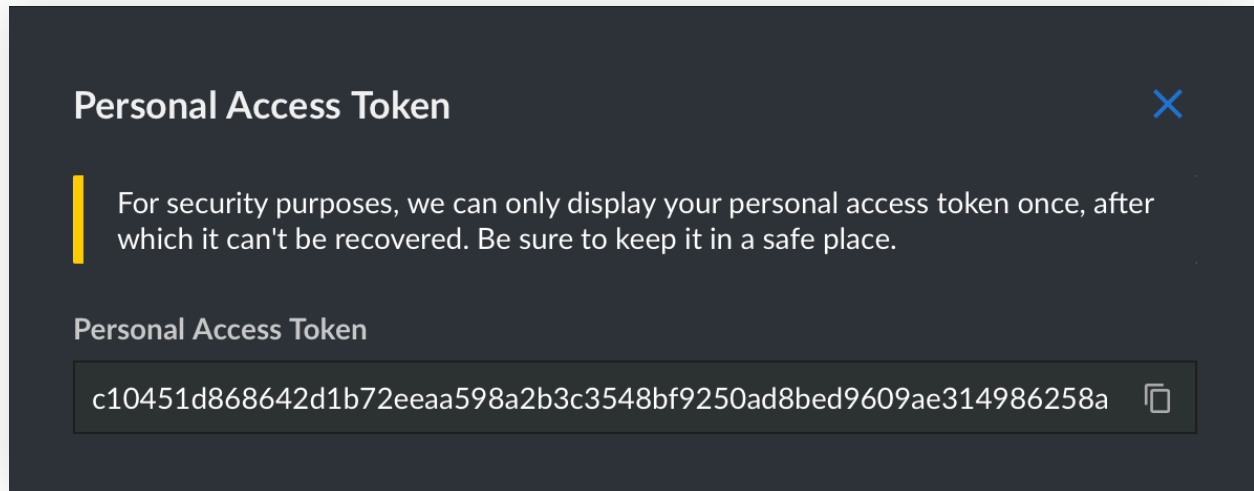
## Create a Linode Personal Access token ( `linode_pa_token` )

1. Login to Linode
2. Navigate to [API Tokens](#)
3. Click **Create a Personal Access Token**
4. Under **Label** add **Deploy Django Terraform PAT**
5. Expiry select **In 1 month** (yeah, do the *shortest* possible; these are the keys to your kingdom)
6. For the services,
  - **Select All:** Select **None** so everything is turned off. If I don't have a service listed below, we don't need it.
  - **Domains:** Select **Read/Write**
  - **Images:** Select **Read**
  - **IPs:** Select **Read/Write**
  - **Linodes:** Select **Read/Write**
  - **NodeBalancers:** Select **Read/Write**
  - **Object Storage:** Select **Read**
  - **Volumes:** Select **Read/Write**

Here's what I have:

Access	None	Read Only	Read/Write
Select All	<input checked="" type="radio"/>	<input type="radio"/>	<input type="radio"/>
Account	<input checked="" type="radio"/>	<input type="radio"/>	<input type="radio"/>
Domains	<input type="radio"/>	<input type="radio"/>	<input checked="" type="radio"/>
Events	<input checked="" type="radio"/>	<input type="radio"/>	<input type="radio"/>
Images	<input type="radio"/>	<input type="radio"/>	<input checked="" type="radio"/>
IPs	<input type="radio"/>	<input type="radio"/>	<input checked="" type="radio"/>
Kubernetes	<input checked="" type="radio"/>	<input type="radio"/>	<input type="radio"/>
Linodes	<input type="radio"/>	<input type="radio"/>	<input checked="" type="radio"/>
Longview	<input checked="" type="radio"/>	<input type="radio"/>	<input type="radio"/>
NodeBalancers	<input type="radio"/>	<input type="radio"/>	<input checked="" type="radio"/>
Object Storage	<input type="radio"/>	<input checked="" type="radio"/>	<input type="radio"/>
StackScripts	<input checked="" type="radio"/>	<input type="radio"/>	<input type="radio"/>
Volumes	<input type="radio"/>	<input type="radio"/>	<input checked="" type="radio"/>

7. Click **Create Token** and you should see:



8. In GitHub Action secrets, add:

Key: **LINODE\_PA\_TOKEN**

Value: **c10451d868642d1b72eeaa598a2b3c3548bf9250ad8bed9609ae314986258aa9**

Naturally, your value will *definitely* be different.

1. In your local **devopts/tf/terraform.tfvars** add your token like:

```
linode_pa_token="c10451d868642d1b72eeaa598a2b3c3548bf9250ad8bed9609ae314986258aa9"  
...
```

## Create new SSH keys for GitHub Actions and Terraform

Creating SSH keys is easy on macOS, Linux, and Windows. It's just `ssh-keygen`. Now, if you just run `ssh-keygen` you will be prompted for all kinds of options. Instead, let's do something more predictable for us all:

### 1. Generate Public / Private SSH Keys

```
mkdir -p keys/  
ssh-keygen -f keys/tf-github -t rsa -N ''  
echo "keys/" >> .gitignore
```

These 3 lines will:

- Create a directory in our project called `keys`
- Generate 2 new ssh keys called `tf-github` and `tf-github.pub` in the folder `keys/`
- Add `keys/` to your `.gitignore` file

`tf-github.pub` (this is your public key) will contain something like:

```
ssh-rsa AAAAB3NzaC1yc2E....removed on purpose...X0UU= jmitch@justins-mbp.lan
```

`tf-github` (this is your private key) will contain something like:

```
-----BEGIN OPENSSH PRIVATE KEY-----  
b3B1bnNzaC1rZXktdjEAAAABAG5vbmUAAAAEbm9uZQAAAAAAAAABAAABlwAAAAAdzc2gtcn  
....removed on purpose...  
qQ/fzw+EoRlc0BAAAAFmptaXRjaEBqdXN0aW5zLW1icC5sYW4BAgME  
-----END OPENSSH PRIVATE KEY-----
```

### 2. On GitHub Action Secrets, add the following:

Key: `SSH_DEVOPS_KEY_PUBLIC`

Value: (enter the contents of `tf-github.pub`) -- be sure to remove any newlines/line breaks at the end.

Key: `SSH_DEVOPS_KEY_PRIVATE`

Value: (enter the contents of `tf-github`) -- newlines/line breaks are okay.

3. In your local `devops/tf/terraform.tfvars` add your public key only like:

```
...
authorized_key=(enter the contents of `tf-github.pub`)
...
```

4. What's the private key for? Ansible. It's for the next step.

5. Can I delete `tf-github.pub` and `tf-github` after they are on GitHub Action secrets?

You can, but I suggest that you do it after you've run this process a few times.

## Generate a root user password for instance(s)

For `root_user_pw` we're going to add a password that's as secure as possible. Here are a few open-source options to create it:

- Python: `python3 -c "import secrets;print(secrets.token_urlsafe(32))"`
- Bash: `openssl rand -base64 32`
- Django: `python -c 'from django.core.management.utils import get_random_secret_key; print(get_random_secret_key())'`

There's probably a lot more out there. Just pick one and generate a key.

```
...
root_user_pw="kbZAjeicfQLXDzasYIr8pUd2--ceEv3XdoU40JTfyXM"
...
```

After you have this password, update your GitHub Action secrets to:

Key: `ROOT_USER_PW`

Value: `kbZAjeicfQLXDzasYIr8pUd2--ceEv3XdoU40JTfyXM`

## Docker-base Django virtual machine count

For `app_instance_vm_count`, we can decide exactly how many Linodes we want running our Docker-based Django App. The more instances we run, the more expensive it will be (um, duh right?). To start, just use `1`.

In GitHub Actions Secrets, add the following:

Key: `DJANGO_VM_COUNT`

Value: `1`

## 7. Create Terraform Files

Now we're going to create a few Terraform files. Terraform is **declarative** which means that we tell it what we want instead of telling it how we want to get there. For comparison, Django is imperative -- we tell Django how to handle URL routing ( `urls.py` ), data modeling ( `models.py` ), templates, and so on.

In this part, we'll declare a bunch of things in Terraform files describing what we want to happen. That's essentially what IaC tools are: *declare* what you want and if done right, IaC tools will deliver that exact thing.

Here are the all of files we'll have in `devops/tf` :

- `main.tf`
- `variables.tf`
- `terraform.tfvars`
- `locals.tf`
- `linodes.tf`
- `outputs.tf`
- `backend`
- `templates/sample.tpl`
- `templates/ansible-inventory.tpl`
- other auto-generated Terraform files that we don't need to worry about.

Terraform treats all of the `.tf` files as *1 big file*. I named these files in this way to follow convention but as long as `.tf` exists on the file, you can name it what you want; including having all of the code in the files as literally 1 big file.

Here are some quick definitions:

- `main.tf` -- the primary entry point for Terraform
- `variables.tf` - this is where we declare external and internal variables that will be used
- `terraform.tfvars` - this is where we store the input values of the variables that correspond to `variables.tf`
- `locals.tf` - I tend to think of these as `_constants_` or immutable variables. That is, what I put in here typically doesn't change based on what's provided via user input and/or `terraform.tfvars` .
- `linodes.tf` - this is the focus of this chapter, the other files exist to ensure this file runs correctly and with CI/CD automation in mind.
- `outputs.tf` - this is for the output data you may want to display in the command line.

Again, think of these as 1 big file, the order doesn't matter but the *namespace* of each resource does.

The resources are basically like this:

```
resourceType "custom_name" {  
  resourceArg = yourValue  
  resourceArgAgain = yourOtherValue  
  resourceArgVar = var.my_variable  
  resourceArgDir = locals.root_dir  
}
```

The above example is to merely show you the strange format that Terraform uses called HCL (HashiCorp Configuration Language)-- it's almost like a Python diction or JavaScript Object Notion but it's not exactly there for either.

Whenever I get stuck trying to diagnose something not working with HCL or Terraform here's what I try:

- **terraform console** (that's the command line command)
- The Terraform [Docs](#) are a fantastic resource, even for providers like [Linode](#)
- The Terraform section in my Try IaC book and the related [github repo](#); seriously I have to look up my old configurations all the time.

Now, let's start making our files.

**main.tf**Location: `devops/tf/main.tf`

```
terraform {
  required_version = ">= 0.15"
  required_providers {
    linode = {
      source = "linode/linode"
      version = "1.27.2"
    }
  }
  backend "s3" {
    skip_credentials_validation = true
    skip_region_validation = true
  }
}

provider "linode" {
  token = var.linode_pa_token
}
```

- **terraform** : this block is required in every Terraform project so we can declare all providers we'll be using (there are many)
- **required\_provider** > **linode** : This block declares the version of Linode's provider we want to use. Check the official Linode Terraform provider [docs here](#) for the latest.
- **backend "s3"** this is how we leverage Object Storage as our Terraform state backend.
- **provider "linode"** when adding providers, in this case, the **linode** one, we need to provide the personal access token. **var.linode\_pa\_token** will be discussed in the next section.



## variables.tf

Location: `devops/tf/variables.tf`

```
variable "linode_pa_token" {
  sensitive = true
}

variable "authorized_key" {
  sensitive = true
}

variable "root_user_pw" {
  sensitive = true
}

variable "app_instance_vm_count" {
  default = 0
}

variable "linode_image" {
  default = "linode/ubuntu20.04"
}
```

Do any of these variables look familiar to you? They should. They match identically to what we did with `terraform.tfvars`. This, of course, was intentional.

So why do we need `terraform.tfvars` and `variables.tf`? I think of `terraform.tfvars` as the *inputs* where `variables.tf` I consider the *argument* declarations.

Let's say `variables.tf` declares a value without a default? Terraform will *prompt* for user input to provide this value. Neat! Using `sensitive = true` means Terraform will not output these values in the logs. Using `default=something` means we can have a fallback value for items missing from `terraform.tfvars`.

To add to this, `terraform.tfvars` will *not* be added to your Git repo ever. We will generate this file in GitHub Actions right along with the `backend` file.

## locals.tf

Location: `devops/tf/locals.tf`

hcl

```
locals {
  tf_dir = "${abspath(path.root)}"
  templates_dir = "${local.tf_dir}/templates"
  devops_dir = "${dirname(abspath(local.tf_dir))}"
  ansible_dir = "${local.devops_dir}/ansible"
}
```

I use `locals.tf` as a place to initialize *constants* or *hard coded* values. Similar to `const myVar = 'SomeValue'` in JavaScript, `locals.tf` are very often used for things that won't change based on `terraform.tfvars`. The [docs](#) state that "[Locals] can be helpful to avoid repeating the same values or expressions" -- so that's how we'll use it.

In this project, I just store the various filesystem paths that I reference throughout the project. That is what's happening with: `${dirname(abspath(path.root))}`.

The easiest way to understand expressions like this is by using `terraform console` :

- `"${}"` : this how you can do String Substitution (it works a lot like JavaScript). [Docs](#)
- `path.root` [docs](#) points to the `main.tf` since that's where we declared `terraform {}`. This is known as the root module.
- `dirname()` and `abspath()` are both filesystem functions. `dirname` gets the directory name of a path. `abspath()` gets the absolute path of a module on the local file system. So `dirname(abspath(path.root))` should be `/path/to/your/proj/devops` where
- `abspath(path.root)` should be `/path/to/your/proj/devops/tf`
- `local.tf_dir` references *another* variable ( `tf_dir` ) *within* the `locals` block.

Neat!

Here's how you can play around with this data, enter the console with:

```
terraform -chdir=devops/tf/ console
```

and here's what you can do:

```
> path.root
"."
> abspath(path.root)
"/Users/cfe/Dev/django-prod/devops/tf/"
> dirname(path.root)
"."
> dirname(abspath(path.root))
"/Users/cfe/Dev/django-prod/devops/tf/"
```

Having `locals` compute these values for us makes it easier to re-use them in other modules. For example, we'll use this in the next part like this:

- `"${local.templates_dir}/ansible-inventory.tpl"`

and

- `"${local.ansible_dir}/inventory.ini"`

## linodes.tf

In this section, we'll build out the requirements discussed before in [this section](#):

- Group A: (2) 2GB Linode Instances (Django App)
- Group B: (1) 4GB Linode Instance (Load Balancer)
- Group C: (1) 2GB Linode Instance (Redis)
- Group D: (1) 2GB Linode Instance (Celery Worker)

Let's convert each of these into their actual instance types [Docs](#):

- Group A: 2x **g6-standard-1** (Django App)
- Group B: 1x **g6-standard-2** (Load Balancer)
- Group C: 1x **g6-standard-1** (Redis)
- Group D: 1x **g6-standard-1** (Celery Worker)

Now, let's start writing the requirements for the Linode provider in: **devops/tf/linodes.tf** .

We'll start by adding just *Group A* from above. Keep in mind that everything in this resource block contains a lot of items we have already discussed and fleshed out. This is merely the implementation. Let's take a look:

*Group A*

```
resource "linode_instance" "app_vm" {
  count = var.app_instance_vm_count > 0 ? var.app_instance_vm_count : 0
  image = var.linode_image
  region = "us-central"
  type = "g6-standard-1"
  authorized_keys = [ var.authorized_key ]
  root_pass = var.root_user_pw
  private_ip = true
  tags = ["app", "app-node"]
}
```

Let's break this down:

- **"linode\_instance" "app\_vm"** This part is telling Terraform that this is using the provider **Linode**'s resource known as **linode\_instance** [docs](#). **app\_vm**, here is just *naming this resource for Terraform and Terraform-only*. If you need to name the instances themselves on Linode, check out the **label** portion below.
- **count=** Count refers, of course, to the number of instances we want to create with these exact specifications. **var.app\_instance\_vm\_count** references the variable in both **variables.tf** and **terraform.tfvars**. If this number is 0, then Terraform will ensure that 0 instances are running. If the number is larger than 0, then Terraform will ensure exactly that number is running.
- **image = var.linode\_image** Linode has many Image types that can be used [here](#). In this case we reference **linode\_image** in **variables.tf** which as a default value of **linode/ubuntu20.04**. This is great because if we omit **linode\_image** in **terraform.tfvars** it will use this default value. We can use *custom images* here as well but we'll leave that for another time.
- **region = "us-central"** I hard-coded the region here, but this can easily be put in **locals** since we re-use it over and over.
- **authorized\_keys = [ var.authorized\_key ]** this is the public SSH key(s) we want this machine to have
- **root\_pass = var.root\_user\_pw** this is the password we want the default root user to have. In the case of **linode/ubuntu20.04** this root user will have the username: **root** but that's not always true for all image types.
- **private\_ip = true** gives us a private IP address so our instances can communicate within the region's network.
- **tags = ["app", "app-node"]** these tags are optional but nice to have when reviewing the Linode Console.
- (optional); You can also use a **label=** here. If you omit it, Linode will create one for you. I omit it here since Linode requires these labels to be unique. You can make them semi-unique by using something like **label = "app\_vm-\${count.index + 1}"**

*Group B*

```
resource "linode_instance" "app_loadbalancer" {
  image = "linode/ubuntu20.04"
  label = "app_vm-load-balancer"
  region = "us-central"
  type = "g6-standard-2"
  authorized_keys = [ var.authorized_key ]
  root_pass = var.root_user_pw
  private_ip = true
  tags = ["app", "app-lb"]

  lifecycle {
    prevent_destroy = true
  }
}
```

This group is *almost* the same as *Group A* with some minor changes. The notable ones are:

- **image = "linode/ubuntu20.04"** : We hard-coded a image type
- **label = "app\_vm-load-balancer"** : we hard-coded a label here so it's easily findable on the Linode Console.
- **type = "g6-standard-2"** : We use a different Linode type
- **lifecycle { prevent\_destroy = true }** : This parameter prevents Terraform from destroying this instance no matter what. This is absolutely optional but I have it simply because this instance is responsible for *all* traffic into our Django project; thus we don't want to accidentally delete this. To delete it, we'll have to log in to the Linode Console.

Let's finish the remaining two Linode Instance resources:

*Group C*

```
resource "linode_instance" "app_redis" {  
  image = var.linode_image  
  label = "app_redis-db"  
  region = "us-central"  
  type = "g6-standard-1"  
  authorized_keys = [ var.authorized_key]  
  root_pass = var.root_user_pw  
  private_ip = true  
  tags = ["app", "app-redis"]  
  
  lifecycle {  
    prevent_destroy = true  
  }  
}
```

The only key thing here is that we do not want to accidentally destroy our Redis data server via Terraform.

## Group D

```
resource "linode_instance" "app_worker" {
  count=1
  image = var.linode_image
  label = "app_worker-${count.index + 1}"
  region = "us-central"
  type = "g6-standard-1"
  authorized_keys = [ var.authorized_key ]
  root_pass = var.root_user_pw
  private_ip = true
  tags = ["app", "app-worker"]

  depends_on = [linode_instance.app_redis]
}
```

A few keys here:

- **count** As we have seen, this argument is optional when we only want 1 instance. The reason I have this here is so I can easily update my code to *horizontally scale* the worker as needed by changing this value to a different one and/or by abstracting it to a variable as well. In other words, this is here to future-proof this resource.
- **label = "app\_worker-\${count.index + 1}"** this of course corresponds with the fact I have the **count** parameter at all.
- **depends\_on = [linode\_instance.app\_redis]** here's something that's new to us. **linode\_instance.app\_redis** refers directly to the block for *Group C*. This **depends\_on** array ensures that Terraform will not create this resource until what it depends on is created. The portion **linode\_instance.app\_redis** will be discussed in a future section. If you're curious how it works right now, spin up **linode console**.

Full code for **devops/tf/linodes.tf**:

```
resource "linode_instance" "app_vm" {
  count = var.app_instance_vm_count > 0 ? var.app_instance_vm_count : 0
  image = var.linode_image
  region = "us-central"
  type = "g6-standard-1"
  authorized_keys = [ var.authorized_key ]
  root_pass = var.root_user_pw
  private_ip = true
  tags = ["app", "app-node"]
}
```



```
resource "linode_instance" "app_loadbalancer" {
  image = "linode/ubuntu20.04"
  label = "app_vm-load-balancer"
  region = "us-central"
  type = "g6-standard-2"
  authorized_keys = [ var.authorized_key ]
  root_pass = var.root_user_pw
  private_ip = true
  tags = ["app", "app-lb"]

  lifecycle {
    prevent_destroy = true
  }
}

resource "linode_instance" "app_redis" {
  image = var.linode_image
  label = "app_redis-db"
  region = "us-central"
  type = "g6-standard-1"
  authorized_keys = [ var.authorized_key ]
  root_pass = var.root_user_pw
  private_ip = true
  tags = ["app", "app-redis"]

  lifecycle {
    prevent_destroy = true
  }
}

resource "linode_instance" "app_worker" {
  count=1
  image = var.linode_image
  label = "app_worker-${count.index + 1}"
  region = "us-central"
  type = "g6-standard-1"
  authorized_keys = [ var.authorized_key ]
  root_pass = var.root_user_pw
  private_ip = true
  tags = ["app", "app-worker"]

  depends_on = [linode_instance.app_redis]
}
```

## outputs.tf

Location: `devops/tf/outputs.tf`

```
output "instances" {
  value = [for host in linode_instance.app_vm.*: "${host.label} : ${host.ip_address}"]
}

output "loadbalancer" {
  value = "${linode_instance.app_loadbalancer.ip_address}"
}

output "redisdb" {
  value = "${linode_instance.app_redis.ip_address}"
}
```

## Templates

Terraform can render content based on a template. First, let's make a sample template:

Add the following to `devops/tf/templates/sample.tpl`

```
Hi ${ varA },

This is a cool ${ otherVar } thing. Here's some other items:

%{ for loopVar in yetAnother ~}
- ${ loopVar }
%{ endfor ~}
```

This syntax is a lot like Jinja or Django Templates just slightly different. You can find more details [here](#), let's use this file so we can see how it works.

Assuming you have `templates_dir` in your `locals.tf` and your current working directory in the command line is `devops/tf/`, you can run:

```
terraform console
```

then

```
templatefile("${local.templates_dir}/sample.tpl", {varA="Something cool",
  otherVar= 123, yetAnother=["what", "is", "this"]})
```

And you should see:

```
Hi Something cool,

This is a cool 123 thing. Here's some other items:

- what
- is
- this
```

Let's use this template so we can prepare our instances to be managed directly by Ansible (which we'll discuss in the [next chapter](#)).

First, let's create our template file in `devops/tf/templates/ansible-inventory.tpl`

```
[webapps]
{% for host in webapps ~}
${host}
{% endfor ~}

[loadbalancer]
${loadbalancer}

[redis]
${redis}

[workers]
{% for host in workers ~}
${host}
{% endfor ~}
```

Now that we have this, we can use the following `templatefile()` declaration:

```
templatefile("${local.templates_dir}/ansible-inventory.tpl", {
    webapps=[for host in linode_instance.app_vm.*: "${host.ip_address}"]
    loadbalancer="${linode_instance.app_loadbalancer.ip_address}"
    redis="${linode_instance.app_redis.ip_address}"
    workers=[for host in linode_instance.app_worker.*: "${host.ip_address}"]
})
```

Before we continue, let's break this down:

First, I highly recommend you test this in the **terraform console** to get your mind around each argument.

- **templatefile()** we just saw this syntax with a basic example
- **for x in resource.resource\_name.\*** is how we loop through resources that have the **count=** parameter. This loop will *only* work *after* the resources have been provisioned.
- **for host in linode\_instance.app\_vm.\*** is for the **linode\_instance** resource with the name **app\_vm**. Each iteration in this loop is set to the variable **host**.
- **for host in linode\_instance.app\_loadbalancer.\*** will fail because **linode\_instance.app\_loadbalancer** does *not* have the **count=** parameter. This is also true for **host in linode\_instance.app\_redis.\***
- Every resource has different attributes associated to it. In this case, we're using the **ip\_address** attribute for each resource. There are many for the **linode\_instance** such as **status**, **private\_ip\_address**, (and many more [here](#)).
- **linode\_instance.app\_redis** (has no **count=** parameter) and **linode\_instance.app\_vm.0** (has a **count=** parameter) is a single instance. This means we can do stuff like:
- **linode\_instance.app\_vm.0.ip\_address** or **linode\_instance.app\_redis.ip\_address**
- **linode\_instance.app\_vm.0.private\_ip\_address** or **linode\_instance.app\_redis.private\_ip\_address**

Now that we understand how to use **templatefile** with our resources. Let's implement it in a new resource called **local\_file** in **devops/tf/linodes.tf**:

```
resource "local_file" "ansible_inventory" {
  content = templatefile("${local.templates_dir}/ansible-inventory.tpl", {
    webapps=[for host in linode_instance.app_vm.*: "${host.ip_address}"]
    loadbalancer="${linode_instance.app_loadbalancer.ip_address}"
    redis="${linode_instance.app_redis.ip_address}"
    workers=[for host in linode_instance.app_worker.*: "${host.ip_address}"]
  })
  filename = "${local.ansible_dir}/inventory.ini"
}
```

Let's break this down:

- **local\_file** allows Terraform to manage local files on your local machine. This file can be almost *anything*, which is pretty cool.
- **"local\_file" "ansible\_inventory"** is merely to let me know this resource is to manage our Ansible inventory ( **inventory.ini** ) file (more on this file in the [next chapter](#)).
- **filename = "\${local.ansible\_dir}/inventory.ini"** This is the destination of the **local\_file** resource that Terraform will manage 100%. If you change this file outside Terraform, Terraform will change it back after you run the correct commands.

Final & full code for `devops/tf/linodes.tf` :

```
resource "linode_instance" "app_vm" {
  count = var.app_instance_vm_count > 0 ? var.app_instance_vm_count : 0
  image = var.linode_image
  region = "us-central"
  type = "g6-standard-1"
  authorized_keys = [ var.authorized_key ]
  root_pass = var.root_user_pw
  private_ip = true
  tags = ["app", "app-node"]
}

resource "linode_instance" "app_loadbalancer" {
  image = "linode/ubuntu20.04"
  label = "app_vm-load-balancer"
  region = "us-central"
  type = "g6-standard-2"
  authorized_keys = [ var.authorized_key ]
  root_pass = var.root_user_pw
  private_ip = true
  tags = ["app", "app-lb"]

  lifecycle {
    prevent_destroy = true
  }
}

resource "linode_instance" "app_redis" {
  image = var.linode_image
  label = "app_redis-db"
  region = "us-central"
  type = "g6-standard-1"
  authorized_keys = [ var.authorized_key ]
  root_pass = var.root_user_pw
  private_ip = true
  tags = ["app", "app-redis"]

  lifecycle {
    prevent_destroy = true
  }
}
```

```
resource "linode_instance" "app_worker" {
  count=1
  image = var.linode_image
  label = "app_worker-${count.index + 1}"
  region = "us-central"
  type = "g6-standard-1"
  authorized_keys = [ var.authorized_key]
  root_pass = var.root_user_pw
  private_ip = true
  tags = ["app", "app-worker"]

  depends_on = [linode_instance.app_redis]
}

resource "local_file" "ansible_inventory" {
  content = templatefile("${local.templates_dir}/ansible-inventory.tpl", {
    webapps=[for host in linode_instance.app_vm.*: "${host.ip_address}"]
    loadbalancer="${linode_instance.app_loadbalancer.ip_address}"
    redis="${linode_instance.app_redis.ip_address}"
    workers=[for host in linode_instance.app_worker.*: "${host.ip_address}"]
  })
  filename = "${local.ansible_dir}/inventory.ini"
}
```

## 8. Run Commands

### Validate Terraform project files

```
terraform -chdir=devops/tf/ validate
```

This will check if your Terraform files are technically valid. If the files are invalid, Terraform will not attempt to make changes. This command can also help you diagnose errors.

**validate** will not check for errors that may/may not occur with any given resource API. For example, if you give a Linode Instance the same name twice, Terraform will not recognize this as an error but the Linode API will raise an error.

### Apply/Update/Create Terraform Resources

```
terraform -chdir=devops/tf/ apply
```

This will prompt you to approve the changes (requiring command line input) if they are valid changes (as in valid Terraform-based changes).

### Automatically Apply/Update/Create Terraform Resources

```
terraform -chdir=devops/tf/ apply -auto-approve
```

This will *not* prompt you to approve the changes, it will make them (if they are valid).



## Destroy Terraform Resources

```
terraform -chdir=devops/tf/ destroy
```

```
terraform -chdir=devops/tf/ apply -destroy
```

These are two commands doing the same thing. Both will prompt you (requiring command line input) to approve the destruction.

## Automatically Destroy Terraform Resources

```
terraform -chdir=devops/tf/ apply -destroy -auto-approve
```

This will *not* prompt you to destroy all resources provisioned by Terraform.

## Using the Console

```
terraform -chdir=devops/tf/ console
```

This will allow you to test/review your Terraform project and various features of Terraform. It's a good idea to use `-chdir=devops/tf/` to ensure the console is working at the root of your Terraform code.

## 9. Custom Domain (Optional)

If you own a domain name, like `tryiac.com`, we can use Terraform to map this domain to our Load Balancer resource ( `linode_instance.app_loadbalancer` ).

Working through this book I have assumed you do *not* own a custom domain. The nice thing is you can add it whenever you need it. Here's the resource block you would add:

```
resource "linode_domain" "tryiac_com" {
  type = "master"
  domain = "tryiac.com"
  soa_email = "hello@tryiac.com"
  tags = ["app", "app-domain"]
}

resource "linode_domain_record" "tryiac_com_root" {
  domain_id = linode_domain.tryiac_com.id
  name = "@"
  record_type = "A"
  ttl_sec = 300
  target = "${linode_instance.app_loadbalancer.ip_address}"
}

resource "linode_domain_record" "tryiac_com_www" {
  domain_id = linode_domain.tryiac_com.id
  name = "www"
  record_type = "A"
  ttl_sec = 300
  target = "${linode_instance.app_loadbalancer.ip_address}"
}
```

Manually mapping domain names to your IP Address is perfectly reasonable as well.

Be sure to add your custom domain to GitHub Action secrets as:

**KEY:** `ALLOWED_HOST`

**VALUE:** `.yourcustom-domain.com`

For my `ALLOWED_HOST`, I would add `.tryiac.com`; notice the leading period ( `.` ) on this domain name; that's intentional.

## 10. Part 1: GitHub Actions for Infrastructure

Now we're going to create `.github/workflows/infra.yaml` to implement our infrastructure workflow for Terraform:

yaml

```
name: 4 - Apply Infrastructure via Terraform & Ansible

on:
  workflow_call:
    secrets:
      DJANGO_VM_COUNT:
        required: true
      LINODE_IMAGE:
        required: true
      LINODE_OBJECT_STORAGE_DEVOPS_BUCKET:
        required: true
      LINODE_OBJECT_STORAGE_DEVOPS_TF_KEY:
        required: true
      LINODE_OBJECT_STORAGE_DEVOPS_ACCESS_KEY:
        required: true
      LINODE_OBJECT_STORAGE_DEVOPS_SECRET_KEY:
        required: true
      LINODE_PA_TOKEN:
        required: true
      ROOT_USER_PW:
        required: true
      SSH_DEVOPS_KEY_PUBLIC:
        required: true
      SSH_DEVOPS_KEY_PRIVATE:
        required: true

  workflow_dispatch:

jobs:
  terraform:
    runs-on: ubuntu-latest
    steps:
      - name: Checkout
        uses: actions/checkout@v2
      - name: Setup Terraform
        uses: hashicorp/setup-terraform@v1
```

```

with:
  terraform_version: 1.1.9
- name: Add Terraform Backend for S3
  run: |
    cat << EOF > devops/tf/backend
    skip_credentials_validation = true
    skip_region_validation = true
    bucket="{{ secrets.LINODE_OBJECT_STORAGE_DEVOPS_BUCKET }}"
    key="{{ secrets.LINODE_OBJECT_STORAGE_DEVOPS_TF_KEY }}"
    region="us-southeast-1"
    endpoint="us-southeast-1.linodeobjects.com"
    access_key="{{ secrets.LINODE_OBJECT_STORAGE_DEVOPS_ACCESS_KEY }}"
    secret_key="{{ secrets.LINODE_OBJECT_STORAGE_DEVOPS_SECRET_KEY }}"
    EOF
- name: Add Terraform TFVars
  run: |
    cat << EOF > devops/tf/terraform.tfvars
    linode_pa_token="{{ secrets.LINODE_PA_TOKEN }}"
    authorized_key="{{ secrets.SSH_DEVOPS_KEY_PUBLIC }}"
    root_user_pw="{{ secrets.ROOT_USER_PW }}"
    app_instance_vm_count="{{ secrets.DJANGO_VM_COUNT }}"
    linode_image="{{ secrets.LINODE_IMAGE }}"
    EOF
- name: Terraform Init
  run: terraform -chdir=./devops/tf init -backend-config=backend
- name: Terraform Validate
  run: terraform -chdir=./devops/tf validate -no-color
- name: Terraform Apply Changes
  run: terraform -chdir=./devops/tf apply -auto-approve

```

In [the next chapter](#) in [Part 2: GitHub Actions for Infrastructure](#), we'll finish off this workflow and update our `all.yaml` workflow as well to also use GitHub Actions.

Chapter 8

# Using Ansible to Configure Infrastructure

## Chapter 8

# Using Ansible to Configure Infrastructure

In this section, we'll learn how to use Ansible to configure our infrastructure to run our application.

Wait, didn't we *just* configure our infrastructure with Terraform? Yes and no. To reiterate what we discussed in the last chapter, Terraform turns things on, Ansible makes them work the correct way.

It's true, that Terraform *can* configure our infrastructure to a degree (using *provisioners*) but it's not nearly as powerful as Ansible is.

Ansible *verifies* the state of all of the Linode instances at runtime. Here are a few examples of what I mean:

- Is NGINX installed? Ansible says yes.
- Is Apache installed? Ansible says yes. Can we remove it? Ansible removes it.
- Is Docker installed? Ansible says no. Can we install it? Ansible installs it.
- Is our system up to date? Ansible says no. Can we update it? Ansible updates it.

This same concept applies across *all* of the machines that Ansible has access to (by way of `inventory.ini` and connection keys ( `ssh keys` , `root password` , etc).

Before we dive in, I will start with: Ansible does not *need* or *care* about Terraform in the least. Ansible is perfectly capable of running all the above (and much more) without aid.

So what does the process look like?

- Install Python 3.6 and up
- Install Ansible via Python's Package Installer (pip)
- Create an Inventory File ( `inventory.ini` )
- Create Playbooks
- Run Ansible on the playbooks

## 0. The Ansible Inventory file

Ansible needs to know about all of the IP addresses of the virtual machines we want to manage. These IP Addresses can come from anywhere:

- Linode
- Amazon Web Services
- Google Cloud
- Azure
- Raspberry Pi
- A Windows laptop that you made the wise decision to turn into a Linux server

The file looks like this:

ini

```
[my_linodes]
127.0.0.1
127.0.0.2
127.0.0.3
127.0.0.4
127.0.0.5
127.0.0.6
127.0.0.7

[my_pis]
127.0.0.8
127.0.0.9
127.0.0.10
127.0.0.11
```

The inventory file syntax, called **INI**, comes directly from the Microsoft Windows INI files as described in the [Python Config Parser](#).

A more realistic inventory file will look like this:

ini

```
[webapps]
65.228.51.122
192.53.185.185
65.228.51.129

[loadbalancer]
172.104.194.157

[redis]
198.58.106.22

[workers]
198.66.106.38
```

As you may recall, we create this `inventory.ini` file with Terraform in [the Templates section](#).

I should point out that you can also use *domain names* in place of IP Addresses. The reason we use IP Addresses here is for simplicity.

## Feel Like Skipping Terraform?

Aside from the inventory file (and keys added to those instances), you can skip Terraform to use Ansible.

Here's what you'll need to do:

1. Log in to Linode
2. Provision at least two instances: one for Django to run on (web apps), and one for NGINX load balancer to run on
3. Add your local SSH keys
4. Create a root user password (and remember it)



# 1. Ansible Basics

Have you ever used `ssh` to log in to a machine? I hope that answer is yes, but the idea is that you use Secure Shells ( `ssh` ) like this:

```
ssh root@some_ip
```

After you log in here. You can start using this virtual machine ( `remote host` ) to do whatever you'd like. Install things, run Django, do web scraping, and text your friends. Whatever. It's your virtual machine, have fun.

Using `ssh` to control a couple of machines is not a big deal. Using `ssh` to automate a couple of machines is a bit trickier but still doable. Using `ssh` to configure 100 machines is downright nuts.

Instead, we need a tool like Ansible (or SaltStack, Puppet Bolt, or Chef like I cover in my book Try Infrastructure as Code) to automate how everything works. Ansible, in a sense, will "ssh" into a remote machine (or group of remote machines) what we need it to do.

To use Ansible we need to do the following:

1. Install Ansible
2. Add an inventory file (done)
3. Create a playbook
4. Run Ansible ( `ansible-playbook main.yaml` )

Playbooks are `yaml` files that declare the *state* we want any given host (or host group) in. The `host` / `host group` can be declared inline with the Playbook or when we run Ansible or both!

A `host` is simply a virtual machine (or `remote host` ) that we have control of. In our case, it's a Linode instance. A `host group` is merely a collection of `hosts` . All of our `host` entries in `inventory.ini` are what we'll use Ansible to control.

Let's install Ansible.

## 2. Install Ansible

In [Getting Started](#), we setup a Python virtual environment. I recommend you stick with that ([venv reference](#)).

Navigate to your project

```
cd path/to/your/cfe-django-blog
```

Activate the virtual environment:

- macOS/Linux: `source venv/bin/activate`
- Windows: `.\venv\Scripts\activate`

Install Ansible

```
$(venv) python -m pip install ansible --upgrade
```

You should *not* add Ansible to your `requirements.txt` because Ansible will *not* be used while running your Django project. Ansible will only be used via GitHub Actions.

## 3. Create a Playbook

Let's just install `nginx` on all of our machines.

Create `devops/ansible/main.yaml` with

yaml

```
- hosts: all
  become: yes
  tasks:
    - name: Install Nginx
      apt:
        name: nginx
        state: present
        update_cache: yes
```

Now let's add our Ansible configuration file `devops/ansible/ansible.cfg`

```
[defaults]
ansible_python_interpreter='/usr/bin/python3'
deprecation_warnings=False
inventory=./inventory.ini
remote_user="root"
retries=2
```

The keys to this file are:

- `inventory=./inventory.ini` - this comes from the previous step
- `remote_user="root"` - the `remote_user` username `root` comes from the Linode Instance we provisioned (image: Ubuntu 20.04)

Let's verify we have:

- `devops/ansible/ansible.cfg`
- `devops/ansible/inventory.ini` (with at least 1 host)
- `devops/ansible/main.yaml`

Now run:

```
cd devops/ansible/
ansible-playbook main.yaml
```

This will install `NGINX` on all machines. Now let's get rid of it:

Update `devops/ansible/main.yaml` with:

`yaml`

```
--
- hosts: all
  become: yes
  tasks:
    - name: Install NGINX
      apt:
        name: nginx
        state: absent
```

Now run:

```
ansible-playbook main.yaml
```

That's Ansible in a nutshell. Now we just add more complex playbooks to ensure our **hosts** are configured *exactly* as we need them to be configured.

## 3. Ansible Variables

In `devops/ansible/vars/main.yaml` add the following:

yaml

```
---
docker_username: "codingforentrepreneurs"
docker_token: "your-dockerhub-token"
docker_appname: "cfe-django-blog"
```

All of these variables were created in [Chapter 3](#). They should also already be within your GitHub Action Secrets.

These variables can now be used throughout our Ansible project just like:

yaml

```
---
- hosts: all
  become: yes
  vars_files:
    - vars/main.yaml
  tasks:
    - name: Install NGINX
      debug: "msg='Hello there from {{ docker_username }}'"
```

Let's update our `.gitignore` to ensure this file is not added to Git (we'll auto-generate it in GitHub Actions):

```
cd path/to/my/project/root/  
echo 'devops/ansible/vars/main.yaml' >> .gitignore
```

## 4. Ansible Configuration File

Now let's add our configuration file. This file must be named `ansible.cfg` and should be at the root of your Ansible files.

`devops/ansible/ansible.cfg`

```
[defaults]  
ansible_python_interpreter='/usr/bin/python3'  
deprecation_warnings=False  
inventory=./inventory.ini  
remote_user="root"  
host_key_checking=False  
private_key_file = ../../keys/tf-github  
retries=2
```

Using this file allows us to keep our `ansible-playbook` commands very minimal by providing four key values:

- `inventory`
- `remote_user`
- `host_key_checking`
- `private_key_file`

**inventory** this is a reference to the file that lists all the instances of Linode Virtual Machines we provisioned on Linode using Terraform in the previous chapter (or manually if you did it that way).

**remote\_user** this is the default user we wish to connect to these instances

**host\_key\_checking** if this is set to `True` your *ssh* connection via *Ansible* may fail.

**private\_key\_file** When we provisioned our instances in the Terraform chapter, we added a public key. This is a reference to the private key's path relative to this file itself. Update this as needed.

Add this file to `.gitignore` as we will recreate it with GitHub Actions as well.

```
echo `devops/ansible/ansible.cfg` >> .gitignore
```

At this point:

## 5. Ansible Docker Role

The first thing we need our instances to have is Docker installed. Using an Ansible Role makes this a pretty straightforward process.

First, we'll create default tasks for this role. Here's what you need to do:

1. Create a role directory in our case `docker-install`
2. Create a task directory within our role directory ie `docker-install/tasks`
3. Add `main.yaml` to that task directory: `docker-install/tasks/main.yaml`. The *only* name that we can customize in this 3 step process is `docker-install`.

First, let's create our directories:

```
mkdir -p devops/ansible/roles/  
mkdir -p devops/ansible/roles/docker-install  
mkdir -p devops/ansible/roles/docker-install/tasks  
mkdir -p devops/ansible/roles/docker-install/handlers
```

Now let's create an Ansible role that will call `docker-install` so that we use Docker whenever we need it:

In `devops/ansible/roles/docker-install/tasks/main.yaml`

yaml

```
- name: Update Apt Cache  
  apt:  
    update_cache: yes  
  
- name: Install System Requirements  
  apt:  
    name: "{{ item }}"  
    state: latest  
  with_items:  
    - curl
```

```
- git
- build-essential
- python3-dev
- python3-pip
- python3-venv

- name: Grab Docker Install Script
  get_url:
    url: https://get.docker.com
    dest: /tmp/get-docker.sh
    mode: 0755
  notify: exec docker script

- name: Verify Docker Command
  shell: command -v docker >/dev/null 2>&1
  ignore_errors: true
  register: docker_exists

- debug: msg="{{ docker_exists.rc }}"

- name: Trigger docker install script if docker not running
  shell: echo "Docker command"
  when: docker_exists.rc != 0
  notify: exec docker script

- name: Verify Docker Compose Command
  shell: command -v docker-compose >/dev/null 2>&1
  ignore_errors: true
  register: docker_compose_exists

- debug: msg="{{ docker_compose_exists.rc }}"

- name: Install docker-compose for Python 3 using pip3
  shell: echo "Install Docker Compose"
  when: docker_compose_exists.rc != 0
  notify: install docker compose
```

In `devops/ansible/roles/docker-install/handlers/main.yaml`

yaml

```
- name: exec docker script
  shell: /tmp/get-docker.sh
  notify: install docker compose

- name: install docker compose
  pip:
    name:
      - docker-compose
    executable: pip3
```

In this case, we used the concept of Ansible Roles to:

- Define standard tasks that the role will take in `tasks/main.yaml`
- Define standard handlers (or notification handlers) that the role will make available in `handlers/main.yaml`

Tasks allow us to just *run* whenever the role is used (more on this later). In other words `tasks/main.yaml` will run in written order whenever this role is added. Handlers, on the other hand, run only when the handler itself is notified by name.

Using Ansible Roles greatly improves the reusability and readability of our Ansible playbooks.

Before we can use these roles, let's uncover a few important setup items we need.

## Local Testing

Yes, I do recommend you test this code locally. To do so, you'll need the following:

- A virtual environment (Chapter 2)
- A built Docker Image Container & Docker Hub Account (Chapter 3)
- `inventory.init` An Ansible inventory file full of IP addresses pointing to the pre-existing infrastructure created by Terraform (Chapter 8)
- `ansible.cfg` (added above)
- `.env.prod` in the root of your project.

If that's true, we're going to install Ansible locally to give this role and playbook a try.



## Install Ansible

```
$(venv) python -m pip install ansible
```

### Copy `.env` to `.env.prod`

When testing locally, we will need to add production environment variables as well. We use a separate env file to do so.

```
cp .env .env.prod
```

After you're done with local testing, it's a good idea to remove `.env.prod` and keep all of those variables stored on GitHub Action Secrets.

### Create `devops/ansible/main.yaml`

yaml

```
---
- hosts: webapps
  become: yes
  roles:
    - roles/docker-install
```

Notice that we have referenced `roles/docker-install`. This means that our recently created role will be added to this `main.yaml` playbook. In other words:

- Everything in `roles/docker-install/tasks/main.yaml` will run by default, and in order of role declaration.
- Everything in `roles/docker-install/handlers/main.yaml` will *only run* if a task uses `notify:` for any given handler. Handlers always run in the order they are defined regardless of when they are called.

## Navigate to Ansible folder

```
cd devops/ansible/
```

## Run playbook

```
ansible-playbook main.yaml
```

## Understanding our `docker-install` role:

The role `devops/ansible/roles/docker-install` exists so that we have:

- [Docker Community Edition](#) installed
- [Docker Compose](#) installed via Python Pip (ideal supported version for linux)

To do this we have several tasks and handlers that need to be run.

Let's go over each task first:

yaml

```
- name: Update Apt Cache
  apt:
    update_cache: yes
```

All this does is run `apt-get update` on each machine. (And yes, it's `sudo` when you have `become: yes` on your playbook)

yaml

```
- name: Install System Requirements
  apt:
    name: "{{ item }}"
    state: latest
  with_items:
    - curl
    - git
    - build-essential
    - python3-dev
    - python3-pip
    - python3-venv
```

This block installs several system-wide dependencies we need for Docker & Docker compose to work.

How?

- **name** : We can write a short descriptive name here
- **apt** allows us to use the **apt get install** command
- **name: "{{ item }}"** works in tandem with the list of packages in the **with\_items** block
- **state: latest** means we will ensure these packages (the ones listed in **with\_items** ) are installed and updated as needed

yaml

```
- name: Grab Docker Install Script
  get_url:
    url: https://get.docker.com
    dest: /tmp/get-docker.sh
    mode: 0755
    notify: exec docker script
```

On <https://get.docker.com> there's a super convenient script to install the latest Docker Community edition. This block will download that script for us:

- **get\_url** will automatically open a URL for us and download the contents
- **url** this, of course, is the URL we want to open
- **dest** is the location on our remote (our Linode instance) we want to store this file. When in doubt, add it to **tmp/** or **opt/**
- **mode: 0755** gives our user permission to simply execute this file
- **notify: ...** , This will notify a handler of our choosing (more on this later) but *only* if this file changes.

yaml

```
- name: Verify Docker Command
  shell: command -v docker >/dev/null 2>&1
  ignore_errors: true
  register: docker_exists

- debug: msg="{{ docker_exists.rc }}"
```

This block installs several system-wide dependencies we need for Docker & Docker compose to work.

How?

- **name** : We can write a short descriptive name here
- **apt** allows us to use the **apt get install** command
- **name: "{{ item }}"** works in tandem with the list of packages in the **with\_items** block
- **state: latest** means we will ensure these packages (the ones listed in **with\_items** ) are installed and updated as needed

yaml

```
- name: Grab Docker Install Script
  get_url:
    url: https://get.docker.com
    dest: /tmp/get-docker.sh
    mode: 0755
  notify: exec docker script
```

The command **command -v docker >/dev/null 2>&1** will check if the command **docker** is available on our remote host. If it does not exist, **docker\_exists** will be a variable we can use elsewhere in Ansible.

- **shell** allows us to run arbitrary shell commands
- **ignore\_errors: true** is great for skipping errors that happen in any given block.
- **register: docker\_exists** is a variable we can reuse in other tasks in Ansible.
- **debug: msg="{{ docker\_exists.rc }}"** is proving the re-usability of **docker\_exists**
- **docker\_exists.rc** will yield a **0** if the command exists.

yaml

```
- name: Trigger docker install script if docker not running
  shell: echo "Docker command"
  when: docker_exists.rc != 0
  notify: exec docker script
```

This block exists solely to trigger the **exec docker script** handler (as we'll see soon).

yaml

```
- name: Verify Docker Compose Command
  shell: command -v docker-compose >/dev/null 2>&1
  ignore_errors: true
  register: docker_compose_exists

- debug: msg="{{ docker_compose_exists.rc }}"

- name: Install docker-compose for Python 3 using pip3
  shell: echo "Install Docker Compose"
  when: docker_compose_exists.rc != 0
  notify: install docker compose
```

This block re-iterates what we did with `docker` but now with `docker-compose`. Keep in mind that the version of Docker Compose we're using is via Python Pip so the command is `docker-compose` and not `docker compose` as you may or may not be used to.

## `docker-install` handlers:

First and foremost, the ordering of the handlers matters. If you order them differently, they will execute differently and potentially cause major issues for your automation pipeline.

In, `devops/ansible/roles/docker-install/handlers/main.yaml` we have:

yaml

```
- name: exec docker script
  shell: /tmp/get-docker.sh
  notify: install docker compose

- name: install docker compose
  pip:
    name:
      - docker-compose
    executable: pip3
```

As you may notice, these resource blocks look *identical* to the `tasks/main.yaml` resource blocks. That's because they are! The only difference here is the name we give these handlers, such as `exec docker script` and `install docker compose`, need to be *unique globally*. Further, to trigger these handlers, we must use the name *exactly* as it's written and within a `notify:`. Remember: *Each handler should have a globally unique name.* (lifted directly from the Ansible Handler [docs](#))

- `notify: install docker compose` will notify the handler `install docker compose`
- `notify: another unknown one` will notify the handler `another unknown one` and definitely won't notify `install docker compose`

Handlers are great for repeatable events you need to be able to trigger at some point in time.

## 6. Ansible Templates

Now we're going to leverage Ansible Templates to create the following:

- NGINX Load Balancer configuration
- Docker Compose production `.yaml` file

These items will be filled out based on 4 things:

- Ansible Playbooks
- Ansible Roles
- `inventory.ini`
- `vars/main.yaml`

First off, let's create our `docker-compose.prod.yaml` file:

`templates/docker-compose.yaml.jinja2`

yaml

```
version: "3.9"
services:
  watchtower:
    image: index.docker.io/containrrr/watchtower:latest
    restart: always
    volumes:
      - /var/run/docker.sock:/var/run/docker.sock
      - /root/.docker/config.json:/config.json
    command: --interval 30
  profiles:
    - app
```

```
app:
  image: index.docker.io/{{ docker_username }}/{{ docker_appname }}:latest
  restart: always
  env_file: ./env
  container_name: {{ docker_appname }}
  environment:
    - PORT=8080
  ports:
    - "80:8080"
  expose:
    - 80
  volumes:
    - ./certs:/app/certs
  profiles:
    - app
redis:
  image: redis
  restart: always
  ports:
    - "6379:6379"
  expose:
    - 6379
  volumes:
    - redis_data:/data
  entrypoint: redis-server --appendonly yes
  profiles:
    - redis

volumes:
  redis_data:
```

Next up, our load balancer NGINX configuration file:

`templates/nginx-lb.conf.jinja`

conf

```
{% if groups['webapps'] %}
upstream myproxy {
    {% for host in groups['webapps'] %}
        server {{ host }};
    {% endfor %}
}
{% endif %}

server {
    listen 80;
    server_name localhost;
    root /var/www/html;

    {% if groups['webapps'] %}
    location / {
        proxy_pass http://myproxy;
        proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
        proxy_set_header Host $host;
        proxy_redirect off;
    }
    {% endif %}
}
```



## 7. Ansible & Docker-based Django App on Docker Hub

Now let's create our Django application role. This role is going to do several things:

1. Copy environment variables
2. Copy our database certificate
3. Add our Docker Compose File
4. Build/Run Docker Compose (as needed)

Notice that I *never* copy my Django code here; this is done on purpose. It's also important to understand that Ansible *can* configure our servers to run our Django apps (or nearly any app for that matter) without the need for Docker.

In some cases, it is perfectly reasonable to skip using Docker and just opt for Ansible to configure your environment, this is done a lot in the IT world already.

In most cases, however, I believe that leveraging Docker is far more advantageous. Here are a few reasons in addition to what we discussed in the [Docker Chapter](#):

1. If Docker is running, your container can probably run too
2. If you need to upgrade to Kubernetes, your entire applications are ready to be ported right now
3. GitHub is the *only source of truth* for our code. (Replace GitLab and/or any other Git repo hosting services)
4. Once built, containers can run *right now*
5. If we change the tech stack, the Docker container and possibly Docker compose file are the *only* configuration changes we'll need

With all that, let's create the `django-app` role.

### Create The `django-app` Role

```
mkdir -p develops/ansible/django-app
mkdir -p develops/ansible/django-app/handlers
mkdir -p develops/ansible/django-app/tasks
```

## Add `django-app` Role Tasks

In `devops/ansible/django-app/tasks/main.yaml`

yaml

```
- name: Waiting to connect to remote
  wait_for_connection:
    sleep: 10
    timeout: 600

- name: Ensure destination config dir exists
  file:
    path: /var/www/app/certs/
    state: directory

- name: Copy .env file
  ansible.builtin.copy:
    src: "{{ playbook_dir | dirname | dirname }}/.env.prod"
    dest: /var/www/app/.env

- name: Host to Env File
  shell: "echo \"\n\nWEBAPP_NODE_HOST={{ inventory_hostname }}\" >> /var/www/app/.env"

- name: Load Balancer to Env File
  shell: "echo \"\n\nLOAD_BALANCER_HOST={{ groups['loadbalancer'][0] }}\" >> /var/www/app/.env"
  when: "groups['loadbalancer']|length == 1"

- name: Copy db cert file
  ansible.builtin.copy:
    src: "{{ playbook_dir | dirname | dirname }}/certs/db.crt"
    dest: /var/www/app/certs/db.crt

- name: Add Docker Compose
  template:
    src: ./templates/docker-compose.yaml.jinja2
    dest: /var/www/app/docker-compose.prod.yaml
```

## Update Django `settings.py`

In the last step, we have two Ansible-related environment variables added:

- `WEBAPP_NODE_HOST={{ inventory_hostname }}`
- `LOAD_BALANCER_HOST={{ groups['loadbalancer'][0] }}`

These two settings are for our Django application. For Django to run correctly in production, we must update `ALLOWED_HOSTS` in `settings.py`.

In `cfeblog/settings.py` add the following:

python

```
ALLOWED_HOSTS = []

# Domain name or other GitHub Action Host Value
ALLOWED_HOST = os.environ.get("ALLOWED_HOST")
if ALLOWED_HOST:
    ALLOWED_HOSTS = [ALLOWED_HOST]

# Individual Web App Linode Host IP
WEBAPP_NODE_HOST = os.environ.get("WEBAPP_NODE_HOST")
if WEBAPP_NODE_HOST:
    ALLOWED_HOSTS = [WEBAPP_NODE_HOST]

# Nginx Load Balancer Linode Host IP
LOAD_BALANCER_HOST = os.environ.get("LOAD_BALANCER_HOST")
if LOAD_BALANCER_HOST:
    ALLOWED_HOSTS = [LOAD_BALANCER_HOST]
```

## Add `django-app` Role Handler

In `devops/ansible/django-app/handlers/main.yaml`

yaml

```
- name: docker-compose start django app
  shell: docker-compose -f /var/www/app/docker-compose.prod.yaml --profile app up -d
  timeout: 300

- name: docker-compose force rebuild django app
  shell: |
    cd /var/www/app/
    docker-compose -f docker-compose.prod.yaml --profile app stop
    docker-compose rm -f
    docker-compose -f /var/www/app/docker-compose.prod.yaml pull
    docker-compose -f docker-compose.prod.yaml --profile app up -d

- name: docker-compose start redis
  shell: docker-compose -f /var/www/app/docker-compose.prod.yaml --profile redis up -d

- name: docker-compose force rebuild redis
  shell: |
    cd /var/www/app/
    docker-compose -f docker-compose.prod.yaml --profile redis stop --remove-orphans
    docker-compose rm -f
    docker-compose -f /var/www/app/docker-compose.prod.yaml --profile app pull
    docker-compose -f docker-compose.prod.yaml --profile redis up -d
```

The commands above are standard `docker-compose` commands. We use `--profile app` or `--profile redis` to target specific profiles from our `docker-compose.prod.yaml` template that we created above. If we needed to add additional services, we would need to update this template.

## Update `main.yaml`

In `devops/ansible/main.yaml` :

yaml

```
---
- hosts: webapps
  become: yes
  roles:
    - docker-install
    - django-app
  vars_files:
    - vars/main.yaml
  tasks:
    - name: Login to Docker via vars/main.yaml
      shell: "echo \"{{ docker_token }}\" | docker login -u {{ docker_username }}
        --password-stdin"
    - name: Run our Django app in the Background
      shell: echo "Running Docker Compose for Django App"
      notify: docker-compose start django app
```

Let's break this all down:

- First off, remember all handlers will run after all tasks are complete. So that means that `docker-compose start django app` will only run after we log in to Docker.
- Here we use `hosts:webapps` to target the group of hosts ( `[webapps]` ) directly from `inventory.ini` .
- We use `become: yes` so we have root privilege to execute commands
- We need both `docker-install` and `django-app` roles for this playbook to run
- The `tasks` here merely exist to *log in to Docker hub* and to *run the `docker-compose` handler* from the `django-app` role.

## Playbook for Force Docker Container Rebuild

1. Create `playbooks` folder

```
mkdir -p devops/ansible/playbooks
```

2. Create the `force-rebuild.yaml` playbook:

In `devops/ansible/playbooks/force-rebuild-django-app.yaml` :

yaml

```
---
- hosts: webapps
  become: yes
  roles:
    - docker-install
    - docker-app
  vars_files:
    - vars/main.yaml
  tasks:
    - name: Rebuild Django App on Web App Hosts
      shell: echo "Forcing Django App Rebuild"
      notify: docker-compose force rebuild django app
```

Once again, keep in mind that we're using just the `webapps` group of hosts from the `inventory.ini` .

If you ever need to use this playbook just run:

```
cd devops/ansible/
ansible-playbook playbooks/force-rebuild-django-app.yaml
```

## The Magic of Watchtower

In our `docker-compose.yaml.jinja` template, we have a service called `watchtower` that references the image `containrrr/watchtower`. This image exists *precisely* to update any/all of our Docker containers when they are updated. Let's take a look at the declaration in `docker-compose.yaml.jinja`:

yaml

```
watchtower:
  image: index.docker.io/containrrr/watchtower:latest
  restart: always
  volumes:
    - /var/run/docker.sock:/var/run/docker.sock
    - /root/.docker/config.json:/config.json
  command: --interval 30
```

- `watchtower`: is the name of the service (you can name it anything)
- `image: index.docker.io/containrrr/watchtower:latest` is the latest image for it
- `restart: always` will start this container running if the Linode reboots
- The volume `/var/run/docker.sock:/var/run/docker.sock` is so the `watchtower` service is aware of *all* running containers on this instance (in our case it's our `app` service)
- The volume `/root/.docker/config.json:/config.json` is so that `watchtower` can pull private images from our Docker registry and, in our case, from Docker hub
- `command: --interval 30` means that watchtower will check every 30 seconds for a new version of our container.

So what does this mean?

Well, every time we build and push our Docker container for our Django app, this `watchtower` service will *automatically* update our Linode instance's running container. It does this *pretty* gracefully and honestly. Much better than running a force rebuild, as we did in the last section ( `ansible-playbook playbooks/force-rebuild-django-app.yaml` ).

In my opinion, `containrrr/watchtower` is a means to an end and that end is leveraging Kubernetes. We'll leave that for another time but know that Kubernetes has much better graceful updates (and rollbacks) of container deployments than what I proposed here. Leveraging `docker-compose` instead of Kubernetes has the added benefit of simplicity without the overhead and learning curve that comes with Kubernetes.

Let's add in our load balancer.

## 8. Ansible Load Balancer Role

### Create The `nginx-lb` Role

```
mkdir -p devops/ansible/roles/nginx-lb
mkdir -p devops/ansible/roles/nginx-lb/tasks
mkdir -p devops/ansible/roles/nginx-lb/handlers
```

### Add The `nginx-lb` Tasks

In `devops/ansible/roles/nginx-lb/tasks/main.yaml` :

yaml

```
- name: Waiting to connect
  wait_for_connection:
    sleep: 10
    timeout: 600

- name: Install NGINX
  apt:
    name: nginx
    state: present
    update_cache: yes

- name: Ensure nginx is started and enabled to start at boot.
  service:
    name: nginx
    state: started
    enabled: yes

- name: Add NGINX Config
  template:
    src: ./templates/nginx-lb.conf.jinja
    dest: /etc/nginx/sites-available/default
    notify: reload nginx

- name: Enable New NGINX Config
  file:
    src: /etc/nginx/sites-available/default
    dest: /etc/nginx/sites-enabled/default
    state: link
```



All this does is install `nginx`, the `nginx` service is started, and add a new configuration that combines the recently created `./templates/nginx-lb.conf.jinja` with our `inventory.ini`.

So, why are we not running `nginx` through Docker? After everything I laid out before, this is a valid question and a worthy one at that. The answer: `nginx` is incredibly efficient and requires very little setup to run incredibly well. The only change that will likely occur in this app is if we add additional web app nodes, which would just require us to run the `nginx-lb` role once again.

## Add The `nginx-lb` Handlers

In `devops/ansible/roles/nginx-lb/handlers/main.yaml`:

yaml

```
- name: reload nginx
  service:
    name: nginx
    state: reloaded

- name: restart nginx
  service:
    name: nginx
    state: restarted
```

## Update `main.yaml`

In `devops/ansible/main.yaml` :

`yaml`

```
---
- hosts: webapps
  become: yes
  roles:
    - docker-install
    - django-app
  vars_files:
    - vars/main.yaml
  tasks:
    - name: Login to Docker via vars/main.yaml
      shell: "echo \"{{ docker_token }}\" | docker login -u {{ docker_username }}
        --password-stdin"
    - name: Run our Django app in the Background
      shell: echo "Running Docker Compose for Django App"
      notify: docker-compose start django app

- hosts: loadbalancer
  become: yes
  roles:
    - nginx-lb
```

If you're following locally, run;

```
cd devops/ansible
ansible-playbook main.yaml
```

## 9. Part 2: GitHub Actions for Infrastructure

Let's update our `.github/workflows/infra.yaml` file to:

yaml

```
name: 4 - Apply Infrastructure via Terraform and Ansible
```

```
on:
```

```
  workflow_call:
```

```
    secrets:
```

```
      ALLOWED_HOST:
```

```
        required: false
```

```
      DJANGO_SECRET_KEY:
```

```
        required: true
```

```
      DJANGO_VM_COUNT:
```

```
        required: true
```

```
      DOCKERHUB_APP_NAME:
```

```
        required: true
```

```
      DOCKERHUB_TOKEN:
```

```
        required: true
```

```
      DOCKERHUB_USERNAME:
```

```
        required: true
```

```
      LINODE_BUCKET_REGION:
```

```
        required: true
```

```
      LINODE_BUCKET_ACCESS_KEY:
```

```
        required: true
```

```
      LINODE_BUCKET_SECRET_KEY:
```

```
        required: true
```

```
      LINODE_IMAGE:
```

```
        required: true
```

```
      LINODE_OBJECT_STORAGE_DEVOPS_BUCKET:
```

```
        required: true
```

```
      LINODE_OBJECT_STORAGE_DEVOPS_TF_KEY:
```

```
        required: true
```

```
      LINODE_OBJECT_STORAGE_DEVOPS_ACCESS_KEY:
```

```
        required: true
```

```
      LINODE_OBJECT_STORAGE_DEVOPS_SECRET_KEY:
```

```
        required: true
```

```
      LINODE_BUCKET:
```

```
        required: true
```

```
      LINODE_PA_TOKEN:
```

```
        required: true
```

```
      MYSQL_DB_CERT:
```

```
        required: true
```

```

MYSQL_DATABASE:
  required: true
MYSQL_HOST:
  required: true
MYSQL_ROOT_PASSWORD:
  required: true
MYSQL_TCP_PORT:
  required: true
MYSQL_USER:
  required: true
ROOT_USER_PW:
  required: true
SSH_PUB_KEY:
  required: true
SSH_DEVOPS_KEY_PUBLIC:
  required: true
SSH_DEVOPS_KEY_PRIVATE:
  required: true
workflow_dispatch:

jobs:
  terraform_terraform:
    runs-on: ubuntu-latest
    steps:
      - name: Checkout
        uses: actions/checkout@v2
      - name: Setup Terraform
        uses: hashicorp/setup-terraform@v1
        with:
          terraform_version: 1.1.9
      - name: Add Terraform Backend for S3
        run: |
          cat << EOF > devops/tf/backend
          skip_credentials_validation = true
          skip_region_validation = true
          bucket="{{ secrets.LINODE_OBJECT_STORAGE_DEVOPS_BUCKET }}"
          key="{{ secrets.LINODE_OBJECT_STORAGE_DEVOPS_TF_KEY }}"
          region="us-southeast-1"
          endpoint="us-southeast-1.linodeobjects.com"
          access_key="{{ secrets.LINODE_OBJECT_STORAGE_DEVOPS_ACCESS_KEY }}"
          secret_key="{{ secrets.LINODE_OBJECT_STORAGE_DEVOPS_SECRET_KEY }}"
          EOF

```

```

- name: Add Terraform TFVars
  run: |
    cat << EOF > devops/tf/terraform.tfvars
    linode_pa_token="{{ secrets.LINODE_PA_TOKEN }}"
    authorized_key="{{ secrets.SSH_DEVOPS_KEY_PUBLIC }}"
    root_user_pw="{{ secrets.ROOT_USER_PW }}"
    app_instance_vm_count="{{ secrets.DJANGO_VM_COUNT }}"
    linode_image="{{ secrets.LINODE_IMAGE }}"
    EOF
- name: Terraform Init
  run: terraform -chdir=./devops/tf init -backend-config=backend
- name: Terraform Validate
  run: terraform -chdir=./devops/tf validate -no-color
- name: Terraform Apply Changes
  run: terraform -chdir=./devops/tf apply -auto-approve
- name: Add MySQL Cert
  run: |
    mkdir -p certs
    cat << EOF > certs/db.crt
    {{ secrets.MYSQL_DB_CERT }}
    EOF
- name: Add SSH Keys
  run: |
    cat << EOF > devops/ansible/devops-key
    {{ secrets.SSH_DEVOPS_KEY_PRIVATE }}
    EOF
- name: Update devops private key permissions
  run: |
    chmod 400 devops/ansible/devops-key
- name: Install Ansible
  run: |
    pip install ansible
- name: Add Production Environment Variables to Instance
  run: |
    cat << EOF > .env.prod
    ALLOWED_HOST={{ secrets.ALLOWED_HOST }}
    # required keys
    DJANGO_SECRET_KEY={{ secrets.DJANGO_SECRET_KEY }}
    DATABASE_BACKEND=mysql
    DJANGO_DEBUG=""
    DJANGO_STORAGE_SERVICE=linode
    # mysql db setup
    MYSQL_DATABASE={{ secrets.MYSQL_DATABASE }}
    MYSQL_USER={{ secrets.MYSQL_USER }}

```

```

MYSQL_PASSWORD=${{ secrets.MYSQL_PASSWORD }}
MYSQL_ROOT_PASSWORD=${{ secrets.MYSQL_DB_ROOT_PASSWORD }}
MYSQL_TCP_PORT=${{ secrets.MYSQL_DB_PORT }}
MYSQL_HOST=${{ secrets.MYSQL_DB_HOST }}
# static files connection
LINODE_BUCKET=${{ secrets.LINODE_BUCKET }}
LINODE_BUCKET_REGION=${{ secrets.LINODE_BUCKET_REGION }}
LINODE_BUCKET_ACCESS_KEY=${{ secrets.LINODE_BUCKET_ACCESS_KEY }}
LINODE_BUCKET_SECRET_KEY=${{ secrets.LINODE_BUCKET_SECRET_KEY }}
EOF
- name: Adding or Override Ansible Config File
  run: |
    cat << EOF > devops/ansible/ansible.cfg
    [defaults]
    ansible_python_interpreter='/usr/bin/python3'
    deprecation_warnings=False
    inventory=./inventory.ini
    remote_user="root"
    host_key_checking=False
    private_key_file = ./devops-key
    retries=2
    EOF
- name: Adding Ansible Variables
  run: |
    mkdir -p devops/ansible/vars/
    cat << EOF > devops/ansible/vars/main.yaml
    ---
    docker_appname: "${{ secrets.DOCKERHUB_APP_NAME }}"
    docker_token: "${{ secrets.DOCKERHUB_TOKEN }}"
    docker_username: "${{ secrets.DOCKERHUB_USERNAME }}"
    EOF
- name: Run main playbook
  run: |
    ANSIBLE_CONFIG=devops/ansible/ansible.cfg ansible-playbook devops/ansible/
    main.yaml

```

At this point, I encourage you to go through this workflow line-by-line to see if you understand exactly what's going on. Here's a hint - it runs everything that we did in the last two chapters.

The most notable exception is this line:

```
ANSIBLE_CONFIG=devops/ansible/ansible.cfg ansible-playbook devops/ansible/main.yaml
```

Using `ANSIBLE_CONFIG` along with a path to our `ansible.cfg` file allows us to run this command *anywhere* on our machine just as long as our paths are correct (to the config file as well as the initial playbook).

## Part 2: Update `.github/workflows/all.yaml` to support Ansible workflow.

yaml

```
name: 0 - Run Everything

on:
  workflow_dispatch:

jobs:
  test_django:
    uses: ../github/workflows/test-django-mysql.yaml
  build_container:
    needs: test_django
    uses: ../github/workflows/container.yaml
    secrets:
      DOCKERHUB_APP_NAME: ${ secrets.DOCKERHUB_APP_NAME }
      DOCKERHUB_USERNAME: ${ secrets.DOCKERHUB_USERNAME }
      DOCKERHUB_TOKEN: ${ secrets.DOCKERHUB_TOKEN }
  update_infra:
    needs: build_container
    uses: ../github/workflows/infra.yaml
    secrets:
      ALLOWED_HOST: ${ secrets.ALLOWED_HOST }
      DJANGO_SECRET_KEY: ${ secrets.DJANGO_SECRET_KEY }
      DJANGO_VM_COUNT: ${ secrets.DJANGO_VM_COUNT }
      DOCKERHUB_APP_NAME: ${ secrets.DOCKERHUB_APP_NAME }
      DOCKERHUB_TOKEN: ${ secrets.DOCKERHUB_TOKEN }
      DOCKERHUB_USERNAME: ${ secrets.DOCKERHUB_USERNAME }
      LINODE_BUCKET_REGION: ${ secrets.LINODE_BUCKET_REGION }
```

```

LINODE_BUCKET_ACCESS_KEY: ${{ secrets.LINODE_BUCKET_ACCESS_KEY }}
LINODE_BUCKET_SECRET_KEY: ${{ secrets.LINODE_BUCKET_SECRET_KEY }}
LINODE_IMAGE: ${{ secrets.LINODE_IMAGE }}
LINODE_OBJECT_STORAGE_DEVOPS_BUCKET: ${{ secrets.LINODE_OBJECT_STORAGE_DEVOPS_
    BUCKET }}
LINODE_OBJECT_STORAGE_DEVOPS_TF_KEY: ${{ secrets.LINODE_OBJECT_STORAGE_DEVOPS_TF_
    KEY }}
LINODE_OBJECT_STORAGE_DEVOPS_ACCESS_KEY: ${{ secrets.LINODE_OBJECT_STORAGE_
    DEVOPS_ACCESS_KEY }}
LINODE_OBJECT_STORAGE_DEVOPS_SECRET_KEY: ${{ secrets.LINODE_OBJECT_STORAGE_
    DEVOPS_SECRET_KEY }}
LINODE_BUCKET: ${{ secrets.LINODE_BUCKET }}
LINODE_PA_TOKEN: ${{ secrets.LINODE_PA_TOKEN }}
MYSQL_DB_CERT: ${{ secrets.MYSQL_DB_CERT }}
MYSQL_DATABASE: ${{ secrets.MYSQL_DATABASE }}
MYSQL_HOST: ${{ secrets.MYSQL_HOST }}
MYSQL_ROOT_PASSWORD: ${{ secrets.MYSQL_ROOT_PASSWORD }}
MYSQL_TCP_PORT: ${{ secrets.MYSQL_TCP_PORT }}
MYSQL_USER: ${{ secrets.MYSQL_USER }}
ROOT_USER_PW: ${{ secrets.ROOT_USER_PW }}
SSH_PUB_KEY: ${{ secrets.SSH_PUB_KEY }}
SSH_DEVOPS_KEY_PUBLIC: ${{ secrets.SSH_DEVOPS_KEY_PUBLIC }}
SSH_DEVOPS_KEY_PRIVATE: ${{ secrets.SSH_DEVOPS_KEY_PRIVATE }}

```

#### collectstatic:

```

needs: test_django
uses: ../github/workflows/staticfiles.yaml
secrets:
  LINODE_BUCKET: ${{ secrets.LINODE_BUCKET }}
  LINODE_BUCKET_REGION: ${{ secrets.LINODE_BUCKET_REGION }}
  LINODE_BUCKET_ACCESS_KEY: ${{ secrets.LINODE_BUCKET_ACCESS_KEY }}
  LINODE_BUCKET_SECRET_KEY: ${{ secrets.LINODE_BUCKET_SECRET_KEY }}

```



## 10. Future Considerations

### Command Shortcuts with Make & a Makefile

Do you remember all the commands to run Terraform and Ansible? I rarely do, which is why I almost always create a Makefile (like we do in [Appendix F](#)).

### Environment Variable Updates

There's a good chance that your environment variables will change as your project grows.

- GitHub Action Secrets
- `.github/workflows/infra.yaml`
- `.github/workflows/all.yaml`
- Local `.env` and `.env.prod`

### Making Model Changes

If your Django/Python code changes, you must run `python manage.py makemigrations` -- I suggest doing this during development, and in a development environment. If you run `makemigrations` your `entrypoint.sh` will automatically run the migrations needed.

This can cause issues in production, which is why it's critical to ensure your development and production environments are as similar as possible.

### Custom Domains with Linode, Certbot, Let's Encrypt

In the [last chapter](#), we showed you an optional way to add a custom domain to your load balancer instance. Once you have this custom domain mapped, you can use [Certbot](#) (Let's Encrypt) to add a free HTTPS setup.

This process may require manual setup initially but it's well worth it. This is a great guide from Linode on how to implement Let's Encrypt on your load balancer: <https://www.linode.com/docs/guides/install-lets-encrypt-to-create-ssl-certificates/>

Keep in mind that using NGINX without Docker makes setting up Certbot/Let's Encrypt much easier.

Once you set up Certbot, your NGINX configuration will look something like:

Location: `/etc/nginx/sites-enabled/default`

Contents:

**conf**

```
upstream myproxy {
    server 123.23.123.23;
    server 123.23.123.23;
    server 123.23.123.23;
}

server {
    server_name tryiac.com www.tryiac.com *.tryiac.com;
    root /var/www/html;

    location / {
        proxy_pass http://myproxy;
        proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
        proxy_set_header Host $host;
        proxy_redirect off;
    }

    listen 443 ssl; # managed by Certbot
    ssl_certificate /etc/letsencrypt/live/tryiac.com/fullchain.pem; # managed by
    Certbot
    ssl_certificate_key /etc/letsencrypt/live/tryiac.com/privkey.pem; # managed by
    Certbot
    include /etc/letsencrypt/options-ssl-nginx.conf; # managed by Certbot
    ssl_dhparam /etc/letsencrypt/ssl-dhparams.pem; # managed by Certbot
}

server {
    if ($host = www.tryiac.com) {
        return 301 https://$host$request_uri;
    } # managed by Certbot
    if ($host = tryiac.com) {
        return 301 https://$host$request_uri;
    } # managed by Certbot

    listen 80;
    server_name tryiac.com www.tryiac.com *.tryiac.com;
    return 404; # managed by Certbot
}
```

Naturally, you would want to replace all instances of `tryiac.com` with whatever domain you're using.

Now, let's just update your `templates/nginx-lb.conf.jinja` with the certbot's (Let's Encrypt) additions:

**conf**

```
{% if groups['webapps'] %}
upstream myproxy {
    {% for host in groups['webapps'] %}
        server {{ host }};
    {% endfor %}
}
{% endif %}

server {
    server_name tryiac.com www.tryiac.com *.tryiac.com;
    root /var/www/html;

    {% if groups['webapps'] %}
    location / {
        proxy_pass http://myproxy;
        proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
        proxy_set_header Host $host;
        proxy_redirect off;
    }
    {% else %}
    index index.html;
    location / {
        try_files $uri $uri/ =404;
    }
    {% endif %}

    listen 443 ssl; # managed by Certbot
    ssl_certificate /etc/letsencrypt/live/tryiac.com/fullchain.pem; # managed by
        Certbot
    ssl_certificate_key /etc/letsencrypt/live/tryiac.com/privkey.pem; # managed by
        Certbot
    include /etc/letsencrypt/options-ssl-nginx.conf; # managed by Certbot
    ssl_dhparam /etc/letsencrypt/ssl-dhparams.pem; # managed by Certbot
}
```

Naturally, you would want to replace all instances of `tryiac.com` with whatever domain you're using.

Now, let's just update your `templates/nginx-lb.conf.jinja` with the certbot's (Let's Encrypt) additions:

`conf`

```
server {
    if ($host = www.tryiac.com) {
        return 301 https://$host$request_uri;
    } # managed by Certbot
    if ($host = tryiac.com) {
        return 301 https://$host$request_uri;
    } # managed by Certbot

    listen 80;
    server_name tryiac.com www.tryiac.com *.tryiac.com;
    return 404; # managed by Certbot
}
```

Once again replace `tryiac.com` where needed and default to Certbot's (Let's Encrypt) paths for the SSL certificates.

Chapter 9

# Final Thoughts

## Chapter 9

# Final Thoughts

This book has been about sustainably and reliably deploying Django in production using Docker containers, NGINX for load balancing, virtual machines on Linode, a managed database on Linode, a managed object storage on Linode, CI/CD with GitHub, and IaC automation with Terraform and Ansible. What I love about each piece of technology we discussed here is that there's so much more we could dive into. The majority of these chapters could probably have books dedicated to them on their own.

I hope this book will be the beginning or continuation of your journey into the fascinating world of production workloads and applications. Production may feel like the end of a journey, but I believe it's really where the learning begins.

The technical piece of *how to get into production* is important for sure, but what might be more important is answering:

*Does this project even need to exist?*

or

*Does anyone even want this project?*

or

*Am I creating a solution for something that doesn't need a solution?*

or

*Do I have the minimal number of features to test the value this project offers?*

The reason I pose these questions now is to challenge a common trap we engineers and entrepreneurs often fool ourselves into thinking:

*This project needs to exist because it's a good idea*

This statement may even be true, but it's important to consistently test our assumption that it is. Learning while doing work that matters has much more significance to one's well-being than learning while working on something that doesn't. What can make matters worse is working on something that never mattered in the first place, especially if you wasted hours/days/months/years/decades on it. Be a scientist and constantly test assumptions, observe outcomes, form new assumptions, and repeat.

I hope that you take this book as a guide for taking action on continuously testing and deploying your projects and your ideas.

Thank you for taking the time with me.

I hope you enjoyed it. If you did, please shoot me send me a message on Twitter: [@justinmitchel](https://twitter.com/justinmitchel).

Thank you!

Justin Mitchel

# Appendix



## Appendix A

# GitHub Actions Secrets Reference

The final GitHub Action secrets for this book are as follows:

```
1. `ALLOWED_HOST`
2. `DJANGO_DEBUG`
3. `DJANGO_SECRET_KEY`
4. `DJANGO_STORAGE_SERVICE`
5. `DJANGO_VM_COUNT`
6. `DOCKERHUB_APP_NAME`
7. `DOCKERHUB_TOKEN`
8. `DOCKERHUB_USERNAME`
9. `LINODE_BUCKET`
10. `LINODE_BUCKET_REGION`
11. `LINODE_BUCKET_ACCESS_KEY`
12. `LINODE_BUCKET_SECRET_KEY`
13. `LINODE_IMAGE`
14. `LINODE_OBJECT_STORAGE_DEVOPS_BUCKET`
15. `LINODE_OBJECT_STORAGE_DEVOPS_BUCKET_ENDPOINT`
16. `LINODE_OBJECT_STORAGE_DEVOPS_TF_KEY`
17. `LINODE_OBJECT_STORAGE_DEVOPS_ACCESS_KEY`
18. `LINODE_OBJECT_STORAGE_DEVOPS_SECRET_KEY`
19. `LINODE_PA_TOKEN`
20. `MYSQL_DATABASE`
21. `MYSQL_DB_CERT`
22. `MYSQL_DB_HOST`
23. `MYSQL_DB_ROOT_PASSWORD`
24. `MYSQL_DB_PORT`
25. `MYSQL_PASSWORD`
26. `MYSQL_USER`
27. `ROOT_USER_PW`
28. `SSH_DEVOPS_KEY_PUBLIC`
29. `SSH_DEVOPS_KEY_PRIVATE`
```

If you do not have these secrets, there's a good chance your project will not work as intended.



## Appendix B

# Final Project Structure

We have a lot of code in this book, so I created this appendix for your reference. Use this as a guide for the final project structure. You can find the final project code at:

<https://github.com/codingforentrepreneurs/deploy-django-linode-mysql>

## Final Project Structure

```
.
├── Dockerfile
├── LICENSE
├── Makefile
├── README.md
├── articles
│   ├── __init__.py
│   ├── __pycache__
│   ├── admin.py
│   ├── apps.py
│   ├── management
│   │   ├── __init__.py
│   │   ├── __pycache__
│   │   └── commands
│   │       ├── __init__.py
│   │       ├── __pycache__
│   │       └── backup_articles.py
│   ├── migrations
│   │   ├── 0001_initial.py
│   │   ├── 0002_article_updated_by.py
│   │   ├── __init__.py
│   │   └── __pycache__
│   ├── models.py
│   ├── templates
│   │   └── articles
│   │       ├── article_detail.html
│   │       └── article_list.html
│   ├── tests.py
│   ├── urls.py
│   ├── utils.py
│   └── views.py
└── certs
```

```
├── db-test.crt
├── db.crt
cfe-django-blog.code-workspace
cfeblog
├── __init__.py
├── __pycache__
├── asgi.py
├── dbs
│   ├── __init__.py
│   ├── __pycache__
│   ├── mysql.py
│   └── postgres.py
├── settings.py
├── storages
│   ├── __init__.py
│   ├── __pycache__
│   ├── backends.py
│   ├── client.py
│   ├── conf.py
│   ├── mixins.py
│   └── services
│       ├── __init__.py
│       ├── __pycache__
│       └── linode.py
├── urls.py
└── wsgi.py
config
├── entrypoint.sh
db-init
├── init.sql
└── mysql-config.cfg
devops
├── ansible
│   ├── ansible.cfg
│   ├── django-app
│   │   ├── handlers
│   │   │   └── main.yaml
│   │   └── tasks
│   │       └── main.yaml
│   └── docker-install
│       ├── handlers
│       │   └── main.yaml
│       └── tasks
│           └── main.yaml
```

```

├── inventory.ini
├── main.yaml
├── nginx-lb
│   ├── handlers
│   │   └── main.yaml
│   └── tasks
│       └── main.yaml
├── templates
│   ├── docker-compose.yaml.jinja2
│   └── nginx-lb.conf.jinja
├── vars
│   └── main.yaml
├── tf
│   ├── backend
│   ├── linodes.tf
│   ├── locals.tf
│   ├── main.tf
│   ├── outputs.tf
│   ├── templates
│   │   └── ansible-inventory.tpl
│   └── variables.tf
├── docker-compose.prod.yaml
├── docker-compose.yaml
├── fixtures
│   ├── articles.json
│   └── auth.json
├── keys
│   ├── tf-github
│   └── tf-github.pub
├── manage.py
├── mediafiles
│   ├── articles
│   │   └── hello-world
│   │       └── 07472194-a6d5-11ec-887f-acde48001122.jpg
├── requirements.txt
├── staticfiles
│   └── empty.txt
├── staticroot
│   └── empty.txt
├── templates
│   ├── base.html
│   └── navbar.html

```

44 directories, 73 files

## Appendix C

# **.gitignore and .dockerignore** **Reference Examples**

**.gitignore / .dockerignore :**

```
# mysql related
db-init/
certs/db.crt

# environment variables
.env.prod

# devops related
devops/tf/backend
devops/ansible/vars/main.yaml
devops/ansible/inventory.ini
devops/ansible/ansible.cfg
keys/tf-github
keys/tf-github.pub

# django related
staticroot/admin/
scripts/

# Byte-compiled / optimized / DLL files
__pycache__/
*.py[cod]
*$py.class

# C extensions
*.so

# Distribution / packaging
.Python
build/
develop-eggs/
dist/
downloads/
eggs/
.eggs/# .tfstate files
```

```
lib/
lib64/
parts/
sdist/
var/
wheels/
pip-wheel-metadata/
share/python-wheels/
*.egg-info/
.installed.cfg
*.egg
MANIFEST

# PyInstaller
# Usually these files are written by a python script from a template
# before PyInstaller builds the exe, so as to inject date/other infos into it.
*.manifest
*.spec

# Installer logs
pip-log.txt
pip-delete-this-directory.txt

# Unit test / coverage reports
htmlcov/
.tox/
.nox/
.coverage
.coverage.*
.cache
nosetests.xml
coverage.xml
*.cover
*.py,cover
.hypothesis/
.pytest_cache/

# Translations
*.mo
*.pot

# Django stuff:
*.log
local_settings.py
```

```
db.sqlite3
db.sqlite3-journal

# Flask stuff:
instance/
.webassets-cache

# Scrapy stuff:
.scrapy

# Sphinx documentation
docs/_build/

# PyBuilder
target/

# Jupyter Notebook
.ipynb_checkpoints

# IPython
profile_default/
ipython_config.py

# pyenv
.python-version

# pipenv
# According to pypa/pipenv#598, it is recommended to include Pipfile.lock in version
# control.
# However, in case of collaboration, if having platform-specific dependencies or
# dependencies
# having no cross-platform support, pipenv may install dependencies that don't work,
# or not
# install all needed dependencies.
#Pipfile.lock

# PEP 582; used by e.g. github.com/David-OConnor/pyflow
__pypackages__/

# Celery stuff
celerybeat-schedule
celerybeat.pid

# SageMath parsed files
```

```
*.sage.py

# Environments
.env
.venv
env/
venv/
ENV/
env.bak/
venv.bak/

# Spyder project settings
.spyderproject
.spyproject

# Rope project settings
.ropeproject

# mkdocs documentation
/site

# mypy
.mypy_cache/
.dmypy.json
dmypy.json

# Pyre type checker
.pyre/

# Local .terraform directories
**/.terraform/*

# .tfstate files
*.tfstate
*.tfstate.*

# Crash log files
crash.log
crash.*.log

# Exclude all .tfvars files, which are likely to contain sensitive data, such as
# password, private keys, and other secrets. These should not be part of version
# control as they are data points which are potentially sensitive and subject
# to change depending on the environment.
```

```
*.tfvars
*.tfvars.json

# Ignore override files as they are usually used to override resources locally and
# are not checked in.
override.tf
override.tf.json
*_override.tf
*_override.tf.json

# Include override files you do wish to add to version control using negated pattern
# !example_override.tf

# Include tfplan files to ignore the plan output of command: terraform plan
  -out=tfplan
# example: *tfplan*

# Ignore CLI configuration files
.terraformrc
```



## Appendix D

# Self-Hosted Runners with GitHub Actions

It should be no surprise that GitHub Actions have limited resources available for you to use. Not to worry, we can upgrade what resource(s) we may need with our own custom backend (self-hosted runner) to execute the GitHub Actions workflows in any given repository.

Here are some of the resources available to us by default ([read more here](#)):

- 14GB of SSD Disk Space
- 7GB of RAM
- 2-Core CPU

Amazingly, we can run much of what we need for free with these specs. If you want to use more than 14GB (or these specs), you can easily use a [self-hosted runner](#) for GitHub Actions, which means that instead of using GitHub's virtual machines, you can bring your own.

The process is simple:

1. Boot a virtual machine (perhaps on Linode?)
2. Run the code given to you to be connected as a runner (see below)
3. Turn on/off the virtual machine as needed

To set up, all we need to do is:

1. Go to your repo's settings (such as <https://github.com/your-username/your-private-repo/settings>)
2. Select Actions
3. Runners
4. Click **New self-hosted runner**, a few options for running a self-hosted runner will appear
5. Change **runs-on** wherever needed:

To use our **self-hosted** runner, we need to let our workflows know:

Change **runs-on: ubuntu-latest** to **runs-on: self-hosted**

From this:

yaml

```
name: Some workflow

on:
  workflow_dispatch:
jobs:
  mysql_client:
    runs-on: ubuntu-latest
    steps:
      - name: Update packages
        run: sudo apt-get update
```

To this:

yaml

```
name: Some workflow

on:
  workflow_dispatch:
jobs:
  mysql_client:
    runs-on: self-hosted
    steps:
      - name: Update packages
        run: sudo apt-get update
```

And that's it! Pretty neat, huh?

## Appendix E

# Using the GitHub CLI for Setting Action Secrets

The GitHub CLI can do nearly anything you need to do for your applications and is a great tool for automation. One of the core tenants of this book is portability, which is why I opted to leave this section as an appendix item instead of making it a core requirement for the book.

You can probably use the GitHub CLI without using GitHub, but that's a customization rabbit hole I do not want to go down in this book.

Yes, the `.github/workflows` folder was designed for GitHub, but as we discussed in [Chapter 2](#) will allow us to execute these workflows anywhere, thus making them portable.

Either way, using the GitHub CLI to set our GitHub Action Secrets automatically is a good idea. Here's how you'll do that:

## 1. Install GitHub CLI

### [Reference Docs](#)

#### macOS with Homebrew

```
brew install gh
```

#### Windows with Chocolatey

```
choco install gh
```

## Linux Ubuntu

This is copied directly from [this document](#) on the GitHub CLI [repo](#)

```
curl -fsSL https://cli.github.com/packages/githubcli-archive-keyring.gpg | sudo dd
of=/usr/share/keyrings/githubcli-archive-keyring.gpg
echo "deb [arch=$(dpkg --print-architecture) signed-by=/usr/share/keyrings/github
cli-archive-keyring.gpg] https://cli.github.com/packages stable main" | sudo tee /
etc/apt/sources.list.d/github-cli.list > /dev/null
sudo apt update
sudo apt install gh
```

## 2. Log in to GitHub CLI ( `gh` )

1. Create a Personal Access Token [here](#) (docs).

Include the following scope(s) at a minimum:

- `workflow`
- `read:org`
- `read:user`

2. Add Personal Access Token to a file called `my-gh-persona-token.txt`

3. Authenticate with GH

```
gh auth login --with-token < my-gh-persona-token.txt
```

### 3. Set a GitHub Action Secret via CLI

Assuming you have your database certificate available (via `db.crt`), you can run:

```
gh secret set MYSQL_DB_CERT -a actions --repo your-username/your-private-rep < db.crt
```

Or a full example like:

```
gh secret set MYSQL_DB_CERT -a actions --repo codingforentrepreneurs/deploy-  
django-linode-mysql < db.crt
```

There are many other commands and options for setting secrets in the GitHub CLI docs [here](#).

## Appendix F

# Makefile

In many of my projects, I use a **Makefile** so that I can write commands like:

```
make infra_up
make infra_down
make migrate
```

For me, using **make** is just an easy way to remember which commands I use for my local development.

### Installation

#### macOS / Linux

If you are running macOS or *Linux*, you should not have to install anything.

#### Windows with Chocolatey

```
choco install make
```

## Example Makefile

path/to/your/project/Makefile

makefile

```
tf_console:
    terraform -chdir=devops/tf/ console

tf_plan:
    terraform -chdir=devops/tf/ plan

tf_apply:
    terraform -chdir=devops/tf/ apply

tf_upgrade:
    terraform -chdir=devops/tf/ init -upgrade

ansible:
    ANSIBLE_CONFIG=devops/ansible/ansible.cfg venv/bin/ansible-playbook devops/
    ansible/main.yaml

infra_up:
    terraform -chdir=devops/tf/ apply
    ANSIBLE_CONFIG=devops/ansible/ansible.cfg venv/bin/ansible-playbook devops/
    ansible/main.yaml

infra_down:
    terraform -chdir=devops/tf/ apply -destroy

infra_init:
    terraform -chdir=devops/tf/ init -backend-config=backend
```

Now, each block corresponds to a command:

```
make tf_console
```

This will enter our Terraform console in relation to the Terraform files in `devops/tf/`

```
make ansible
```

This will use the virtual environment version of Ansible (we don't need to activate the virtual environment for this to run correctly) and then apply the `main.yaml` playbook.

```
make infra_up
```

This will make all infrastructure changes via Terraform and all infrastructure configuration changes via Ansible.





CLOUD COMPUTING SERVICES FROM



# Cloud Computing Developers Trust

[linode.com](https://linode.com) | Support: 855-4-LINODE | Sales: 844-869-6072  
249 Arch St., Philadelphia, PA 19106 Philadelphia, PA 19106