Backend performance with Django

Using Django since 2015

Worked for multiple startups (including 2 years at BackMarket)

CTO for 2.5 years

Why ?

Make the application faster

Reduce costs

Reduce load on a specific resource

Improve RAM, CPU usage

Many other good reasons :)

How to monitor ?

The **first step to performances improvements**, is always to have a proper monitoring system.
Pick any Saas in the market: Datadog, Scout APM, New Relic, Sentry APM, Dynatrace, etc.

You can’t improve what you can’t measure

What **you need** to be able to monitor in production when talking about latency

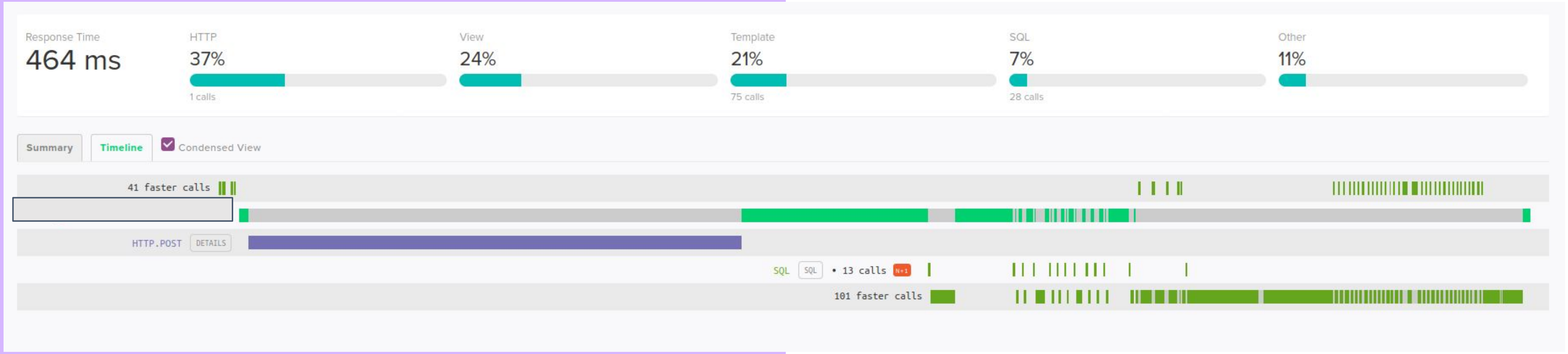
The latency of your application

The most used routes in your application

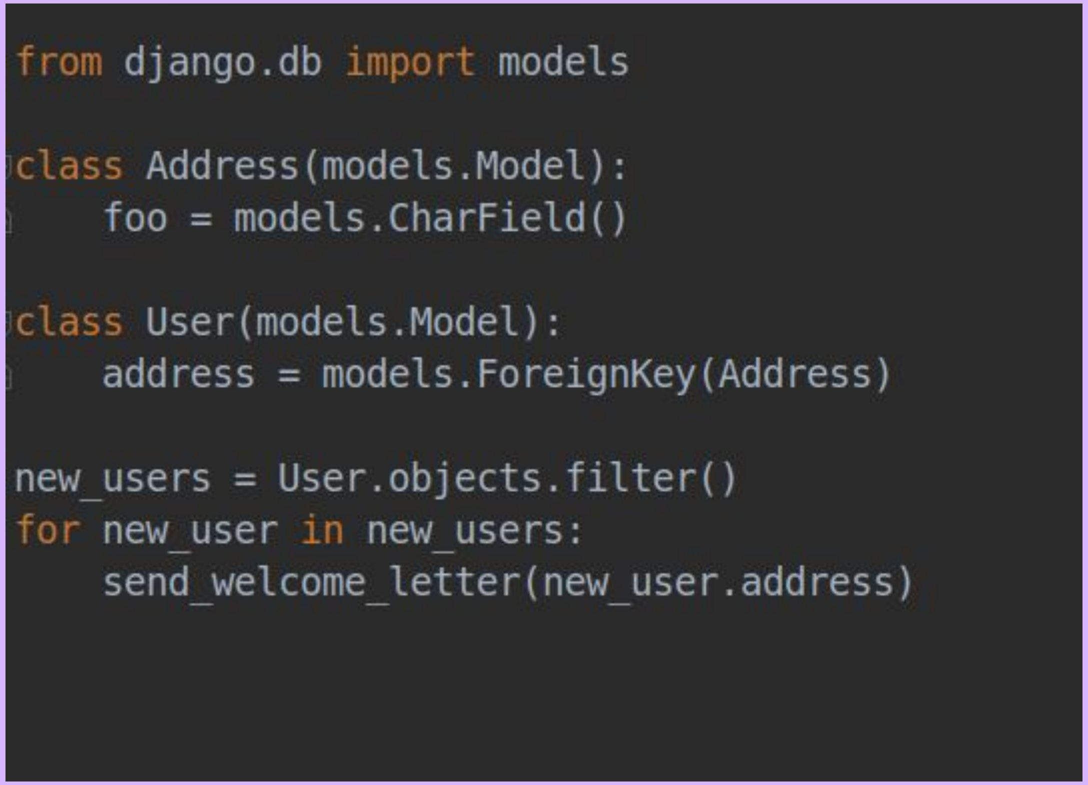
The slowest routes in your application

The routes in which your server spent the more time (number of request * average latency)

Traces for a specific route / URL



Database



```python
from django.db import models

class Address(models.Model):
    foo = models.CharField()

class User(models.Model):
    address = models.ForeignKey(Address)

new_users = User.objects.filter()
for new_user in new_users:
    send_welcome_letter(new_user.address)
```

How do you think this code would scale ?

How many queries are made in this simple snippet ?

Answer

N + 1 queries

Here comes **select_related**

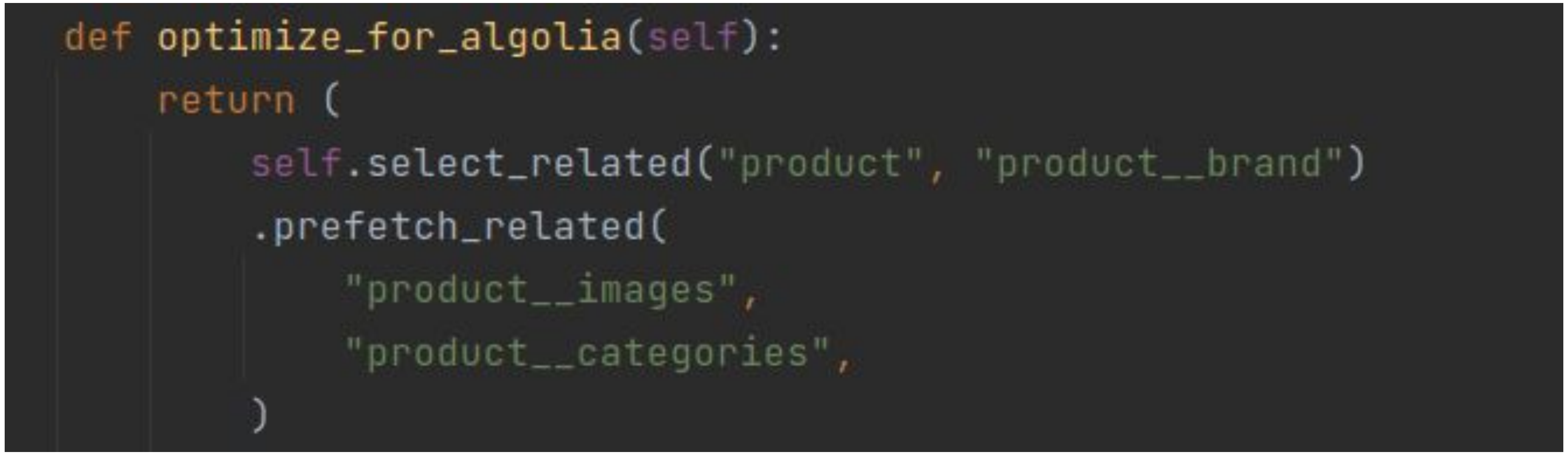
User.objects.
select_related(“address”)

- Returns a QuerySet that will “follow” foreign-key relationships.
- You can refer to any ForeignKey or OneToOneField relation in the list of fields passed to select_related().
- Previous example would be only 1 query when using select_related
    - Uses a JOIN operation to do so

Here comes **prefetch_related**

Pizza.objects.
prefetch_related(“toppings”)

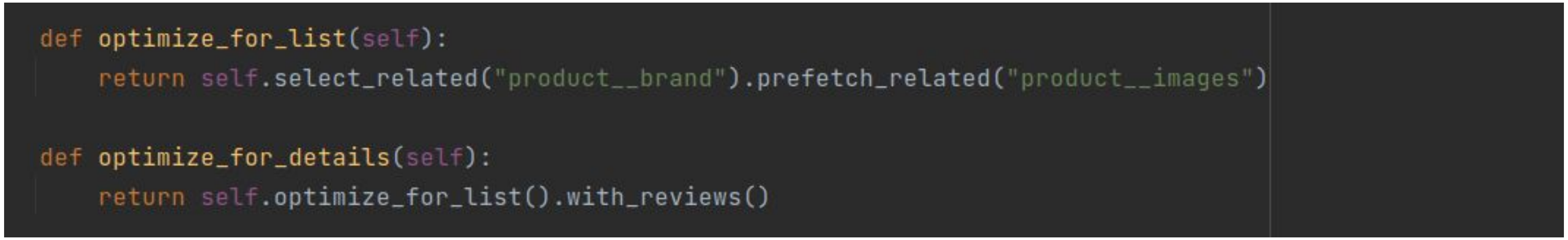
- prefetch_related uses 2 queries and works with ManyToMany and ManyToOne relationship
- returns a queryset of objects with appropriate data already prefetched



```python
def optimize_for_algolia(self):
    return (
        self.select_related("product", "product__brand")
        .prefetch_related(
            "product__images",
            "product__categories",
        )
```

1 query to fetch the Model + Product + Brand
1 query to fetch the images
1 query to fetch the categories

We will always be making 3 queries to get all the data. O(3) instead of O(4n + 1)

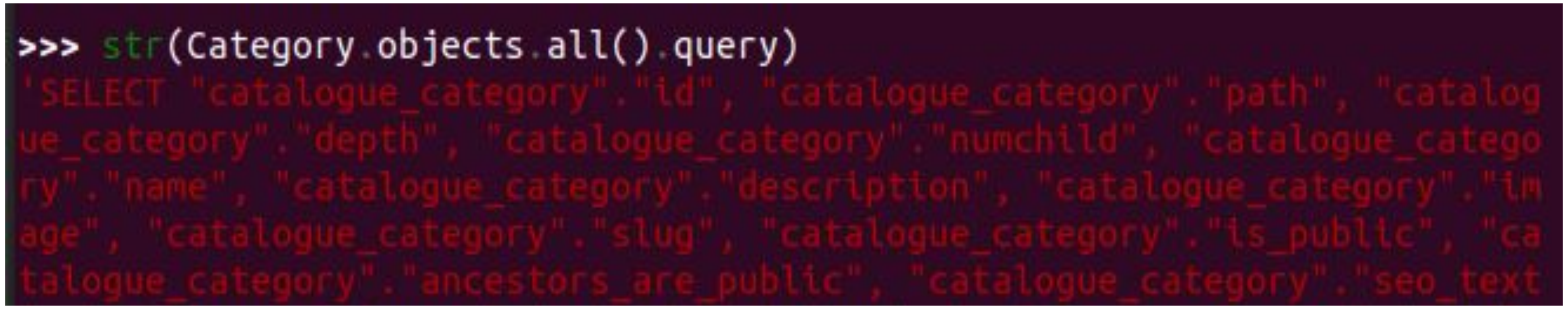


```python
def optimize_for_list(self):
    return self.select_related("product__brand").prefetch_related("product__images")

def optimize_for_details(self):
    return self.optimize_for_list().with_reviews()
```

Use the ORM to compose methods instead of duplicating the logic in every view.

Use different levels of granularity when applicable (optimize for list view, for detail view, etc).

Model's in a Django app generally carries a lot of data.

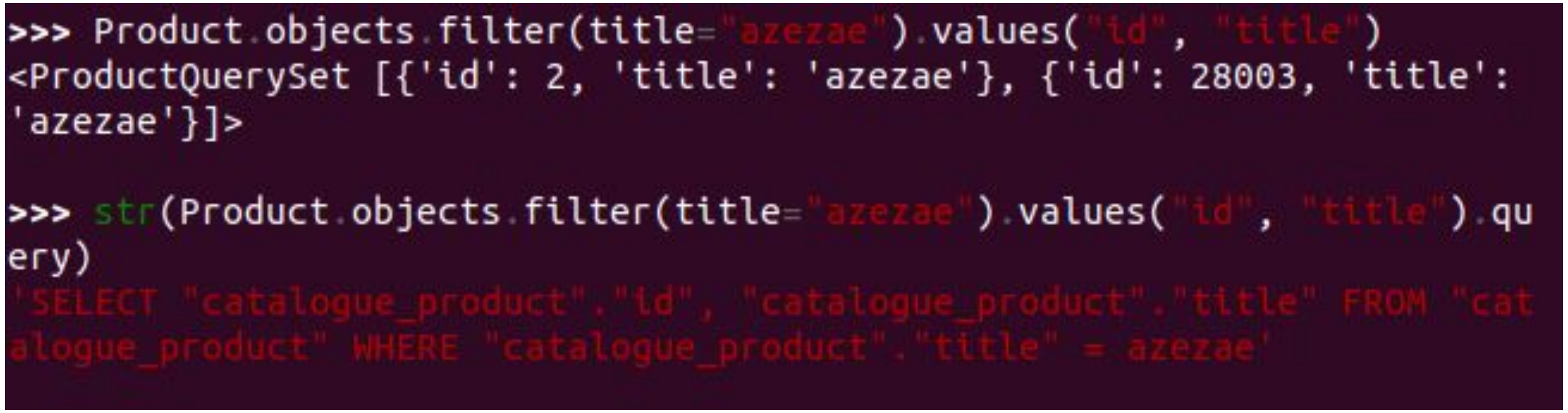


```
>>> str(Category.objects.all().query)
'SELECT "catalogue_category"."id", "catalogue_category"."path", "catalog
ue_category"."depth", "catalogue_category"."numchild", "catalogue_catego
ry"."name", "catalogue_category"."description", "catalogue_category"."im
age", "catalogue_category"."slug", "catalogue_category"."is_public", "ca
talogue_category"."ancestors_are_public", "catalogue_category"."seo_text
```

Do you really need

all this data ?

The .values() method

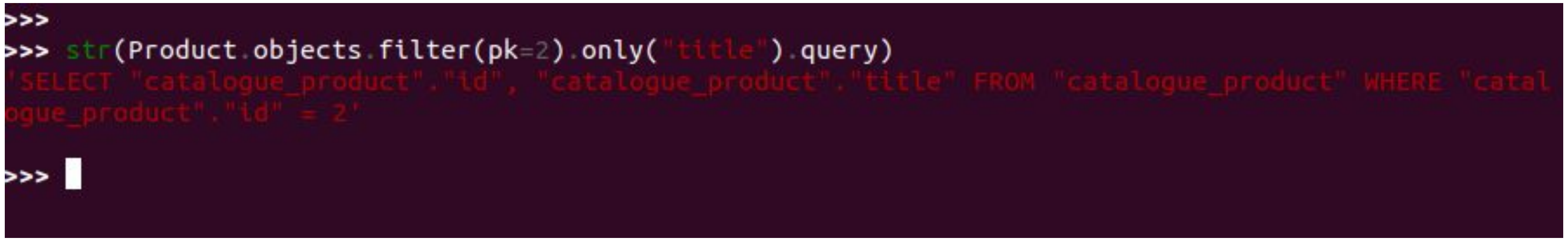


```
>>> Product.objects.filter(title="azezae").values("id", "title")
<ProductQuerySet [{'id': 2, 'title': 'azezae'}, {'id': 28003, 'title':
'azezae'}]>

>>> str(Product.objects.filter(title="azezae").values("id", "title").qu
ery)
'SELECT "catalogue_product"."id", "catalogue_product"."title" FROM "cat
alogue_product" WHERE "catalogue_product"."title" = azezae'
```

Returns an iterable queryset of dicts.

Nice but not super handy to use: you need to adapts your code to use dict instead of the model instance.

The .only() method



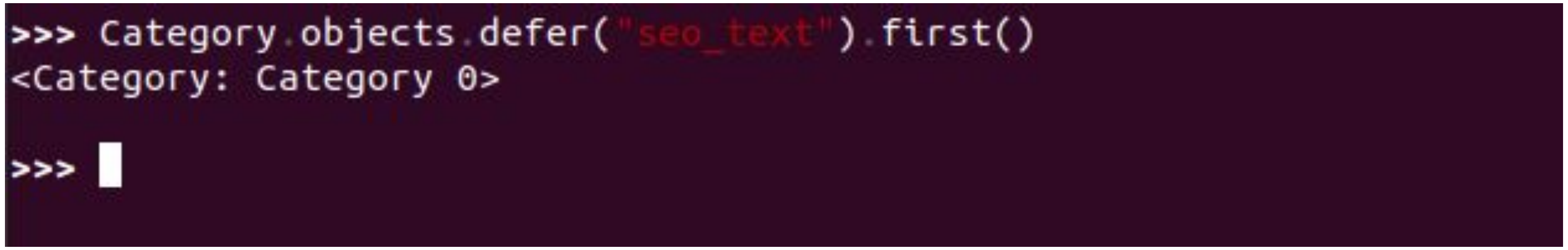
```
>>>
>>> str(Product.objects.filter(pk=2).only("title").query)
'SELECT "catalogue_product"."id", "catalogue_product"."title" FROM "catalogue_product" WHERE "catal
ogue_product"."id" = 2'

>>> 
```

Returns an object with only a few fields queried.

The .only() method

When using only() you only **fetch a few fields.** Be careful of hidden usage of fields inside model property. Always monitor your change with assertNumQueries or the SQL tab of Django Debug Toolbar

The .defer() method



```
>>> Category.objects.defer("seo_text").first()
<Category: Category 0>

>>> 
```

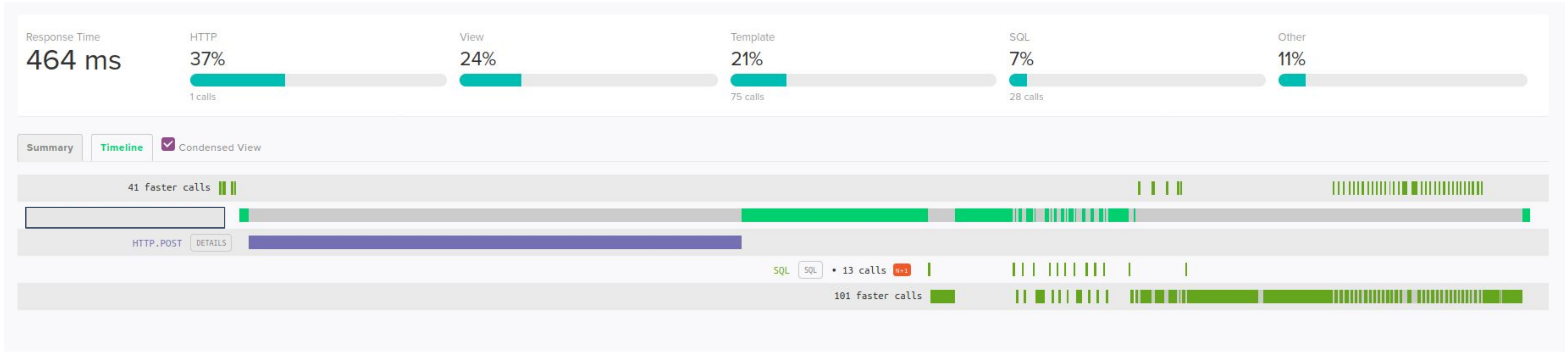
Returns an object with all fields loaded except the one given in the defer method.

Useful to avoid loading big fields, but every new fields added to the model will still be loaded.

Search Engines, PSP, CRM, etc.

External calls

External calls are generally long



Response Time
464 ms

HTTP
37%
1 calls

View
24%

Template
21%
75 calls

SQL
7%
28 calls

Other
11%

No solutions to make them faster - no control

Defer all non critical calls to tasks / queues

Need to mitigate the issue to avoid going down when an external service is slow

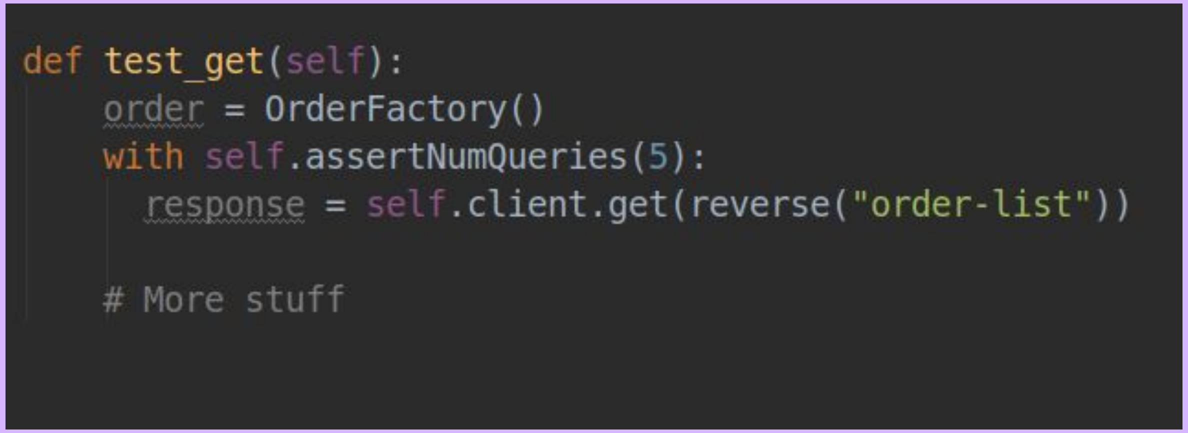
Use timeouts

«Nearly all production code should use this parameter in nearly all requests.
Failure to do so can cause your program to hang indefinitely.»

Requests Documentation

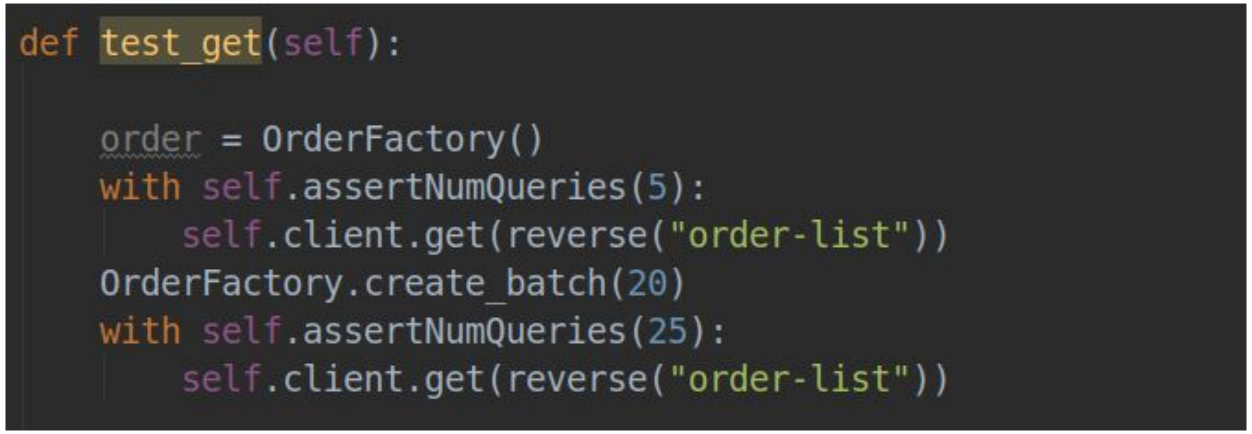
What to improve tomorrow when I get back to work ?

What people do



```python
def test_get(self):
    order = OrderFactory()
    with self.assertNumQueries(5):
      response = self.client.get(reverse("order-list"))

    # More stuff
```

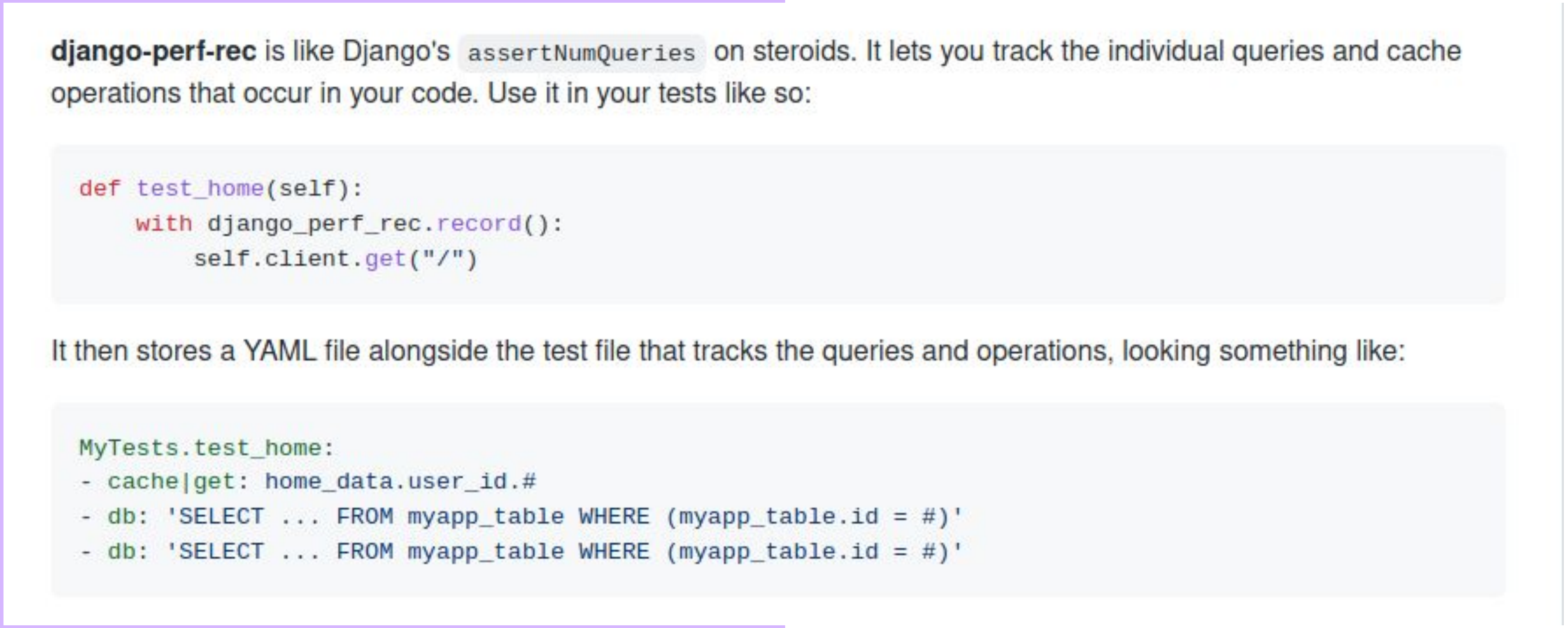
What could be done



```python
def test_get(self):

    order = OrderFactory()
    with self.assertNumQueries(5):
        self.client.get(reverse("order-list"))
    OrderFactory.create_batch(20)
    with self.assertNumQueries(25):
        self.client.get(reverse("order-list"))
```

Going further testing SQL queries

Github : adamchainz / django-perf-rec



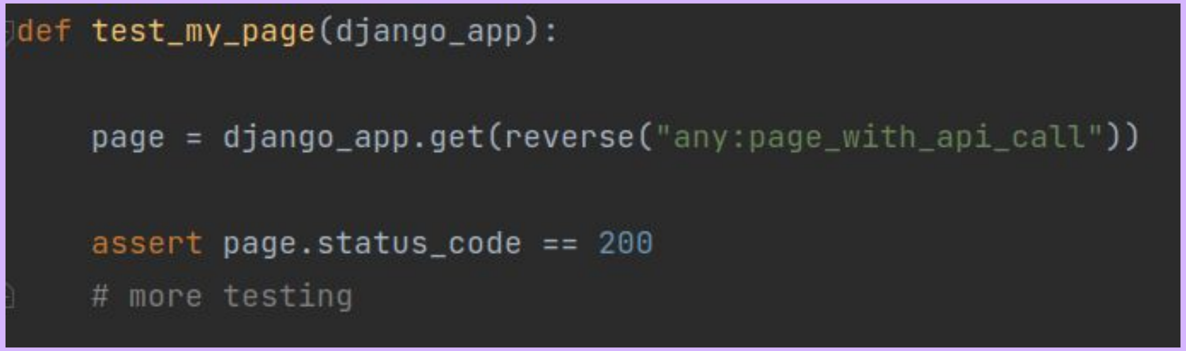
django-perf-rec is like Django's `assertNumQueries` on steroids. It lets you track the individual queries and cache operations that occur in your code. Use it in your tests like so:

```
def test_home(self):
    with django_perf_rec.record():
        self.client.get("/")
```

It then stores a YAML file alongside the test file that tracks the queries and operations, looking something like:

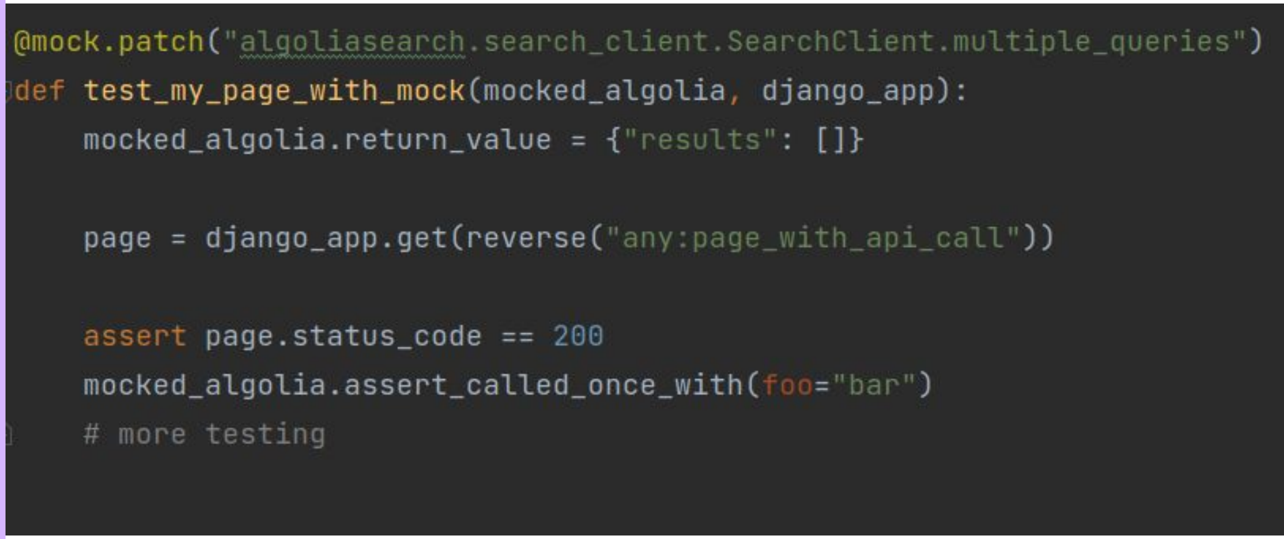
```
MyTests.test_home:
- cache|get: home_data.user_id.#
- db: 'SELECT ... FROM myapp_table WHERE (myapp_table.id = #)'
- db: 'SELECT ... FROM myapp_table WHERE (myapp_table.id = #)'
```

What people do



```python
def test_my_page(django_app):

    page = django_app.get(reverse("any:page_with_api_call"))

    assert page.status_code == 200
    # more testing
```

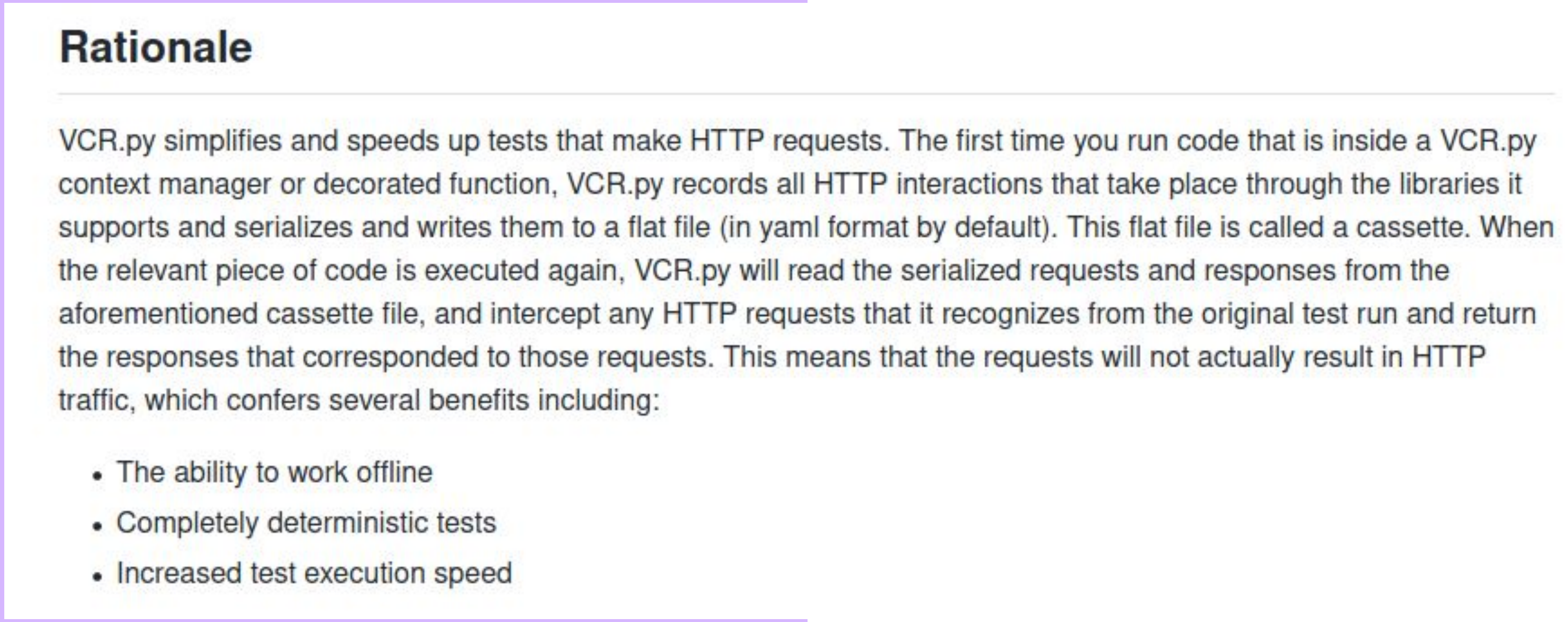
What should be done



```python
@mock.patch("algoliasearch.search_client.SearchClient.multiple_queries")
def test_my_page_with_mock(mocked_algolia, django_app):
    mocked_algolia.return_value = {"results": []}

    page = django_app.get(reverse("any:page_with_api_call"))

    assert page.status_code == 200
    mocked_algolia.assert_called_once_with(foo="bar")
    # more testing
```

Going further testing Network calls

Github : kevin1024 / vcrpy



Rationale

VCR.py simplifies and speeds up tests that make HTTP requests. The first time you run code that is inside a VCR.py context manager or decorated function, VCR.py records all HTTP interactions that take place through the libraries it supports and serializes and writes them to a flat file (in yaml format by default). This flat file is called a cassette. When the relevant piece of code is executed again, VCR.py will read the serialized requests and responses from the aforementioned cassette file, and intercept any HTTP requests that it recognizes from the original test run and return the responses that corresponded to those requests. This means that the requests will not actually result in HTTP traffic, which confers several benefits including:

- The ability to work offline
- Completely deterministic tests
- Increased test execution speed

A few more ideas:

Django Templates are slow

Latency is only one metric

Queuing can help
You don’t need to send email or create PDF inside a request/response cycle

SQL indexes / DBA Stuff

The final checklist

☐ I monitor the latency of my app in production

☐ I am alerted in case of error / latency issues in prod

☐ Django Debug toolbar is installed and used by everyone

☐ My top 5 views by server time spent are optimized (select_related, etc.)

☐ All my external calls have a timeout aligned with the expected quality of service

☐ All critical pages have a performance test on the number of queries made

☐ Any non mandatory external call / computation are off-loaded in crons or tasks

☐ All external calls are tested / mocked (tests can run offline)

☐ For critical views / long computations I load only the minimum needed data from my database (values, only, defer)

☐ Long running tasks are optimized in terms of RAM usage (batches, iterator, etc.)

Antonin MOREL

Any question ?

Come and see me or get in touch on LinkedIn

https://www.linkedin.com/in/antonin-morel-959b5452/