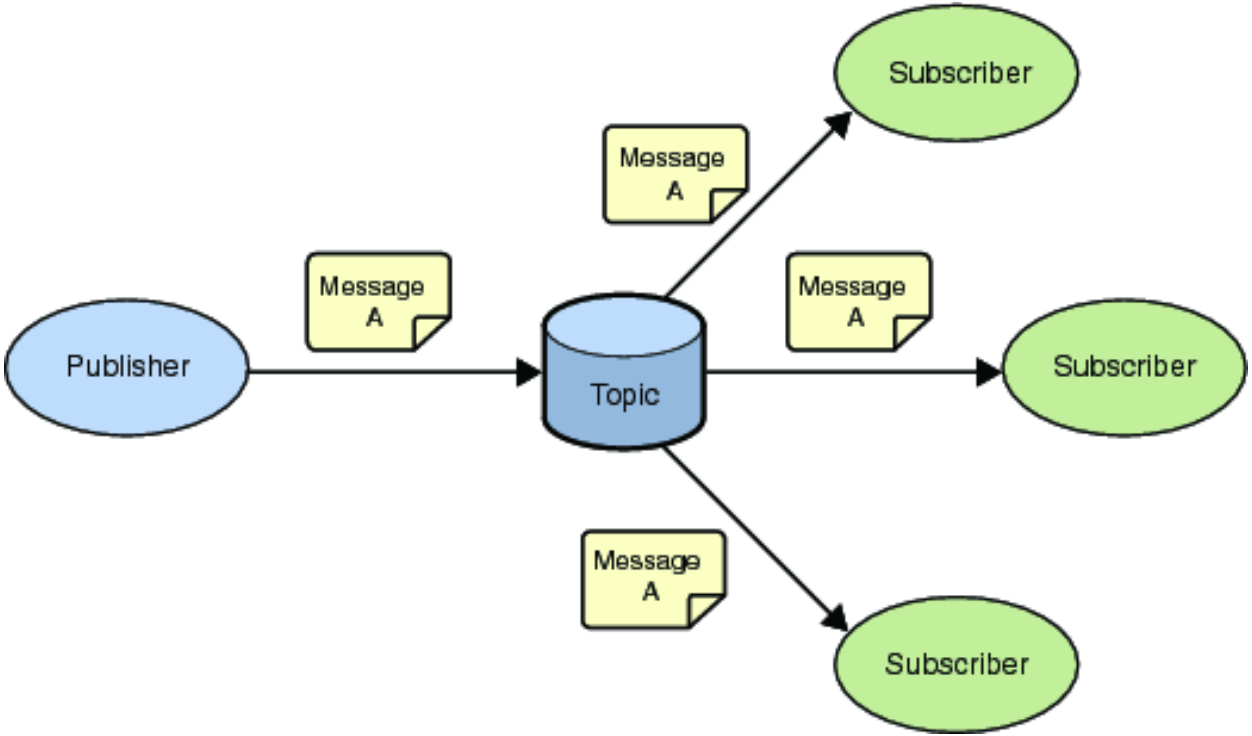
Blinker

Signaling For Python

Publish-Subscribe Pattern



What is Blinker?

- Blinker provides simple object-to-object & broadcast signalling for Python objects
- Lightweight & flexible, yet extremely powerful

Decoupling with Blinker

- Blinker handles brokering of events
- Publishers don’t know about subscribers
- Subscribers don’t know about publishers
- Publishers don’t need to import subscribers, & vice-versa
- Signals are multicast, so easy to reuse

Signals

- Signals are channels to notify subscribers that event *x* has occurred.
- Signals are registered to a Namespace by their name, or to the global Namespace (not recommended)
  - Namespaces are simply dictionaries of signals stored by name
- Broadcasting methods may only pass 1 argument (typically the sender), but may include additional data via kwargs
- Broadcasting is done via the signal.send() method:
  - signal.send(me, data_val=1)
- The send method returns a list of tuples with the results from receivers:
  - [(function_a, result), (function_b, result)]

Signals (cont'd)

- Subscribers may register to a signal in a number of ways:
  - @signal.connect (as a decorator)
  - @signal.connect_via(x) (as a decorator, when sender == *x*)
  - signal.connect(function)
  - with signal.connected_to(receiver): (execute block w/ signal temporarily connected)
- Subscribers may be disconnected:
  - signal.disconnect(function)
- Broadcasters may be notified that a receiver has been connected or disconnected to signal:
  - @signal.receiver_connected.connect
  - @signal.receiver_disconnected.connect

Blinker In Action: Flask

- Signals built in to Flask (since version 0.6):
  - template_rendered
  - request_started
  - request_finished
  - got_request_exception
  - appcontext_tearing_down
  - appcontext_pushed
  - appcontext_popped
  - message_flashed

Flask Example: Sentry

```python
from flask import Flask
from raven.contrib.flask import Sentry
from flask.signals import got_request_exception

app = Flask(__name__)

sentry = Sentry(app, dsn=app.config['SENTRY_DSN'])

@got_request_exception.connect
def log_exception_to_sentry(app, exception=None, **kwargs):
    """
    Logs an exception to sentry

    :param app: The current application
    :param exception: The exception that occurred
    """
    sentry.captureException(exception)

```

Use-Case: User Onboarding

- As a part of our application, we'd like to introduce a user-onboarding wizard
- Wizard "steps" are scattered throughout the application:
  - Submit Identification for Verification
  - Add a payment method
  - Add funds to the account
  - Create a campaign with creatives and bids

Use-Case: User-Onboarding (cont'd)

- Introduced Signals for several events around the system
  - Keep onboarding model independent of core / product-specific models
  - Reusable
- Subscribe to signals in onboarding model to populate completed steps

Use Case 2: User Alerts

- Need to maintain a list of events that occurred recently that require user attention (either dismiss or fix)
  - Example: Account ran out of funds
- Once user has added funds to the account, we can dismiss the alert

Use-Case 2: User Alerts (cont'd)

- Solution 1: add a call to dismiss any alerts once the user added funds
  - Drawbacks: Intrusive, and may have multiple ways to add funds (credit card, wire transfer, etc)
- Solution 2: re-use the account funded signal
  - Extremely easy to implement, keeps user alerts code independent of account funding code

Demo

Best Practices

- Use a Namespace to avoid potential collisions with other Signals
- Accept **kwargs in subscribing methods
  - Allows for additional data to be sent if needed by other subscribers with little resistance
- Keep signal declarations at the highest level, independent of any other code to prevent dependency nightmares