# Quade Documentation

*Release 0.2.2*

**Paul Baranay**

**Feb 02, 2018**

# Contents


**1 Quickstart** **3**

**2 Installation** **7**

**3 Models** **9**

**4 Settings** **11**
4.1 Available Settings . . . . . 11
4.2 Convenience Classes and Methods . . . . . 12

**5 Using Celery** **13**

**Python Module Index** **15**

Quade is a Django app that makes quality assurance testing easier. It handles the burden of repeatedly setting up the same data, and helps you track the progress of your tests.

Dive in with our *Quickstart*!

# CHAPTER 1

# Quickstart

1. Follow the *installation guide*:
   1. Install Quade with pip:

      ```
      pip install quade
      ```

   2. Add Quade to your project's `INSTALLED_APPS`:

      ```
      INSTALLED_APPS = [
          ...
          'quade.apps.QuadeConfig',
      ]
      ```

   3. Initialize the Quade settings by adding these lines to your Django settings:

      ```
      import quade

      QUADE = quade.Settings()
      ```

      (By default, Quade will only be available when `DEBUG=True`. For more details, see *Settings*.)

   4. Add the django_jinja template handler to your `TEMPLATES` setting, while retaining any other template handlers you have. For example:

      ```
      TEMPLATES = [
          # Existing code, leave in-place
          {
              'BACKEND': 'django.template.backends.django.DjangoTemplates',
              ...
          },
          # New code to add
          {
              'BACKEND': 'django_jinja.backend.Jinja2',
              'DIRS': [],
              'APP_DIRS': True,
      ```

```
        'OPTIONS': {
            'app_dirname': 'jinja2',
        },
    },
]
```

5. Add Quade to your URL configuration, e.g.:

```
# urls.py

urlpatterns = [
    ...
    url(r'^quade/', include('quade.urls'))
]
```

6. Run migrations:

```
python manage.py migrate quade
```

2. Create a fixtures file, `fixtures.py`, in the root of your project:

```
# fixtures.py

import uuid

from django.contrib.auth import get_user_model

from quade.managers import register


User = get_user_model()


@register
def user():
    new_user = User.objects.create(
        first_name='Alyssa',
        last_name='Hacker',
        username='alyssa-{}'.format(uuid.uuid4())
    )
    return "User #{}".format(new_user.pk)
```

3. Update your settings to make Quade aware of your fixtures file:

```
QUADE = quade.settings(fixtures_file='fixtures')
```

4. Enter the Django shell and create a *`Scenario`* that uses this fixture:

```
from quade.models import Scenario
Scenario.objects.create(
    config=[('user', {})],
    description='Single User',
    slug='single-user',
    status=Scenario.Status.ACTIVE
)
```

5. Start your project's webserver and visit the main Quade page. You will be able to generate new users on demand by selecting the "Single User" scenario and executing it.

6. Begin creating your own fixtures and *Scenarios*!

# CHAPTER 2

# Installation

1. Install Quade with pip:

```
pip install quade
```

2. Add Quade to your project's `INSTALLED_APPS`:

```
INSTALLED_APPS = [
    ...
    'quade.apps.QuadeConfig',
]
```

3. Initialize the Quade settings by adding these lines to your Django settings:

```
import quade

QUADE = quade.Settings()
```

   (By default, Quade will only be available when `DEBUG=True`. For more details, see *Settings*.)

4. Add the django_jinja template handler to your `TEMPLATES` setting, while retaining any other template handlers you have. For example:

```
TEMPLATES = [
    # Existing code, leave in-place
    {
        'BACKEND': 'django.template.backends.django.DjangoTemplates',
        ...
    },
    # New code to add
    {
        'BACKEND': 'django_jinja.backend.Jinja2',
        'DIRS': [],
        'APP_DIRS': True,
        'OPTIONS': {
            'app_dirname': 'jinja2',
```

```
        },
    },
]
```

5. Add Quade to your URL configuration, e.g.:

```
# urls.py

urlpatterns = [
    ...
    url(r'^quade/', include('quade.urls'))
]
```

6. Run migrations:

```
python manage.py migrate quade
```

You're ready to roll!

CHAPTER 3

# Models

**class** quade.models.**Scenario**(**args*, ***kwargs*)

A specific scenario that QA tests will be run against.

The heart of a `Scenario` is the *config* attribute. *config* is a list of several fixtures, possibly with arguments, which are executed sequentially to create and modify objects in the database.

**class** quade.models.**Record**(**args*, ***kwargs*)

A record of setting up, and possibly executing, a particular *Scenario*.

**execute_test**()

Public method for attempting to execute a test scenario. Exceptions put the record in the FAILED state and re-raise.

**class** quade.models.**RecordedObject**(**args*, ***kwargs*)

A joiner table for creating a generic many-to-many relation between *Records* and any other objects.

CHAPTER 4

# Settings

Quade includes a Settings class that encapsulates various configuration options.

## 4.1 Available Settings

**class** quade.**Settings**(***kwargs*)

**access_test_func**
A callable that restricts access to Quade's views. The callable should accept one argument (a Django view class).

Default: *is_superuser()*

**allowed_envs**
Controls which environments Quade can run on.

Allowed values are:

- *DebugEnvs*
- *AllEnvs*
- a string – your `django.conf.settings.ENV` is compared exactly to this string
- an iterable – your `django.conf.settings.ENV` is checked exactly for membership in the iterable
- any callable that accepts one argument, the Django settings object

If Quade is not enabled, fixtures will not be loaded, but views will be available to users with appropriate access, application models and tables can be accessed, etc.

Default: *DebugEnvs*

**fixtures_file**
A dotted path to the location of your fixtures.

Default: `quade.fixtures`

**use_celery**

Whether to use Celery to set up a Record.

Enabling this is useful if your fixtures take a long time to run.

See *Using Celery* for additional details.

Default: `False`

## 4.2 Convenience Classes and Methods

**class** `quade.settings.`**AllEnvs**

Enable Quade unconditionally.

**class** `quade.settings.`**DebugEnvs**

Enable Quade if the environment has DEBUG = True.

`quade.settings.`**is_superuser**(*view*)

Allow access to the view if the requesting user is a superuser.

# CHAPTER 5

# Using Celery

By default, Quade will execute `Records` synchronously. If you wish to execute them asynchronously instead, Quade ships with a Celery task for this purpose. To enable it, specify *use_celery=True* in your Quade *setting*.

You must also ensure that Celery is installed and properly configured. You can install Celery as a dependency of Quade by specifying the celery extra during installation:

```
pip install quade[celery]
```

# Python Module Index

## q

quade, 11
quade.models, 9
quade.settings, 12

# Index


## A

access_test_func (quade.Settings attribute), 11
AllEnvs (class in quade.settings), 12
allowed_envs (quade.Settings attribute), 11

## D

DebugEnvs (class in quade.settings), 12

## E

execute_test() (quade.models.Record method), 9

## F

fixtures_file (quade.Settings attribute), 11

## I

is_superuser() (in module quade.settings), 12

## Q

quade (module), 11
quade.models (module), 9
quade.settings (module), 12

## R

Record (class in quade.models), 9
RecordedObject (class in quade.models), 9

## S

Scenario (class in quade.models), 9
Settings (class in quade), 11

## U

use_celery (quade.Settings attribute), 12