Zero Downtime Migrations in Django

Ryan Scott

Introduction

- Software Engineer @ Percipient Networks
- Several years using Python and Django for fun
- 7 months using Python and Django professionally
- ~1 year using Python and Django professionally

Topics

— What does zero downtime mean?

— What is a zero downtime deployment?

— What is a migration?

— How does Django query for data?

— How do you migrate without downtime?

WEBSITE
UNDER
CONSTRUCTION

What does zero downtime mean?

- End-users can continue using the website without noticing anything while you deploy.
- All services remain functioning.
- No downtime when you make changes!

Zero downtime deployments

- Without zero downtime deployments, zero downtime migrations are useless.
- Deploying without downtime requires multiple Django servers (load balanced).
- Rolling deploy: Deploy to one web server at a time
- Blue/green deploy: Stand up a duplicate set of web servers, swap them out after the deploy has finished

What's a migration?

- Migrations allow Django to update the database schema from changes you've made to models.
- Migrations maintain a history of your database state.
- Migrations produce consistent results so that you can mirror development and production database state.

Zero downtime migrations

- How do you make database changes without breaking Django?
- How do you keep your Django app running while changing the database?

Example model

— A Customer represents a company with a name and a plan.

```
class Customer(models.Model):
    company_name = models.CharField(max_length=32)
    plan = models.CharField(max_length=32)  # paid, free, etc.
```

How does Django query for data?

— Django query:

```
Customer.objects.all()
```

— Translates to this SQL query:

```
SELECT company_name, plan FROM customer;
```

— NOT:

```
SELECT * FROM customer;
```

Leveraging Django's querying method

- Django only queries for fields that it knows about.
- Fields can exist in the database without being defined in Django.
- Django will raise an exception if a field is defined, but isn't in the database.

Migration Strategies

- Adding a field
- Removing a field

Adding a field

- Add a new field for the company's address. The field *can* be blank.

```
class Customer(models.Model):
    company_name = models.CharField(max_length=32)
    plan = models.CharField(max_length=32)  # paid, free, etc.
    company_address = models.CharField(blank=True, max_length=100)
```

Adding a field

Steps to deploy the new field:

1. Migrate the database so that the new field exists.
2. Deploy your code to Django.

This ensures that the field exists in the database *before* Django starts using it.

Removing a field

— Remove the plan field because we decided we don't need it

```
class Customer(models.Model):
    company_name = models.CharField(max_length=32)
    # Removed plan field
    company_address = models.CharField(blank=True, max_length=100)
```

Removing a field

Steps to deploy the deleted field:

1. Deploy your code to Django.
2. Migrate the database so that the old field is deleted.

This ensures that Django stops using the field *before* it is deleted from the database.

Complex migration strategies

— `RunPython` can be used in Django migrations for migrating data (backfilling).

— Exceptions in `RunPython` functions can leave your migration in a weird state.

— Management commands often work better for data migrations because they are easier to test and can be run repeatedly.

Complex migration strategies

— More complex migrations can be designed by building on these basic steps.

— *Remember*: Understanding how Django queries for data will help you determine the order for migrating and deploying.

— Testing complex migrations: `github.com/plumdog/django_migration_testcase`

Thank you

github.com/percipient/talks

ryan@strongarm.io