Signals

▶ *Changelog*

Starting with Flask 0.6, there is integrated support for signalling in Flask. This support is provided by the excellent blinker library and will gracefully fall back if it is not available.

What are signals? Signals help you decouple applications by sending notifications when actions occur elsewhere in the core framework or another Flask extensions. In short, signals allow certain senders to notify subscribers that something happened.

Flask comes with a couple of signals and other extensions might provide more. Also keep in mind that signals are intended to notify subscribers and should not encourage subscribers to modify data. You will notice that there are signals that appear to do the same thing like some of the builtin decorators do (eg: **`request_started`** is very similar to **`before_request()`**). However, there are differences in how they work. The core **`before_request()`** handler, for example, is executed in a specific order and is able to abort the request early by returning a response. In contrast all signal handlers are executed in undefined order and do not modify any data.

The big advantage of signals over handlers is that you can safely subscribe to them for just a split second. These temporary subscriptions are helpful for unit testing for example. Say you want to know what templates were rendered as part of a request: signals allow you to do exactly that.

Subscribing to Signals

To subscribe to a signal, you can use the **`connect()`** method of a signal. The first argument is the function that should be called when the signal is emitted, the optional second argument specifies a sender. To unsubscribe from a signal, you can use the **`disconnect()`** method.

For all core Flask signals, the sender is the application that issued the signal. When you subscribe to a signal, be sure to also provide a sender unless you really want to listen for signals from all applications. This is especially true if you are developing an extension.

For example, here is a helper context manager that can be used in a unit test to determine which templates were rendered and what variables were passed to the template:

```
from flask import template_rendered
from contextlib import contextmanager

@contextmanager
```

```
def captured_templates(app):
    recorded = []
    def record(sender, template, context, **extra):
        recorded.append((template, context))
    template_rendered.connect(record, app)
    try:
        yield recorded
    finally:
        template_rendered.disconnect(record, app)
```

This can now easily be paired with a test client:

```
with captured_templates(app) as templates:
    rv = app.test_client().get('/')
    assert rv.status_code == 200
    assert len(templates) == 1
    template, context = templates[0]
    assert template.name == 'index.html'
    assert len(context['items']) == 10
```

Make sure to subscribe with an extra `**extra` argument so that your calls don't fail if Flask introduces new arguments to the signals.

All the template rendering in the code issued by the application *app* in the body of the `with` block will now be recorded in the *templates* variable. Whenever a template is rendered, the template object as well as context are appended to it.

Additionally there is a convenient helper method (**`connected_to()`**) that allows you to temporarily subscribe a function to a signal with a context manager on its own. Because the return value of the context manager cannot be specified that way, you have to pass the list in as an argument:

```
from flask import template_rendered

def captured_templates(app, recorded, **extra):
    def record(sender, template, context):
        recorded.append((template, context))
    return template_rendered.connected_to(record, app)
```

The example above would then look like this:

```
templates = []
with captured_templates(app, templates, **extra):
    ...
    template, context = templates[0]
```

Blinker API Changes:

The `connected_to()` method arrived in Blinker with version 1.1.

Creating Signals

If you want to use signals in your own application, you can use the blinker library directly. The most common use case are named signals in a custom `Namespace`.. This is what is recommended most of the time:

```
from blinker import Namespace
my_signals = Namespace()
```

Now you can create new signals like this:

```
model_saved = my_signals.signal('model-saved')
```

The name for the signal here makes it unique and also simplifies debugging. You can access the name of the signal with the `name` attribute.

For Extension Developers:

If you are writing a Flask extension and you want to gracefully degrade for missing blinker installations, you can do so by using the `flask.signals.Namespace` class.

Sending Signals

If you want to emit a signal, you can do so by calling the `send()` method. It accepts a sender as first argument and optionally some keyword arguments that are forwarded to the signal subscribers:

```
class Model(object):
    ...

    def save(self):
        model_saved.send(self)
```

Try to always pick a good sender. If you have a class that is emitting a signal, pass `self` as sender. If you are emitting a signal from a random function, you can pass `current_app._get_current_object()` as sender.

Passing Proxies as Senders:

Never pass **`current_app`** as sender to a signal. Use `current_app._get_current_object()` instead. The reason for this is that **`current_app`** is a proxy and not the real application object.

Signals and Flask's Request Context

Signals fully support The Request Context when receiving signals. Context-local variables are consistently available between **`request_started`** and **`request_finished`**, so you can rely on **`flask.g`** and others as needed. Note the limitations described in Sending Signals and the **`request_tearing_down`** signal.

Decorator Based Signal Subscriptions

With Blinker 1.1 you can also easily subscribe to signals by using the new **`connect_via()`** decorator:

```
from flask import template_rendered

@template_rendered.connect_via(app)
def when_template_rendered(sender, template, context, **extra):
    print 'Template %s is rendered with %s' % (template.name, context)
```

Core Signals

Take a look at Signals for a list of all builtin signals.