Implementing and managing feature flags

How to effectively manage your feature flags in a monolithic Django application

Mehdi Raddadi, Senior Developer
mehdi.raddadi@gitguardian.com

GitGuardian

GitGuardian

In numbers

Inception

Founded in **2017**
Automated secrets detection, in public and private repositories

1

Growth

62 employees
37 R&D
$44M raised in December 2021

3

Tech

~1B public commits scanned/year
~2.5M secrets detected in 2021
#1 GitHub app for security
~1.3M repositories scanned

2

Some of our clients

DATADOG talend instacart
cloudbakers AUTOMOX

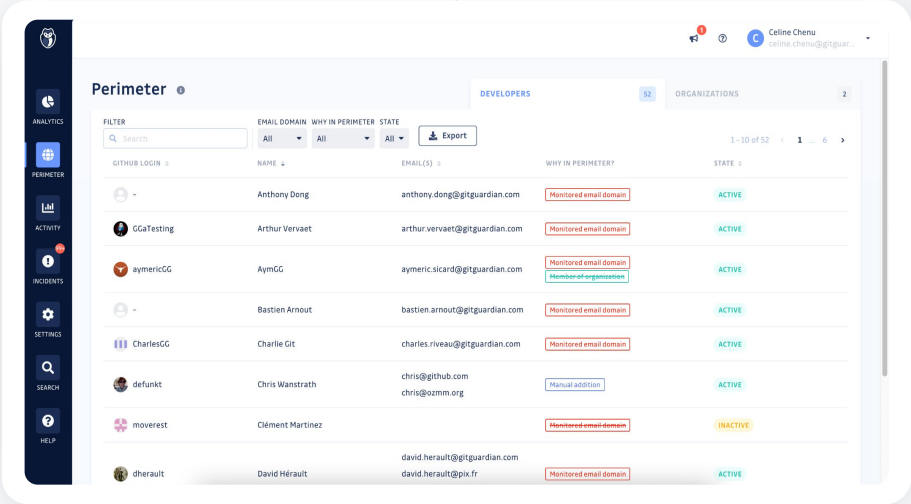
4

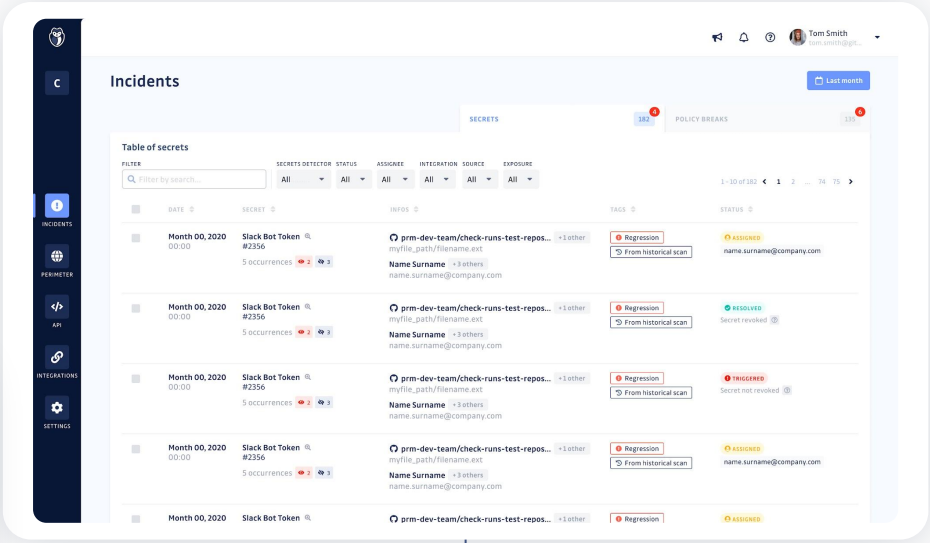
GitGuardian

Products

Public Monitoring

Auto-discover who your developers are on GitHub
Monitor under-the-radar activity
Be alerted in real-time





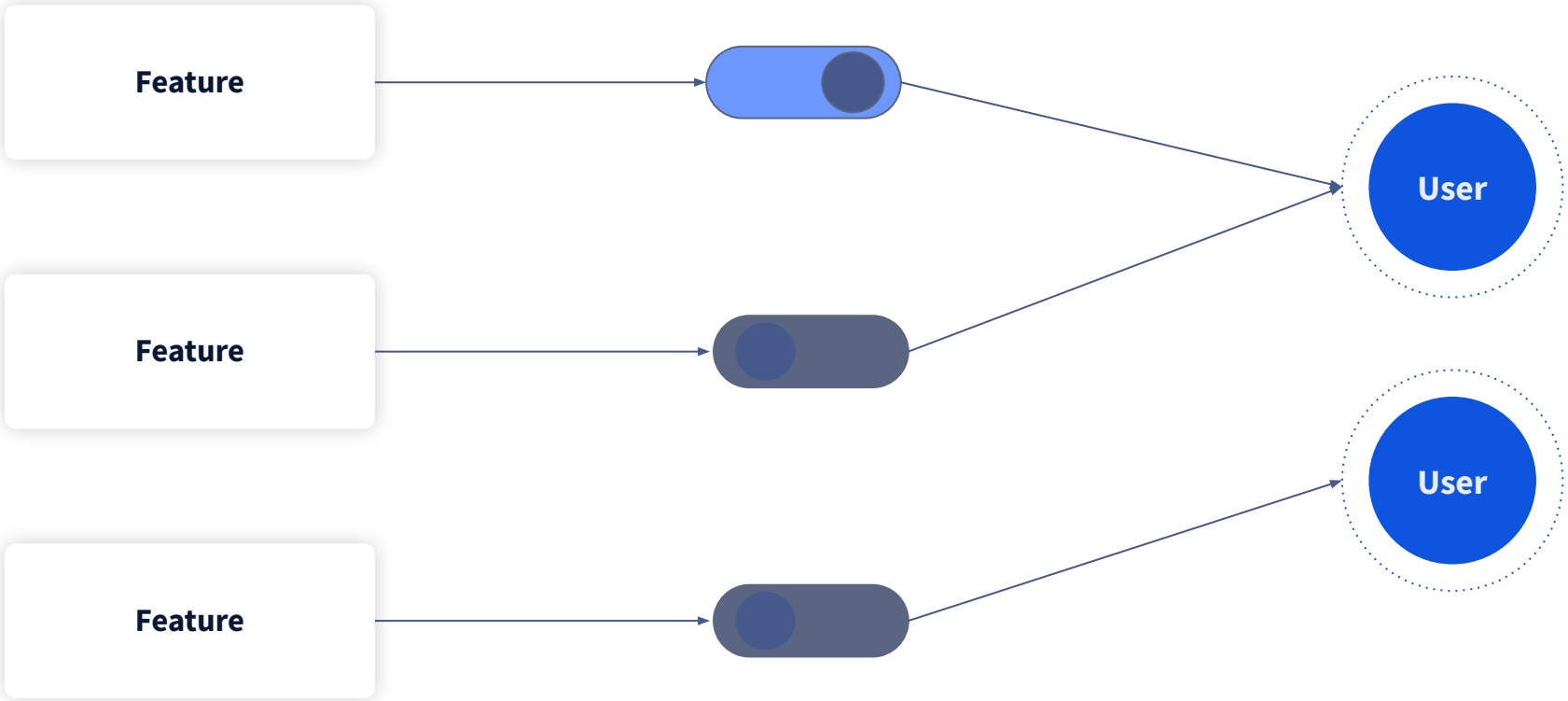
Internal Monitoring

Monitor your repositories
Detect secret leaks
Be alerted in real-time
Integrate remediation to your workflow

Basics

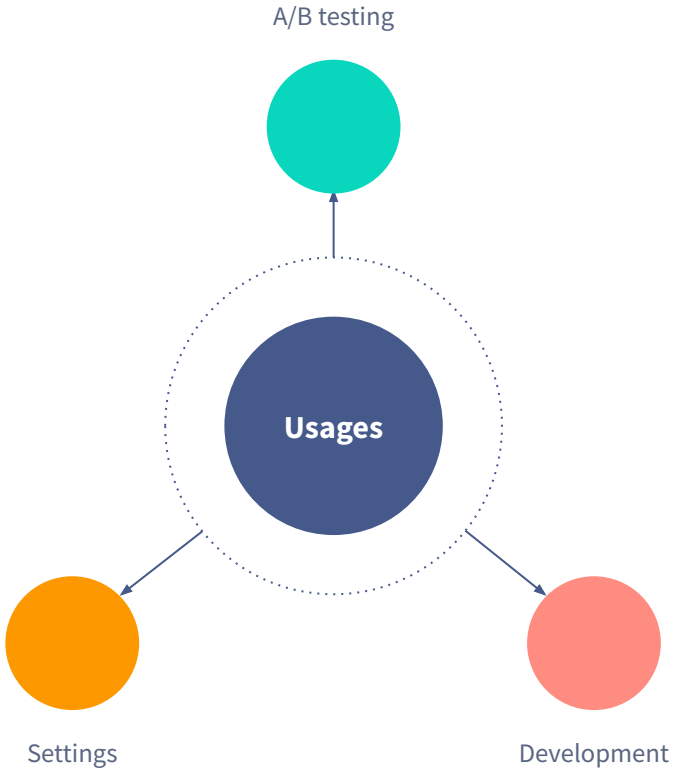
Basics

Feature Flag definition



Basics

Usages



First implementations

First Implementations

Django Solo

Code

```python
# models.py

from django.db import models
from solo.models import SingletonModel


class SiteConfiguration(SingletonModel):
    site_name = models.CharField(max_length=255, default='Site Name')
    maintenance_mode = models.BooleanField(default=False)

    def __str__(self):
        return "Site Configuration"

    class Meta:
        verbose_name = "Site Configuration"
```

On the fly

Settings|preferences can be updated with the admin or an endpoint.

User

Possible but all the logic would need to be coded.

Database

Migrations are needed for every new feature flag.

First Implementations

Poor man example

Code

```python
from django.db import models

class PoorManFeatureFlag(models.Model):
    settings = models.JSONField()
```

On the fly

Settings|preferences can be updated with the admin or an endpoint.

User

Possible but all the logic would need to be coded.

Database

No migrations needed.

First Implementations

Django Dynamic Preferences

Code

```python
# we create some section objects to link related preferences together

general = Section('general')
discussion = Section('discussion')

# We start with a global preference
@global_preferences_registry.register
class SiteTitle(StringPreference):
    section = general
    name = 'title'
    default = 'My site'
    required = False

@global_preferences_registry.register
class MaintenanceMode(BooleanPreference):
    name = 'maintenance_mode'
    default = False

# now we declare a per-user preference
@user_preferences_registry.register
class CommentNotificationsEnabled(BooleanPreference):
    """Do you want to be notified on comment publication ?"""
    section = discussion
    name = 'comment_notifications_enabled'
    default = True
```

2

On the fly

Settings|preferences can be updated with the admin or an endpoint.

User

Can be linked to the users.

Database

No migration needed.

Be Production Ready!

Be production Ready!

Account Preferences

Code

```python
class AccountPreferenceModel(PerInstancePreferenceModel):
    instance = models.ForeignKey(
        Account, on_delete=models.CASCADE, verbose_name="
    )

    class Meta:
        app_label = "app_configuration"
        verbose_name = "Account preference"
```

```python
class AccountPreferenceRegistry(PerInstancePreferenceRegistry)
```

```python
account_preferences_registry = AccountPreferenceRegistry()
```

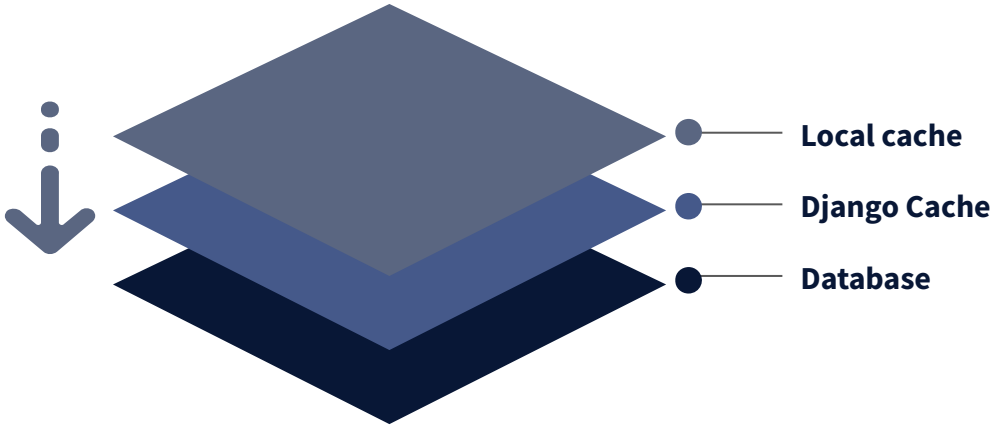
```python
@account_preferences_registry.register
class AllowLegacySSOUrl(AbstractAccountPreference, NullableBooleanPreference):
```

PerInstancePreferenceModel

- Set your preferences per User/Account
- Finer grained preferences

Be production Ready!

Improvements - Requests optimization



2

Faster

Local Cache is 10x faster than Redis.

Requests

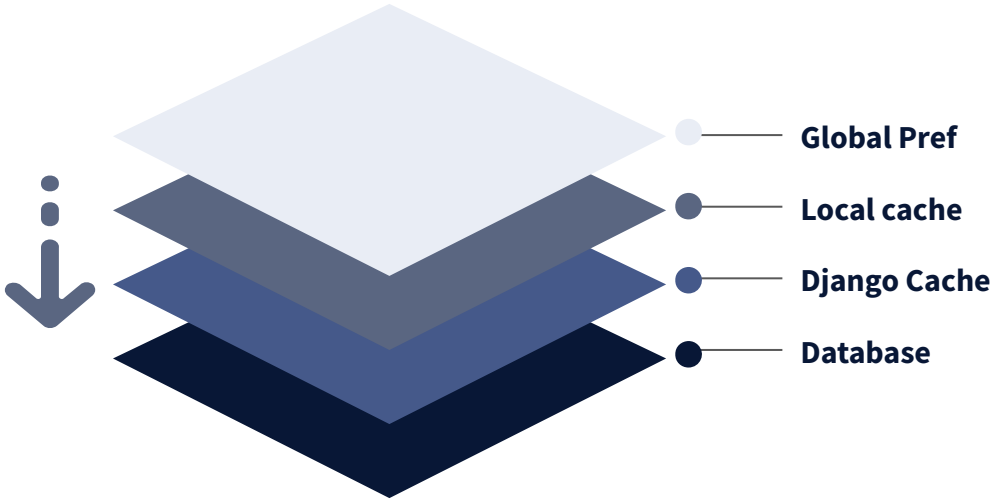
Less network requests.
Less database requests.

Rewrite

Adapted most of the library methods.

Be production Ready!

Improvements - Hierarchy



2

Dependency

Create a hierarchy with depends_on.

Inheritance - Custom property

has_global_preference_fallback.

Rewrite

Adapted most of the library methods.

Be production Ready!

Improvements

Cache

- Faster than DB
- Local Cache

Database Requests

- Adapt manager to decrease requests

Models

- Adapt models for fallback
- Adapt managers for optimization
- Nullable Preferences

Hierarchy

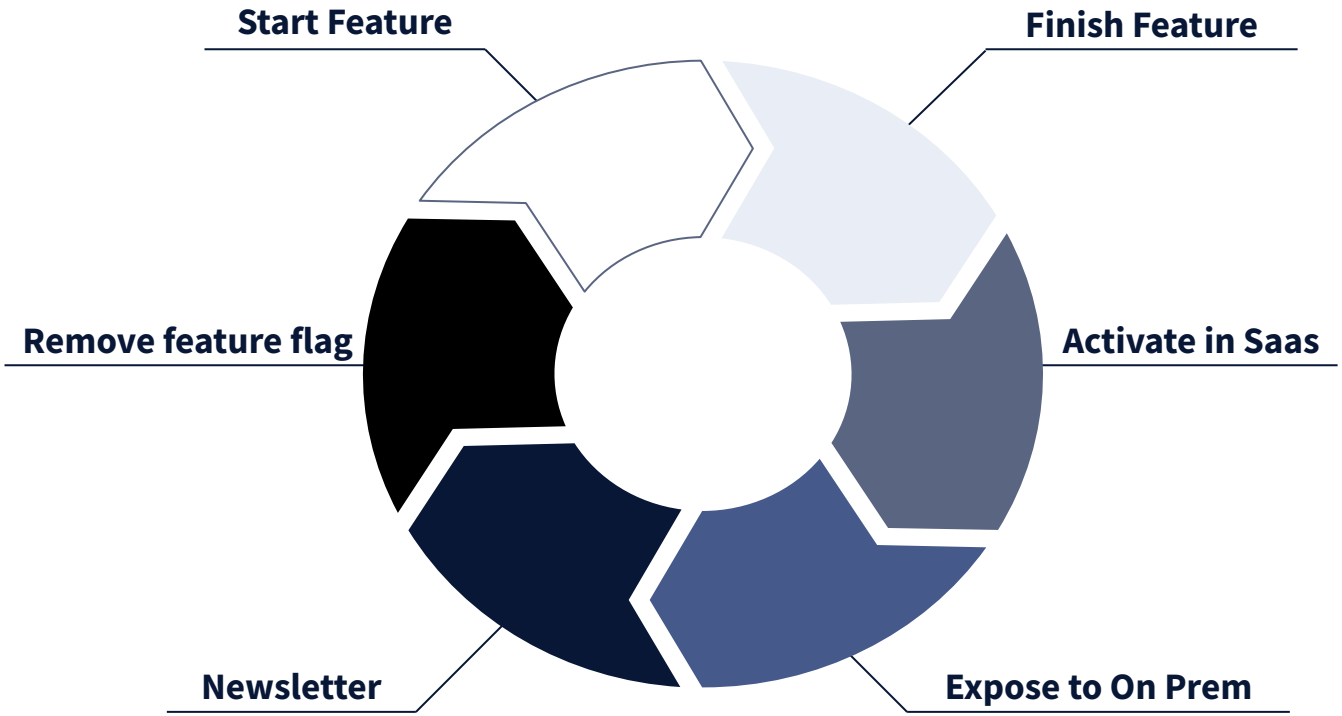
- Dependency between preferences
- Fallback on the global when necessary

Creation

- Create Preference only when necessary
- No overflow of the database

Be production Ready!

Lifecycle



Contact us - We are hiring!

careers@gitguardian.com

Links

https://github.com/lazybird/django-solo/
https://django-dynamic-preferences.readthedocs.io/en/latest/