Cached Instances for Django REST Framework Documentation

Release 0.3.4

John Whitlock

Aug 14, 2016

Contents

Speed up Django REST Framework (DRF) reads by storing instance data in cache.

This code was split from browsercompat. You may be interested in viewing the browsercompat source code for a full example implementation.

- Code: https://github.com/jwhitlock/drf-cached-instances
- Free software: Mozilla Public License Version 2.0
- Documentation: https://drf-cached-instances.readthedocs.io

CHAPTER 1

How it works

In a normal DRF view, a Django queryset is used to load an object or list of objects. A serializer is used to convert the objects into the "native" representation, and then a renderer works on this native representation. If the serializer includes data from related models, then multiple database queries may be required to generate a native representation. Some database efficiency can be gained by using `select_related`, but a minimum of one query is needed, which is unfortunate for an API with heavy read usage.

This project replaces the Django queryset with a cache-aware proxy class, making it possible to serve a read request with zero database requests (to retrieve an instance) or one request (to get the primary keys for a list view). It is suitable for APIs with heavy read operations and lots of linking between related instances.

When using the cache, Django objects are serialized to JSON. Only the attributes needed for the DRF native representation are stored in the cache. This include the JSON representation of fields such as foreign keys, reverse relations, and dates and times. These serialized objects are stored by primary key in the cache. When an instance is found in the cache, no database reads are needed to render the DRF representation. If the instance is not in the cache, it is serialized and stored, so that future reads will be faster.

The API implementor writes methods to handle JSON serialization, loading from the database, and identifying invalid cache entries on changes. There are a few integration points, including a mixin for views to load data from the cache. With only a few changes to existing code, your read views could be a lot faster.

CHAPTER 2

Project status

This code is used for the browsercompat project, which was developed from 2015 - 2016, but is on hold as of August 2016. Since this was the primary user of this code, it may be a while before more features are implemented.

Contents:

2.1 About this repository

This project was created from the cookiecutter-django-jw template, using a process like this:

```
$ cd ~/src/
$ cookiecutter https://github.com/jwhitlock/cookiecutter-django-jw.git
config_path is ~/.cookiecutterrc
full_name (default is "John Whitlock")? John Whitlock
email (default is "john@factorialfive.com")? john@factorialfive.com
github_username (default is "jwhitlock")?  jwhitlock
repo_name (default is "boilerplate")? drf-cached-instances
project_name (default is "Python Boilerplate")? Cached Instances for Django REST␣
↪Framework
app_name (default is "boilerplate")? drf_cached_instances
site_name (default is "bpsite")? sample_site
project_short_description (default is "Python Boilerplate contains all the␣
↪boilerplate you need to create a Python package.")? Cached instances for Django␣
↪REST Framework
release_date (default is "2015-01-11")? 2014-11-05
year (default is "2014")? 2014
version (default is "0.1.0")? 0.1.0
$ cd drf-cached-instances
$ git init
Initialized empty Git repository in ~/src/drf-cached-instances/.git/
$ git add .
$ git commit -am "Initial commit"
[master (root-commit) 75bc198] Initial commit
 61 files changed, 21366 insertions(+)
 create mode 100644 .coveragerc
 create mode 100644 .editorconfig
 create mode 100644 .gitignore
 ...
 create mode 100644 setup.cfg
 create mode 100755 setup.py
 create mode 100644 tox.ini
$ mkvirtualenv drf-cached-instances
```

```
New python executable in drf-cached-instances/bin/python2.7
Also creating executable in drf-cached-instances/bin/python
Installing setuptools, pip...done.
$ pip install -r requirements.txt
 (development requirements installed)
$ ./manage.py migrate
```

2.1.1 Development Features

After a basic install, you can run:

- `make qa` - runs flake8 for PEP8 and PEP257 compliance checking. Runs coverage to confirm 100% code coverage for the app.
- `make install_jslint` - Install node and jslint in the virtualenv. Future runs of `make qa` will include jslint checking of project javascript.
- `make qa-all` - All the checks of `make qa`, plus building documentation, building and checking packaging, and running tox against a range of Python Django versions.
- `make` - See other make targets.
- `./manage.py test` - Run tests with the nose test runner
- `./manage.py test --failed --ipdb --ipdb-failures` - Run tests. Keep track of which tests failed, and only run those in the future. When an issue appears, debug in an interactive ipdb session.
- `DEBUG=1 ./manage.py runserver_plus` - Run in debug mode. Includes Django Debug Toolbar for peeking behind the scenes, and interactive tracebacks on failures.

2.1.2 Deployment Features

sample_site/settings.py is built using the 12factor principle of getting configuration from the environment. The default configuration is release mode. You'll need to set environment variables to match your desired configuration.

In development, some ways to set the environment are:

- On the command line: `DEBUG=1 ./manage.py runserver`
- Export settings: `export DEBUG=1; ./manage.py runserver`
- As part of virtualenv initialization: `vim $VIRTUAL_ENV/bin/postactivate` for the `export DEBUG=1` set statements, and `vim $VIRTUAL_ENV/bin/predeactivate` for the `export DEBUG=` clear statements.

Heroku deployment is included. This can be done with the 'Deploy to Heroku' button, or manually:

```
$ heroku apps:create drf-cached-instances
```

Then config for development:

```
$ heroku config:set EXTRA_INSTALLED_APPS=gunicorn STATIC_ROOT=static DEBUG=1
```

Or for production:

```
$ heroku config:set EXTRA_INSTALLED_APPS=gunicorn STATIC_ROOT=static DEBUG=0 ALLOWED_
↪HOSTS=drf-cached-instances.herokuapp.com SECURE_PROXY_SSL_HEADER=HTTP_X_FORWARDED_
↪PROTOCOL,https
```

When you've got the app configured, deploy your code to run it:

```
$ git push heroku master
$ heroku open
```

2.2 Installation

At the command line:

```
$ easy_install drf-cached-instances
```

Or, if you have virtualenvwrapper installed:

```
$ mkvirtualenv drf-cached-instances
$ pip install drf-cached-instances
```

2.3 Usage

drf-cached-instances is designed to work with Django REST Framework (DRF), using a cache for read-only operations such as getting an instance or list of instances. You may also want Celery for asynchronously updating the cache.

There are a few steps needed to integrate drf-cached-instances into your project. See the sample app `sample_poll_app` for a small example, or browsercompat for a fuller example.

2.3.1 Create an app-specific cache strategy

drf-cached-instances requires that you specify how a model is cached, by adding methods to the Cache class. Each model requires three functions:

1. A serializer, which turns a Django instance into a JSON-serializable dictionary,
2. A loader, which loads a Django instance and related objects from the database, and
3. An invalidator, which specifies which instance caches are possibly invalid when an instance is updated.

The naming convention for these functions are `{model}_{version}_{function}`. For example, the serializer for the User model for the 'v1' API would be `user_v1_serializer`. API/cache versioning is option, and the default version name is 'default'.

Here's an example of a customized Cache:

```
from django.contrib.auth.models import User
from drf_cached_instances.cache import BaseCache

class MyCache(BaseCache):

    """Cache for my application."""

    def user_default_serializer(self, obj):
        """Convert a User to a cached instance representation."""
        if not obj:
            return None
        self.user_default_add_related_pks(obj)
        return dict((
```

```
            ('id', obj.id),
            ('username', obj.username),
            self.field_to_json('DateTime', 'date_joined', obj.date_joined),
        ))

    def user_default_loader(self, pk):
        """Load a User from the database."""
        try:
            obj = User.objects.get(pk=pk)
        except User.DoesNotExist:
            return None
        else:
            self.user_default_add_related_pks(obj)
            return obj

    def user_default_add_related_pks(self, obj):
        """Add related primary keys to a User instance."""
        if not hasattr(obj, '_votes_pks'):
            obj._votes_pks = list(obj.votes.values_list('pk', flat=True))

    def user_default_invalidator(self, obj):
        """Invalidate cached items when the User changes."""
        return []
```

2.3.2 Use the cache in views

If you are using viewsets, add the *CachedViewMixin* to your viewset declarations:

```
from django.contrib.auth.models import User
from drf_cached_instances.mixins import CachedViewMixin
from rest_framework.viewsets import ModelViewSet
from rest_framework.serializers import DateField, ModelSerializer

class UserSerializer(ModelSerializer):

    """DRF serializer for Users."""

    created = DateField(source='date_joined', read_only=True)

    class Meta:
        model = User
        fields = ('id', 'username', 'created')

class UserViewSet(CachedViewMixin, ModelViewSet):

    """API endpoint that allows users to be viewed or edited."""

    queryset = User.objects.all()
    serializer_class = UserSerializer
```

2.3.3 Add signal hooks to update the cache

When an instance is updated, the cache is invalid and needs to be updated. This can be done by adding signal hooks for model modifications in models.py:

```
from django.contrib.auth.models import User
from django.db.models.signals import post_delete, post_save, m2m_changed
from django.dispatch import receiver
from .cache import MyCache

def update_cache_for_instance(model_name, instance_pk, instance):
    cache = MyCache()
    version = cache.default_version
    to_update = cache.update_instance(
        model_name, instance_pk, instance, version)
    for related_name, related_pk, related_version in to_update:
        update_cache_for_instance(
            related_name, related_pk, version=related_version)

@receiver(post_delete, sender=User, dispatch_uid='post_delete_update_cache')
def post_delete_user_update_cache(sender, instance, **kwargs):
    update_cache_for_instance('User', instance.pk, instance)

@receiver(post_save, sender=User, dispatch_uid='post_save_update_cache')
def post_save_user_update_cache(sender, instance, created, raw, **kwargs):
    if raw:
        return
    update_cache_for_instance('User', instance.pk, instance)
```

This will follow the invalidation logic in the Cache class, to ensure that the cache is consistant across related instances.

2.3.4 Handling cascading cache updates

The `update_cache_for_instance` method uses recursion to ensure the cache is consistant. By default, this populates missing cache entries as well. For highly related instances, this would result in loading a lot of the database into a cold cache, making the first update very slow.

There are a few ways to handle the cold cache problem. The first is to use a asynchronous task system like Celery for updates. This way, updates can return quickly while backend processes warm the cache.

Another method is to use `update_only=True` when calling `cache.update_instance`. This will stop the invalidation chain on cache misses, which may result in an inconsistent cache for cached instances that are a few steps away from the updates instance. Eventual consistency can be maintained by automatically expiring cache entries.

You may want to configure `update_only=True` in development for speed, and use the default `update_only=False` in production.

2.4 Contributing

Contributions are welcome, and they are greatly appreciated! Every little bit helps, and credit will always be given.

You can contribute in many ways:

2.4.1 Types of Contributions

Report Bugs

Report bugs at https://github.com/jwhitlock/drf-cached-instances/issues.

If you are reporting a bug, please include:

- Your operating system name and version.
- Any details about your local setup that might be helpful in troubleshooting.
- Detailed steps to reproduce the bug.

Fix Bugs

Look through the GitHub issues for bugs. Anything tagged with "bug" is open to whoever wants to implement it.

Implement Features

Look through the GitHub issues for features. Anything tagged with "feature" is open to whoever wants to implement it.

Write Documentation

drf-cached-instances could always use more documentation, whether as part of the official Cached Instances for Django REST Framework docs, in docstrings, or even on the web in blog posts, articles, and such.

Submit Feedback

The best way to send feedback is to file an issue at https://github.com/jwhitlock/drf-cached-instances/issues.

If you are proposing a feature:

- Explain in detail how it would work.
- Keep the scope as narrow as possible, to make it easier to implement.
- Remember that this is a volunteer-driven project, and that contributions are welcome :)

2.4.2 Get Started!

Ready to contribute? Here's how to set up drf-cached-instances for local development.

1. Fork the drf-cached-instances repo on GitHub.
2. Clone your fork locally:

```
$ git clone git@github.com:your_name_here/drf-cached-instances.git
```

3. Install your local copy into a virtualenv. Assuming you have virtualenvwrapper installed, this is how you set up your fork for local development:

```
$ mkvirtualenv drf-cached-instances
$ cd drf-cached-instances/
$ pip install -r requirements.txt
$ ./manage.py syncdb
```

4. Create a branch for local development:

```
$ git checkout -b name-of-your-bugfix-or-feature
```

Now you can make your changes locally.

5. When you're done making changes, check that your changes pass flake8 and the tests, including testing other Python versions with tox:

```
$ make qa-all
```

To get flake8 and tox, just pip install them into your virtualenv.

6. Commit your changes and push your branch to GitHub:

```
$ git add .
$ git commit -m "Your detailed description of your changes."
$ git push origin name-of-your-bugfix-or-feature
```

7. Submit a pull request through the GitHub website.

2.4.3 Pull Request Guidelines

Before you submit a pull request, check that it meets these guidelines:

1. The pull request should include tests. Test coverage should be 100%, line and branch.
2. Follow PEP8 and PEP257. `make qa` can be used to check compliance.
3. If the pull request adds functionality, the docs should be updated. Put your new functionality into a function with a docstring, and add the feature to the list in README.rst.
4. The pull request should work for Python 2.6, 2.7, 3.3, and 3.4. Check https://travis-ci.org/jwhitlock/drf-cached-instances/pull_requests and make sure that the tests pass for all supported Python versions. Use `make qa-all` to check locally.

2.4.4 Tips

To run a subset of tests:

```
$ ./manage.py test tests/test_cache.py
```

To mark failed tests:

```
$ ./manage.py test --failed
```

To re-run only the failed tests:

```
$ ./manage.py test --failed
```

2.5 Credits

2.5.1 Development Lead

- John Whitlock <john@factorialfive.com>

2.5.2 Contributors

- fpruitt
- creynold

2.6 History

2.6.1 0.3.4 (2016-08-14)

- Drop support for Django 1.7, Python 2.6
- Expand tests to Django 1.10, Django REST Framework 3.4, Python 3.5

2.6.2 0.3.3 (2015-11-05)

- Add serializer for datetime.timedelta, to support Django 1.8's DurationField (creynold)
- Convert string representations of datetimes, dates, and timedeltas
- Expand tests to Django 1.9 beta 1 and Django REST Framework 3.3

2.6.3 0.3.2 (2015-09-23)

- Remove deprecation warning in Django 1.8
- Expand tests to Django master and Django REST Framework 3.2
- Fix invalid mock.patch tests that break under mock 1.3.0
- Documentation updates and fixes

2.6.4 0.3.1 (2015-06-16)

- Move get_object_or_404 to mixin method, to allow easier extending.

2.6.5 0.3.0 (2015-04-09)

- Tested with Django 1.8
- Tested with Django REST Framework 2.4, 3.0, and 3.1
- CachedModel now supports .pk attribute as an alias, usually to the .id field. DRF 3 uses .pk to determine if a model is saved to database, and returns empty relation data for unsaved fields.
- cache.delete_all_versions() will delete all cached instances of a model and PK. This is useful when changes are made outside of normal requests, such as during a data migration.

2.6.6 0.2.0 (2014-12-11)

- Add `update_only` option to `cache.update_instance`, to support eventual consistency for cold caches.

2.6.7 0.1.0 (2014-11-06)

- First release on PyPI.

2.7 Indices and tables

- genindex
- modindex
- search