Switchboard Documentation

Release 1.2.1

Kyle Adams

Apr 12, 2018

Contents

Switchboard is a feature flipper library for the Pyramid or Pylons stacks (including TurboGears). Originally used to selectively roll out changes to the SourceForge site, the library lets you easily control whether a particular change (a switch) is active.

You can make switches active for a certain percentage of visitors, all visitors to a particular host in a cluster, or if a particular string is present in the query string. Furthermore you can easily create your own conditions to do fancier things like geo-targeting, specific users, etc.

Having a feature flipper allows features to be inserted without creating long-running feature branches. Not having to deal with merge headaches and messy conflicts means your continuous integration builds run smoother. Continuous deployment is now an option because dangerous code can be hidden behind a switch. In short, Switchboard turns you into a continuous delivery ninja.

CHAPTER 1

Documentation

1.1 Upgrading

Upgrading from Switchboard 1.2.x or earlier will require a few changes. Switchboard is now an embeddable WSGI app, which should make integrating it into apps easier for new users, but existing users will need to change (simplify) their approach. An overview of what's changed:

- The *get_user* and *get_request* functions have been removed. Inject objects into the context by extending Switchboard's middleware. Note that using *switchboard.middleware.SwitchboardMiddleware* injects the request into the context automatically.
- The *switchboard.admin.controllers* module has been removed; any code wrapping the *CoreAdminController* class can be removed.
- Any code implementing routing for Switchboard can be removed.
- Post-request cleanup is now handled by *switchboard.middleware.SwitchboardMiddleware*; any custom middleware can either be simplified or removed entirely.

Please see *Other Frameworks* for details on the new approach for integrating Switchboard into another application.

1.2 Installation

Install Switchboard and its dependencies using `pip`:

```
pip install switchboard
```

Next, embed Switchboard and its admin UI within the application. The best approach depends on which application framework is being used.

1.2.1 Pyramid

Switchboard has a pyramid add-on available to make pyramid setup easier:

```
pip install pyramid_switchboard
```

Once the dependency is in place, there are several ways to make sure that `pyramid_switchboard` is active and Switchboard is up and running. They are all equivalent.

1. Add `pyramid_switchboard` to the `pyramid.includes` section of the application's main configuration section:

   ```
   [app:main]
   ...
   pyramid.includes = pyramid_switchboard
   ```

2. Use the `includeme` function via *config.include*:

   ```
   config.include('pyramid_switchboard')
   ```

3. Optionally setup Switchboad for easy use *in templates*.

Once activated, Switchboard's admin UI is accessible at `/_switchboard/` and switches can now be used in the code.

1.2.2 Other Frameworks

Switchboard is compatible with any application framework that uses WebOb as the underlying request/response library. Even if a plugin/add-on doesn't exist, Switchboard can still be setup manually.

Configuration

The first step is to configure Switchboard in the application's config file. Switchboard has only a handful of settings, none of which are required:

Key	Default	Description
switchboard.auto_create	True	Auto-creation of non-existent switches.
switchboard.internal_ips		Comma-delimited list of IPs.

Note that the "switchboard" prefix for the setting keys is also optional. Additionally, Switchboard will need a configured Datastore object.

Initializing

In the application's bootstrap or initialization code, pass the settings into Switchboard's `configure` method:

```
from switchboard import configure

configure(settings, datastore, nested=True)
```

If the setting keys are *not* prefixed with "switchboard" the `nested=True` argument can be omitted.

An example configuration that needs `nested=True`:

```
switchboard.internal_ips=192.168.1.11
```

And one that does not need `nested=True`:

```
internal_ips=192.168.1.11
```

The *datastore* parameter is discussed in *Persisting Data*.

Persisting Data

Switchboard defaults to an in-memory datastore, which means that data is not persistent. To setup a persistent datastore, construct a Datastore object. The details of how that object is built vary depending on what backend is choosen.

First ensure that the necessary libraries are installed in the virtualenv and added to the application's requirements. Those libraries typically consist of one of Datastore's subprojects, along with the choosen backend's Python client.

Once the libraries are in place, initialize the appropriate `Datastore` object and then pass it into `configure`. An example of connecting to Mongo (the backend before Switchboard's 2.0 release):

```
import pymongo
import datastore.mongo
from switchboard import configure

conn = pymongo.Connection()
ds = datastore.mongo.MongoDatastore(conn.switchboard)
configure(settings, ds)
```

An example connecting to Redis with pickle serialization:

```
import redis
import datastore.redis
from switchboard import configure

r = redis.Redis()
ds = datastore.redis.RedisDatastore(r, serializer=pickle)
configure(settings, ds)
```

The Admin UI

The admin UI is a standalone WSGI application; as such it can be embedded as a subapplication within a larger application. See specific documentation for Bottle subapplications, Django embedding, or dispatch middleware for any WSGI application.

> **Warning:** Secure Switchboard
>
> Please configure this subapp so that only admins can access it. Switchboard is a powerful tool and should be adequately secured.

Middleware

The last thing to setup is to handle pre- and post-request tasks. Pre-request tasks can include adding objects to the context (eliminating the need to add them explicitly when querying `is_active`). Post-request tasks include cleaning up caching data once a request is finished. Switchboard includes middleware to handle these tasks. Using it out of the box:

```
from switchboard.middleware import SwitchboardMiddleware
app = SwitchboardMiddleware(app)
```

It can also be extended for further customization, specifically by implementing the `pre_request` method. For example, to add a user object to the context:

```
from switchboard.middleware import SwitchboardMiddleware


class MyMiddleware(SwitchboardMiddleware):

    def pre_request(self, req):
        user = req['user']
        operator.context['user'] = user

    def post_request(self, req, resp):
        pass  # Included just to show what's available.
```

Caching

In some high-volume applications, switch data may need to be cached to maintain high performance. Switchboard supports a cache system, e.g. memcached, via Datastore's `TieredDatastore`:

```
import pylibmc
import pymongo
import datastore.core
import datastore.memcached
import datastore.mongo
from switchboard import configure

mc = pylibmc.Client(['127.0.0.1'])
cache = datastore.memcached.MemcachedDatastore(mc)

conn = pymongo.Connection()
mongo = datastore.mongo.MongoDatastore(conn.switchboard)

ds = datastore.TieredDatastore([cache, mongo])

configure(settings, ds)
```

It is also possible to cache results of `is_active` calls. This speeds up switchboard when the same switches are called multiple times, or when multiple child switches are used (so the parent will only be checked once). The application is required to clear the cache, e.g. for each web request. To enable `is_active` result caching do:

```
operator.result_cache = {}
```

It is recommended to do that in the `pre_request` method of your switchboard *middleware* so that it is reset for each request.

1.3 An Example

Switchboard includes an example application, which is handy both for doing Switchboard development and for playing around with switches and the admin UI in a very simple environment. It also provides a look at a working example of the setup instructions above.

Before running the example application, setup and activate a virtual environment.

To run the example application for the first time: `make install example`. On subsequent runs `make example` will suffice.

At this point a very simple application is now running at `http://localhost:8080` and the admin UI is accessible at `http://localhost:8080/_switchboard/`. The application has one switch (`example`) and outputs text that tells you whether the switch is active.

1.4 Using Switches

By default, Switchboard is set to autocreate switches, which means that a switch just needs to be checked in code and if it doesn't exist it will be created and disabled by default. A switch is always referred to by its key, a string identifier that should be unique.

1.4.1 A Word on Workflow

The developer can choose whether to take advantage of autocreate or not. There are two basic workflows. The first, which uses autocreate, is this:

1. Write the code first. Reference the switch in the code.
2. Test the application in such a way that the code containing the switch is exercised.
3. Refresh the Switchboard admin UI to see the new switch. Modify it as needed.
4. If necessary, re-test the application with the proper switch status and/or condition sets.

The primary advantage of this approach is that there is no chance that the switch key used in the code will differ from the one in Switchboard, e.g., due to a typo. It can also be advantageous, from the perspective of flow, to delay having to exit the code editor until a later time. The disadvantage is having to exercise code twice: once to create the switch and then again to test switch behavior.

Eschewing autocreate:

1. Create the switch in the admin UI. Modify it as needed.
2. Write the code, making sure to use the key of the newly-created switch.
3. Test the application.

This approach minimizes time spent putting the application through its paces, but at the expense of switching between the web browser and the code editor.

Use whatever works.

1.4.2 In Python

To use in Python (views, models, etc.), import the operator singleton and use the `is_active` method to see if the switch is on or not:

```
from switchboard import operator

if operator.is_active('foo'):
    ... do something ...
else:
    ... do something else ...
```

If autocreate is on (and it is by default), the `foo` switch will be automatically created and set to disabled the first time it is referenced. Activating the switch and controlling exactly when the switch is active, are covered in *Managing switches*.

1.4.3 In Views

Switchboard has a convenience decorator for when you want to enable/disable an entire view based on a switch:

```
from switchboard.decorators import switch_is_active

@switch_is_active('admin_user', redirect_to='/login')
def admin_view():
    # Admin stuff happens here.
    return
```

If the `redirect_to` argument is not set and the switch is not active, the client will get a 404 error.

1.4.4 In Templates

Every templating engine has its own take on how (or even if) logic may be used. That said, Switchboard provides a helper to make things easier: `switchboard.template_helpers.is_active`. This function is just a wrapper around `operator.is_active` to make it easier to check a switch. Here are examples in some of the common Python templating engines.

In Jinja, the helper can be setup as a test and used like so:

```
{% if 'foo' is active %}
... do something ...
{% else %}
... do something else ...
{% endif %}
```

Check the application framework's documentation for information on how to setup custom Jinja tests.

In Mako, the helper can be imported directly:

```
<%!
    from switchboard.template_helpers import is_active
%>
...
% if is_active('foo'):
... do something ...
% else:
... do something else ...
% endif
```

1.4.5 In Javascript

The easiest way to use Switchboard in conjunction with Javascript is to set a flag within the template code. Using Mako's syntax in the template:

```
<%!
    from switchboard import operator
%>
```

```
<script>
    window.switches = window.switches || {};
    % if operator.is_active('foo'):
    switches.foo = true;
    % else:
    switches.foo = false;
    % endif
</script>
```

In the Javascript:

```
if (switches.foo) {
    ... do something ...
} else {
    ... do something else ...
}
```

Again, this time using Jinja syntax and the Switchboard-provided "active" test:

```
<script>
    window.switches = {};
    switches.foo = {{ 'true' if 'foo' is active else 'false' }};
</script>
```

1.4.6 Custom Conditions

Switchboard supports custom conditions, allowing application developers to adapt switches to their particular needs. Creating a condition typically consists of extending `switchboard.conditions.ConditionSet`.

An example: if the application needs to activate switches for visitors from a particular country, a custom condition can do the geo lookup on the IP from the request and return the country value:

```
from switchboard.conditions import ConditionSet, Regex
from my_app.geo import country_code_by_addr, client_ip

class GeoConditionSet(ConditionSet):
    countries = Regex()

    def get_namespace(self):
        ''' Namespaces are unique identifiers for each condition set. '''
        return 'geo'

    def get_field_value(self, instance, field_name):
        ''' Should return the expected value for any given field. '''
        if field_name == 'countries':
            return country_code_by_addr(client_ip())

    def get_group_label(self):
        ''' A human-friendly label used in the UI. '''
        return 'Geo'
```

The first thing in the custom condition is to define the fields that makeup the condition. In this case, there is one "countries" field, which is a regex, allowing admins to specify criteria like `(US|CA)` (US or Canada). Here are the fields supported by Switchboard:

- `switchboard.conditions.Boolean` - used for binary, on/off fields

- `switchboard.conditions.Choice` - used for multiple choice dropdowns
- `switchboard.conditions.Range` - used for numeric ranges
- `switchboard.conditions.Percent` - a special type of range specific to percentages
- `switchboard.conditions.String` - string matching
- `switchboard.conditions.Regex` - regex expression matching
- `switchboard.conditions.BeforeDate` - before a date
- `switchboard.conditions.OnOrAfterDate` - on or after a date

Once the fields are defined, there are some methods that need to be implemented. `get_namespace` and `get_group_label` are simple functions that return a key and a UI string respectively. Most of the work happens in the `get_field_value` function, which is responsbile for returning the value that is compared against the user-provided input. Each field type may do the comparison (between the user-provided input and what's returned by `get_field_value`) in a different way; in this case, it's a regex search.

When an admin sets up a Geo condition set and sets the countries field to "US|CA", that input is compared against the country code returned by `get_field_value`. If they match, then the switch passes that particular condition.

1.4.7 Context Objects

Every switch is evaluated (to see if it is active or not) within a particular context. By default, that context includes the request object, which allows Switchboard to specify conditions such as: "make this switch active only for requests with `foo` in the query string." That said, there may be other objects that would be handy to have available in the context. For example, in an e-commerce setting, the Product model may have a `new` flag. By passing the model into the `is_active` method, Switchboard can now activate switches based on that flag:

```
if operator.is_active('foo', my_product):
```

Any objects passed into the `is_active` method after the switch's key will be added to the context. Normally when dealing with context objects, a custom condition will be required to actually evaluate the switch against that object.

1.5 Testing switches

Switchboard provides a decorator that makes it easy to turn a switch on or off for a particular unit test:

```
from switchboard import operator
from switchboard.testutils import switches

@switches(my_switch=True)
def test_my_switch:
    assert operator.is_active('my_switch')
```

1.6 Managing switches

Switches are managed in the admin UI, which is located at the `SWITCHBOARD_ROOT` within the application. The admin UI allows:

- Viewing and searching all switches.
- Reviewing or auditing a switch's history.

- Adding, editing, and removing switches.
- Controlling a switch's status.
- Setting up condition sets for a switch.

Of all these capabilities, the last two are of the most interest, as the status and condition sets determine whether a switch is active.

1.6.1 Statuses

There are four statuses:

- Inactive - disabled for everyone
- Selective - active only for matched conditions
- Inherit - inherit from the parent switch
- Global - active for everyone

Inactive and global are opposite extremes: the switch is turned on or off for everyone. The inherit status is used for *Parent-child switches*. The selective status means that the switch is only active if it passes the condition sets.

By default, a switch will be created and set to the inactive status. Typical workflow would be to put code using a switch into production. The corresponding switch will be autocreated the first time the code containing it is executed, thus visible in the admin UI. Once visible, the admin can set any desired conditions before finally activating the switch by setting it to the proper status.

1.6.2 Condition Sets

When a switch is in selective status, Switchboard checks the conditions within the condition set to see if the switch should be active. Conditions are criteria such as "10% of all visitors" or "only logged in users" that can be applied to the request to see if the switch should be active. When a switch is in selective status, it will only be active if it meets the conditions in place.

1.6.3 Parent-child switches

Switchboard allows a switch to inherit conditions from a parent, which can be useful when multiple switches need to share a common condition set. To setup parent-child relationship, simply prefix the switch with the parent's key, using a colon ':' as the separator. The parent-child relationships can be as deep as needed, e.g., `grandparent:parent:child`.

A real world example: using Switchboard to conduct an AB test. AB tests have two gates: the first are the visitors who are part of the test, and the second is to determine who sees which variant. In this example, 10% of site traffic should be in the test, with half (i.e., 5% of traffic) seeing the normal (control) A variant and the other half seeing the B variant. The test is setup with two switches:

- abtest
- abtest:B

The `abtest` switch has a "0-10% of traffic" condition set. The `abtest:B` switch will inherit from `abtest` and can add its own "0-5% of traffic" condition. Half of those in the test will see the B variant, the rest will see the control A variant. The `abtest:B` switch's status should be set to selective, for reasons noted below.

Note that an additional tool, like Google Analytics Content Experiments, is still needed to measure conversion within each variant, but Switchboard can handle traffic segmentation.

Two potential spots of confusion:

1. Child switches *always* inherit from their parents, even when the child switch's status is set to something other than inherit. An inherit status just means the child switch isn't adding to the parent switch's status.
2. It is also important to note that when a parent switch is disabled, it takes precedence over the statuses of any child switches. On the other hand, if the parent switch is enabled, it can be overriden by the child switch, e.g., if the parent has a global status but the child has an inactive status, the child's inactive wins out.

CHAPTER 2

Indices and tables

- genindex
- modindex
- search