silk Documentation

Release 0.3

Michael Ford

Jun 01, 2017

Contents

CHAPTER 1

Quick Start

Silk is installed like any other Django app.

First install via pip:

```
pip install django-silk
```

Add the following to your `settings.py`:

```
MIDDLEWARE_CLASSES = (
    ...
    'silk.middleware.SilkyMiddleware',
    ...
)

INSTALLED_APPS = (
    ...
    'silk'
)
```

Add the following to your `urls.py`:

```
urlpatterns += patterns('', url(r'^silk', include('silk.urls', namespace='silk')))
```

Run `syncdb` to create Silk's database tables:

```
python manage.py syncdb
```

And voila! Silk will begin intercepting requests and queries which you can inspect by visiting `/silk/`

Other Installation Options

You can download a release from github and then install using pip:

```
pip install django-silk-<version>.tar.gz
```

You can also install directly from the github repo but please note that this version is not guaranteed to be working:

```
pip install -e git+https://github.com/mtford90/silk.git#egg=silk
```

CHAPTER 2

Profiling

Silk can be used to profile arbitrary blocks of code and provides `silk_profile`, a Python decorator and a context manager for this purpose. Profiles will then appear in the 'Profiling' tab within Silk's user interface.

Decorator

The decorator can be applied to both functions and methods:

```
@silk_profile(name='View Blog Post')
def post(request, post_id):
    p = Post.objects.get(pk=post_id)
    return render_to_response('post.html', {
        'post': p
    })
```

```
class MyView(View):
        @silk_profile(name='View Blog Post')
        def get(self, request):
                p = Post.objects.get(pk=post_id)
        return render_to_response('post.html', {
                'post': p
        })
```

Context Manager

`silk_profile` can also be used as a context manager:

```
def post(request, post_id):
    with silk_profile(name='View Blog Post #%d' % self.pk):
        p = Post.objects.get(pk=post_id)
        return render_to_response('post.html', {
```

```
                'post': p
        })
```

Dynamic Profiling

Decorators and context managers can also be injected at run-time. This is useful if we want to narrow down slow requests/database queries to dependencies.

Dynamic profiling is configured via the `SILKY_DYNAMIC_PROFILING` option in your `settings.py`:

```
"""
Dynamic function decorator
"""

SILKY_DYNAMIC_PROFILING = [{
    'module': 'path.to.module',
    'function': 'foo'
}]

# ... is roughly equivalent to
@silk_profile()
def foo():
    pass

"""
Dynamic method decorator
"""

SILKY_DYNAMIC_PROFILING = [{
    'module': 'path.to.module',
    'function': 'MyClass.bar'
}]

# ... is roughly equivalent to
class MyClass(object):

    @silk_profile()
    def bar(self):
        pass

"""
Dynamic code block profiling
"""

SILKY_DYNAMIC_PROFILING = [{
    'module': 'path.to.module',
    'function': 'foo',
    # Line numbers are relative to the function as opposed to the file in which it␣
↪resides
    'start_line': 1,
    'end_line': 2,
    'name': 'Slow Foo'
}]

# ... is roughly equivalent to
def foo():
```

```
    with silk_profile(name='Slow Foo'):
        print (1)
        print (2)
    print(3)
    print(4)
```

Note that dynamic profiling behaves in a similar fashion to that of the python mock framework in that we modify the function in-place e.g:

```
""" my.module """
from another.module import foo

# ...do some stuff
foo()
# ...do some other stuff
```

We would profile `foo` by dynamically decorating *my.module.foo* as opposed to `another.module.foo`:

```
SILKY_DYNAMIC_PROFILING = [{
    'module': 'my.module',
    'function': 'foo'
}]
```

If we were to apply the dynamic profile to the functions source module `another.module.foo` *after* it has already been imported, no profiling would be triggered.

CHAPTER 3

Configuration

Authentication/Authorisation

By default anybody can access the Silk user interface by heading to */silk/*. To enable your Django auth backend place the following in *settings.py*:

```
SILKY_AUTHENTICATION = True  # User must login
SILKY_AUTHORISATION = True  # User must have permissions
```

If `SILKY_AUTHORISATION` is `True`, by default Silk will only authorise users with `is_staff` attribute set to `True`.

You can customise this using the following in `settings.py`:

```
def my_custom_perms(user):
    return user.is_allowed_to_use_silk

SILKY_PERMISSIONS = my_custom_perms
```

Request/Response bodies

By default, Silk will save down the request and response bodies for each request for future viewing no matter how large. If Silk is used in production under heavy volume with large bodies this can have a huge impact on space/time performance. This behaviour can be configured with following options:

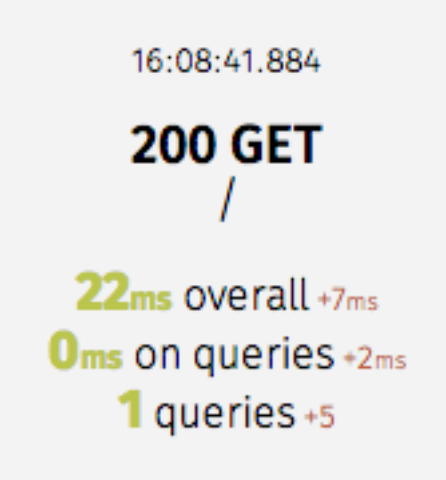
```
SILKY_MAX_REQUEST_BODY_SIZE = -1  # Silk takes anything <0 as no limit
SILKY_MAX_RESPONSE_BODY_SIZE = 1024  # If response body>1024kb, ignore
```

Meta-Profiling

Sometimes its useful to be able to see what effect Silk is having on the request/response time. To do this add the following to your *settings.py*:

```
SILKY_META = True
```

Silk will then record how long it takes to save everything down to the database at the end of each request:



Note that in the above screenshot, this means that the request took 29ms (22ms from Django and 7ms from Silk)

CHAPTER 4

Troubleshooting

The below details common problems when using Silk, most of which have been derived from the solutions to github issues.

Unicode

Silk saves down the request and response bodies of each HTTP request by default. These bodies are often UTF encoded and hence it is important that Silk's database tables are also UTF encoded. Django has no facility for enforcing this and instead assumes that the configured database defaults to UTF.

If you see errors like:

> Incorrect string value: 'xCExBB, xCFx86...' for column 'raw_body' at row...

Then it's likely your database is not configured correctly for UTF encoding.

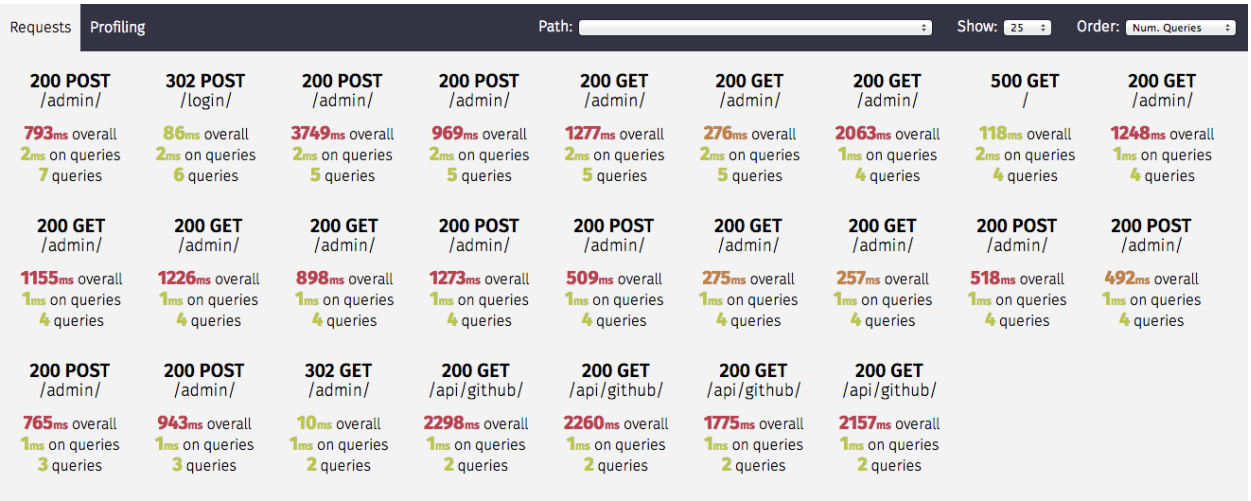
See this github issue for more details and workarounds.

Middleware

The middleware is placement sensitive. If the middleware before `silk.middleware.SilkyMiddleware` returns from `process_request` then `SilkyMiddleware` will never get the chance to execute. Therefore you must ensure that any middleware placed before never returns anything from `process_request`. See the django docs for more information on this.

This GitHub issue also has information on dealing with middleware problems.

Silk is a live profiling and inspection tool for the Django framework. Silk intercepts and stores HTTP requests and database queries before presenting them in a user interface for further inspection:



A **live demo** is available here.

CHAPTER 5

Features

- Inspect HTTP requests and responses
  - Query parameters
  - Headers
  - Bodies
  - Execution Time
  - Database Queries
    - Number
    - Time taken
- SQL query inspection
- Profiling of arbritary code blocks via a Python context manager and decorator
  - Execution Time
  - Database Queries
  - Can also be injected dynamically at runtime e.g. if read-only dependency.
- Authentication/Authorisation for production use

CHAPTER 6

Requirements

- Django: 1.5, 1.6
- Python: 2.7, 3.3, 3.4