Paris.py Meetup

Keeping Django migrations backward compatible

June 25th, 2019

Hi, I’m David

- Currently, API Engineer @ Botify
  - Technical SEO platform

- Formerly, Software Engineer @ 3YOURMIND
  - 3D printing management platform

What is a Django?

- A popular Python web framework
- *“The web framework for perfectionists with deadlines”*
- MVC architecture, but
  - Controller => View
  - View => Template
    - Thus, MTV architecture (?)
- Fast project start, given structure and out of the boxes features
  - Users, groups
  - Administration panel

What a migration system?

- Migrations handle the synchronisation between your models and database
  - Enables the use of the Django ORM
- Python code that represents a database schema change
- History of changes to a model
- Very fast prototyping and easy database schema altering over time
  - For the best and for the worse

Implications

Migrations link tightly the version of the code to the database

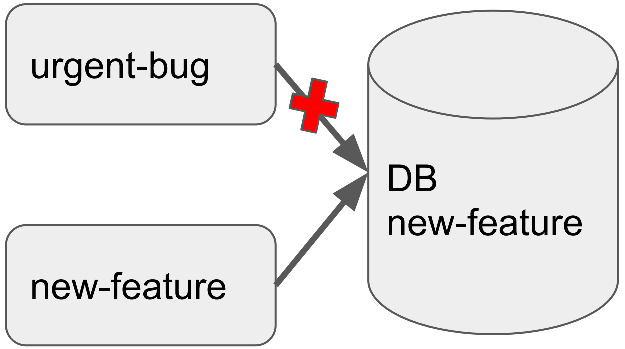
Problems

- Code evolves faster than the database
- Code is shared and copied easier than a database
- The database tends to the latest version of the code
- The database schema is contained in the code

Let’s get through some examples quickly!

Illustration #1 - Local development

1. Add a *NOT NULL* field to a model
2. Urgent bug to fix and switch to another branch
3. Start developing and testing
4. `ERROR: new_field cannot be NULL`
5. Rollback my database
6. `No change: latest version already applied`
7. ??? The code doesn't know the new migration
8. Switch back to feature branch
9. Rollback database
10. Switch back to urgent bug branch



But what if I set a default value?

Problem: Django handles database DEFAULT values on the application side.
And only uses the database to fill existing rows. Example:

my_app/migrations/ABCD_add_field.py

```
operations = [
  migrations.AddField(
    model_name='my_model',
    name='my_field',
    field=models.CharField(
      default=b'my_default',
      max_length=255,
    ),
  ),
]
```

$ python manage.py sqlmigrate

```
-- Add field my_field to my_model

ALTER TABLE `my_app_my_model`
  ADD COLUMN `my_field` varchar(255)
  DEFAULT 'my_default' NOT NULL;

ALTER TABLE `my_app_my_model`
  ALTER COLUMN `my_field` DROP DEFAULT;

COMMIT;
```

Illustration #2 - Development with shared DB

Imagine the same process, but sharing the DB with your colleagues 😱

- This environment shouldn't happen too often for day-to-day/local development purposes
- But pre-production/staging/sandbox environments can be subject to frequent deployments of different migrations

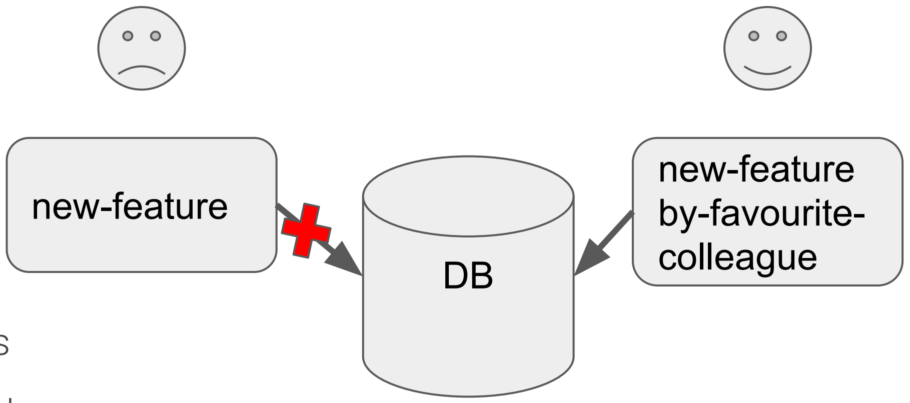


Illustration #2.1 - Development with shared DB

Or even better!

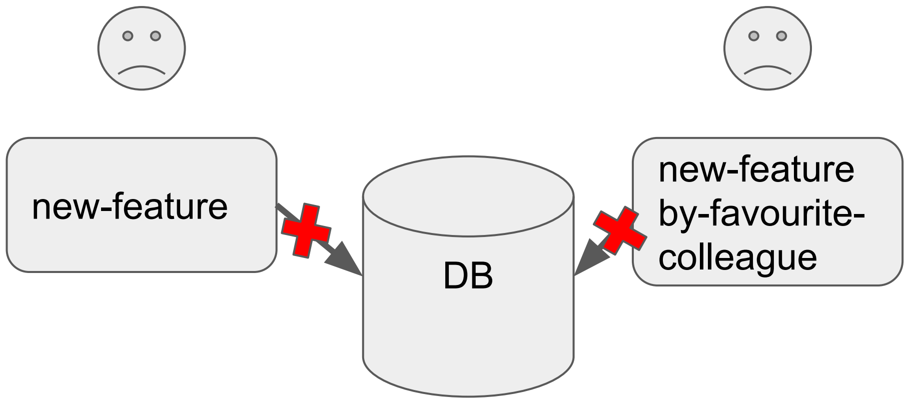


Illustration #3 - Product stable and feature versions

- On-premise software
  - Stable version: few features, few bugs
- Hosted software (SaaS)
  - Agile process and frequent deployments
- But, shared database
  - For important shared and up-to--date business knowledge

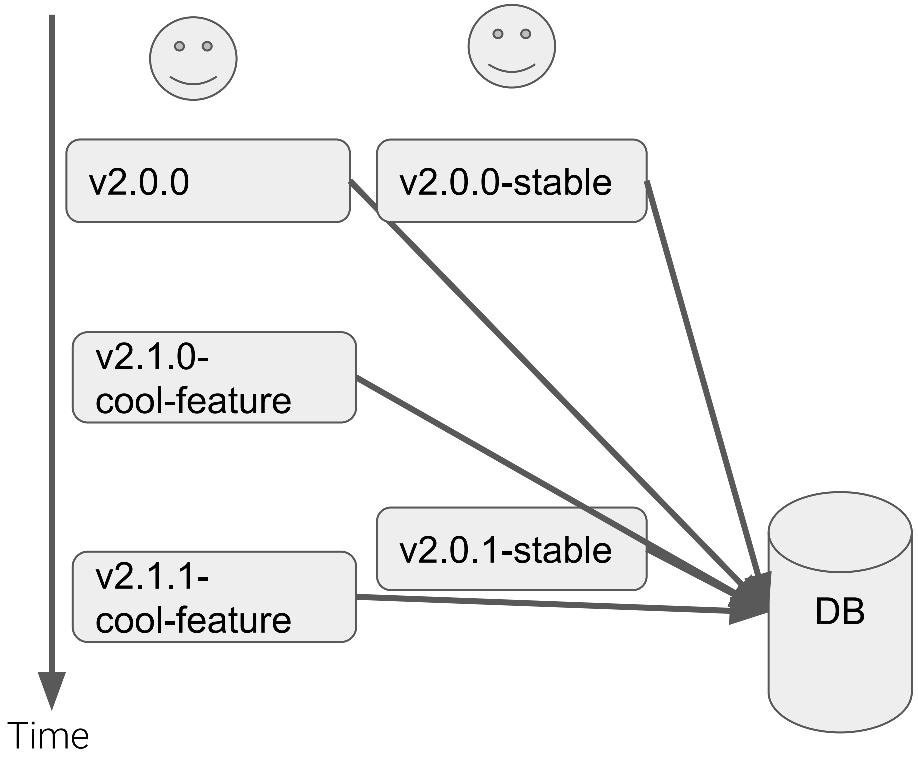


Illustration #4 - Blue/Green Deployment

(or Canary release)

- Along to your old software
- You deploy new software
- And progressively/instantly route traffic to the new software

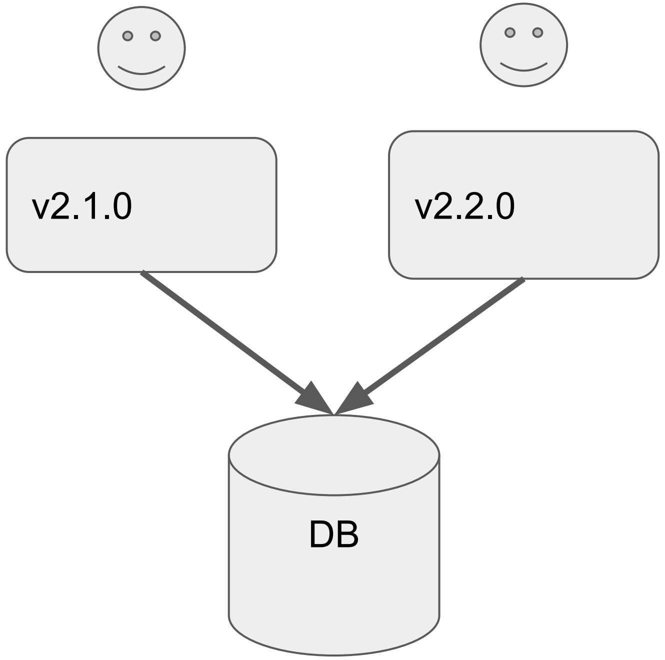
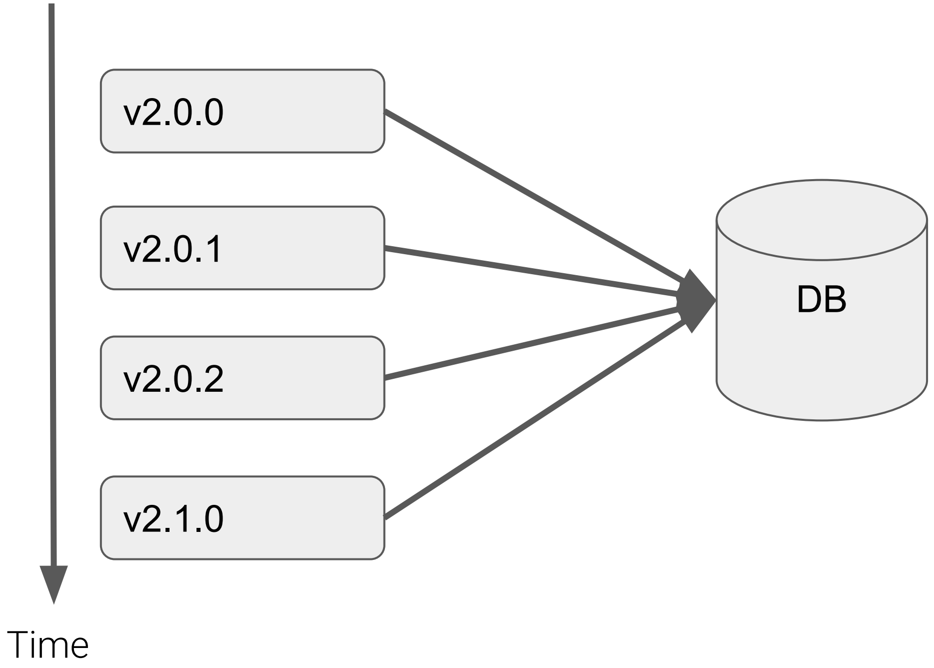


Illustration #5 - Rolling deployment

- You have a large number of machines
- You are very much agile and potentially deploy multiple times each hour

Deploying to all machines takes more than an hour!

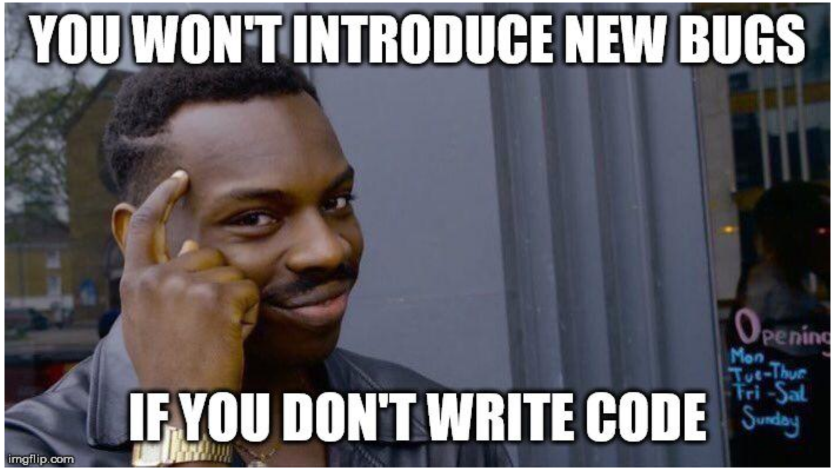


One common pattern

Multiple code versions targeting the same database

How to avoid problems #1 - Avoid doing anything

- Avoid changing your database and your code
- Then you cannot break it from one version to another

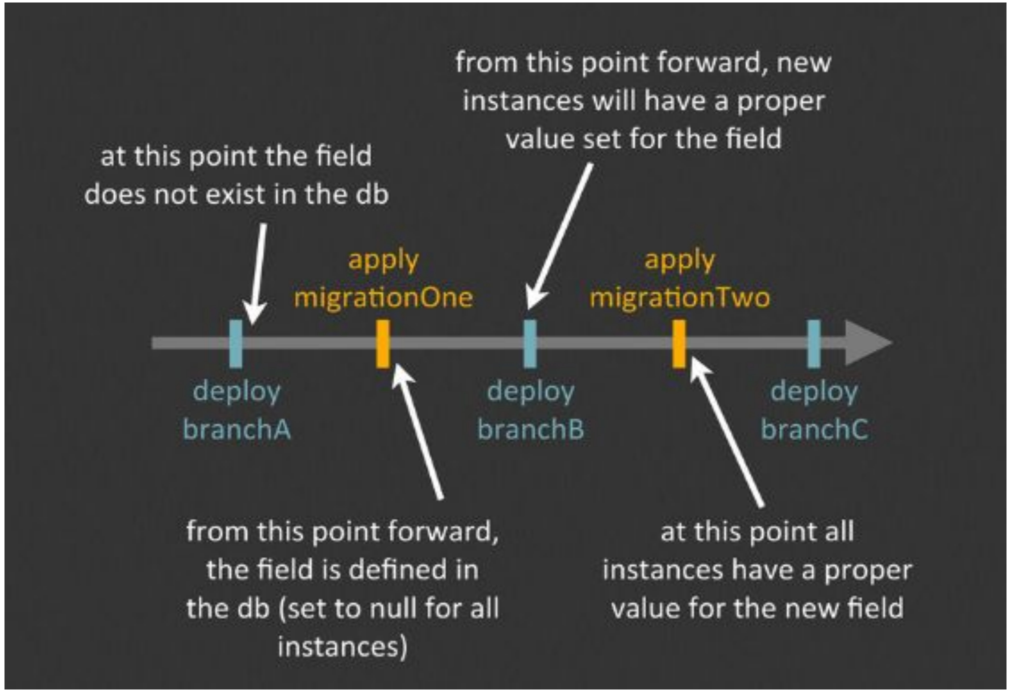


- Not sure business will go well with this philosophy

How to avoid problems #2 - Ignore problems and break anyway

- That might be an acceptable solution
  - Few servers, few developers
  - Some downtime/maintenance time is acceptable
- But difficult rollbacks (with even more downtime)

How to avoid problems #3 - Multiple deployment changes


from this point forward, new instances will have a proper value set for the field
at this point the field does not exist in the db
apply migrationOne
apply migrationTwo
deploy branchA
deploy branchB
deploy branchC
from this point forward, the field is defined in the db (set to null for all instances)
at this point all instances have a proper value for the new field

Advantage:

- You end up with a clean and consistent schema
- Anticipated/no expected downtime

Drawbacks:

- Hard to coordinate through time
  - Especially with multiple changes at the same time
- Complexity
  - Multiple branches
  - Multiple deployments
  - Multiple migrations

Image from: https://speakerdeck.com/nduthoit/the-path-to-smoother-database-migrations

How to avoid problems #4 - Avoid creating problems

- Identify the problem => backward incompatible migrations

1. Don’t allow backward incompatible migrations by default
2. Automatically detect them on each change/Pull Request
3. Do all code changes during one single deployment

Advantage: everything in one PR

The *django-migration-linter*

- An tool to detect the migrations that will break database compatibility with previous version of the code
- https://github.com/3YOURMIND/django-migration-linter
- Easy to integrate into your CI pipeline
  - Django management command with appropriate return code

```
$ python manage.py lintmigrations
(projects, 0001_initial)... OK
(projects, 0002_auto_20190414_1502)... ERR
    RENAMING columns (table: user_projects)
```

What are these incompatible migrations?

- Adding a *NOT NULL* column without default
    - Except if in the same transaction, we actually set a default value
- Dropping a column, renaming a column, renaming a table…
- Potentially altering a column type

And surely many more that are not handled yet by the linter.

Trick for having default value on the database side

- With https://github.com/3YOURMIND/django-add-default-value

my_app/migrations/ABCD_add_field.py

```
operations = [
  migrations.AddField(
    model_name='my_model',
    name='my_field',
    field=models.CharField(
      default=b'my_default',
      max_length=255,
    ),
  ),
  AddDefaultValue(
    model_name='my_model',
    name='my_field',
    value='my_default',
  ),
]
```

$ python manage.py sqlmigrate

```
-- Add field my_field to my_model

ALTER TABLE `my_app_my_model`
  ADD COLUMN `my_field` varchar(255)
  DEFAULT 'my_default' NOT NULL;

ALTER TABLE `my_app_my_model`
  ALTER COLUMN `my_field` DROP DEFAULT;

-- Add to field my_field the default value my_default

ALTER TABLE `my_app_my_model`
  ALTER COLUMN `my_field`
  SET DEFAULT `my_default`;

COMMIT;
```

Downsides of avoiding these changes?

Breaking changes will still need be necessary for data consistency.

Two solutions:

- One breaking change, with expected downtime, from time to time on major version bumps
- Multiple deployments
  - For example: progressively stop using a field before removing it completely and safely

Things the *migration-linter* does not catch (yet!)

- Missing merge migrations (easy to add)
- Warnings about data migrations
    - Potential long operations
    - Potential missing reverse operation for rollbacks
- Warnings about expected downtime
    - Table locking and potentially long operations, depending on the DB vendor
    - Could be crossed with heuristic on table size
- Specific configuration
    - For example that a given table is critical and should never be touched automatically through a migration (because of a huge size for instance)

botify

Thank you
for your attention

Get in touch!
hello@botify.com