Flask at Scale

Miguel Grinberg
@miguelgrinberg

About Me

- Full-Stack Engineer at sdvi
- O'Reilly's Flask Web Development
- The Flask Mega-Tutorial
- blog.miguelgrinberg.com
- A bunch of open source packages

Why Flask?

Which set would you rather have?

I take the tub!

Some Initial Thoughts

- Can Flask Scale? Wrong question!
- Flask is not at the center of the world, and that is a good thing.
- Change is unavoidable, so better make it part of your workflow.
- The best Flask boilerplate/starter project is...

The Ultimate Flask Boilerplate ;-)

```
from flask import Flask
app = Flask(__name__)

@app.route('/')
def hello():
    return 'Hello World!'

if __name__ == '__main__':
    app.run()
```

Slack? Nope, It's Flack! v0.1

(Try it yourself: bit.ly/flackchat)

- Lame attempt at a chat service
- Flask API Backend
  - User registration: `POST` request to `/api/users`
  - Token request: `POST` request to `/api/tokens` (basic auth required)
  - Get users: `GET` request to `/api/users?updated_since=t` (token optional)
  - Get messages: `GET` request to `/api/messages?updated_since=t` (token optional)
  - Post message: `POST` request to `/api/messages` (token required)
  - Messages are written in markdown. Links are scraped and expanded.
  - Unit test suite with code coverage and code linting.
- Backbone JavaScript Client (Backbone??? Are we in 2013 or something?)

Slack? Nope, It's Flack! v0.1

(Try it yourself: bit.ly/flackchat)

```
flack/
├── flack.py
├── templates/
│   └── index.html
├── static/
│   └── client-side js and css files
├── tests.py
└── requirements.txt
```

How to Work with the Code

- Git repository: https://github.com/miguelgrinberg/flack
- Incremental versions are tagged: v0.1, v0.2, etc.
- Some commands to get you started:
  - `git checkout <version-tag>` ← *gets a specific version*
  - `pip install -r requirements.txt` ← *installs dependencies*
  - `python flack.py` ← *runs webserver (early versions)*
- To start client: Visit *http://<ip-address>:5000* on your browser

What's Wrong with Flack v0.1?

- Development
  - The whole backend is in a single, huge Python module.
  - Unit tests use a couple of hacks to configure the application properly.
  - Only way to apply configuration settings is via environment variables or by editing code.
- Production
  - There is no production web server strategy.
  - Messages are rendered during the processing of the request synchronously.
  - Clients have to poll the API very frequently to provide a "real-time" feel.

Photo credit: Simone Mescolini

Part I
Development Scaling

Refactoring Utility Functions v0.2

- Auxiliary functions that perform self-contained tasks can be easily moved to separate module(s).

flack.py

```
from utils import timestamp

timestamp()
```

utils.py

```
def timestamp():
    pass
```

Refactoring Utility Functions v0.2

```
flack/
├── flack.py
├── utils.py
├── templates/
│   └── index.html
├── static/
│   └── client-side js and css files
├── tests.py
└── requirements.txt
```

Refactoring Database Models v0.3

- Two modules that import symbols from each other are a recipe for disaster. This breaks horribly, but probably not how you think it does:

flack.py

```
from models import User
db = SQLAlchemy(app)

def new_user():
    u = User()
```

models.py

```
from flack import db
class User(db.Model):
    pass
```

Refactoring Database Models v0.3

- Solution #1: move imports down on the application side.
- Solution #2: Deal with `__main__` issues as best as possible.

flack.py

```
db = SQLAlchemy(app)
from models import User

def new_user():
    u = User()
```

models.py

```
try:
    from __main__ import db
except ImportError:
    from flack import db
class User(db.Model):
    pass
```

Refactoring Database Models v0.3

```
flack/
├── flack.py
├── models.py
├── utils.py
├── templates/
│   └── index.html
├── static/
│   └── client-side js and css files
├── tests.py
└── requirements.txt
```

Creating an Application Package v0.4

- Avoids the issues with `__main__`
- Code, templates and static files all move together inside the package.
- The application package can export just the symbols that are needed outside (`app` and `db`).
- A more robust start-up script can be built (Flask-Script, click, etc.).
- The start-up script can include maintenance operations:
  - `manage.py runserver` ← Runs the Flask development web server
  - `manage.py shell` ← Starts a Python console with a Flask app context
  - `manage.py createdb` ← Creates the application's database

Creating an Application Package v0.4

```
flack/
├── flack/
│   ├── __init__.py
│   ├── flack.py
│   ├── models.py
│   ├── utils.py
│   ├── templates/
│   │   └── index.html
│   └── static/
│       └── client-side js and css files
├── tests.py
├── manage.py    ← runserver, shell and createdb commands available here
└── requirements.txt
```

Refactoring API Authentication v0.5

- This is an similar to how the models were moved.
- Circular dependencies are handled by putting the imports after the database and models are initialized.

Refactoring API Authentication v0.5

```
flack/
├── flack/
│   ├── __init__.py
│   ├── flack.py
│   ├── auth.py
│   ├── models.py
│   ├── utils.py
│   ├── templates/
│   │   └── index.html
│   └── static/
│       └── client-side js and css files
├── tests.py
├── manage.py
└── requirements.txt
```

Refactoring Tests v0.6

- Moving tests to a package helps keep growing test suites organized.
- The manage.py launcher script can be extended even more:
  - `manage.py test` ← launches tests
  - `manage.py lint` ← runs code linter

Refactoring Tests v0.6

```
flack/
├── flack/
│   ├── __init__.py
│   ├── flack.py
│   ├── auth.py
│   ├── models.py
│   ├── utils.py
│   ├── templates/
│   │   └── index.html
│   └── static/
│       └── client-side js and css files
├── manage.py   ← test and lint commands added here
├── tests/
│   ├── __init__.py
│   └── tests.py
└── requirements.txt
```

Refactoring Configuration v0.7

- Putting the configuration in its own module helps organize different configuration sets (development, production, testing).
- The desired configuration is given in the `FLACK_CONFIG` environment variable.
- A bit less hacky to get unit tests to run on a different database.

Refactoring Configuration v0.7

```
flack/
├── flack/
│   ├── __init__.py
│   ├── flack.py
│   ├── auth.py
│   ├── models.py
│   ├── utils.py
│   ├── templates/
│   │   └── index.html
│   └── static/
│       └── client-side js and css files
├── config.py
├── manage.py
├── tests/
│   ├── __init__.py
│   └── tests.py
└── requirements.txt
```

Creating an API Blueprint v0.8

- Refactoring the API endpoints into a blueprint helps modularize the application. But, there are more cyclic dependencies to sort out.

flack/flack.py

```
app = Flask(__name__)
db = SQLAlchemy(app)

from .api import api as api_blueprint
app.register_blueprint(api_blueprint,
                       url_prefix='/api')
```

flack/api.py

```
from .flack import db

api = Blueprint('api', __name__)

@api.route('/users', methods=['POST'])
def new_user():
    pass
```

Creating an API Blueprint v0.8

```
flack/
├── flack/
│   ├── __init__.py
│   ├── flack.py   ← blueprint is initialized here
│   ├── auth.py
│   ├── models.py
│   ├── utils.py
│   ├── api.py
│   ├── templates/
│   │   └── index.html
│   └── static/
│       └── client-side js and css files
├── config.py
├── manage.py
├── tests/
│   ├── __init__.py
│   └── tests.py
└── requirements.txt
```

Refactoring Request Stats v0.9

- The code that reports request stats can easily be moved to a separate module. Its configuration can be added to the application's config object.

flack/flack.py

```
app = Flask(__name__)

from . import stats
```

flack/stats.py

```
from .flack import app

request_stats = []

def requests_per_second():
    return len(request_stats) /
        app.config['REQUEST_STATS_WINDOW']
```

Refactoring Request Stats v0.9

```
flack/
├── flack/
│   ├── __init__.py
│   ├── flack.py
│   ├── auth.py
│   ├── models.py
│   ├── utils.py
│   ├── api.py
│   ├── stats.py
│   ├── templates/
│   │   └── index.html
│   └── static/
│       └── client-side js and css files
├── config.py
├── manage.py
├── tests/
│   ├── __init__.py
│   └── tests.py
└── requirements.txt
```

Using an Application Factory Function v0.10

- Sometimes it is desirable to work with more than one application.
- Best example: unit tests that need applications with different configurations.

Using an Application Factory Function v0.10

- Flask extensions can use an app specific initialization inside the factory function via the `init_app()` method.

flack/__init__.py

```
db = SQLAlchemy()

def create_app(config_name=None):
    app = Flask(__name__)
    app.config.from_object(config[config_name])

    db.init_app(app)

    # ...
    return app
```

Using an Application Factory Function v0.10

- Not having a global `app` means a number of things need to change:
  - The `app.route` decorator cannot be used, so all endpoints need to be moved to blueprints.
  - Any references to `app` (such as `app.config[...]`) need to be removed.
  - Use the `current_app` context variable to access the application.
  - Manually push the app context when working outside of a request (such as in a background thread).

Using an Application Factory Function v0.10

```
flack/
├── flack/
│   ├── __init__.py   ← application factory function is here
│   ├── flack.py      ← endpoints that serve client application moved to main blueprint; app context used in thread
│   ├── auth.py
│   ├── models.py
│   ├── utils.py
│   ├── api.py
│   ├── stats.py
│   ├── templates/
│   │   └── index.html
│   └── static/
│       └── client-side js and css files
├── config.py
├── manage.py
├── tests/
│   ├── __init__.py
│   └── tests.py
└── requirements.txt
```

Creating an API Package v0.11

- Replacing the API module with a package leaves more space for growth by having a module per resource.

flack/api/__init__.py

```
from flask import Blueprint

api = Blueprint('api', __name__)

from . import tokens, users, messages
```

flack/api/tokens.py

```
from . import api

@api.route('/tokens', methods=['POST'])
def new_token():
    pass
```

Creating an API Package v0.11

```
flack/
├── flack/
│   ├── __init__.py
│   ├── flack.py
│   ├── auth.py
│   ├── models.py
│   ├── utils.py
│   ├── api/
│   │   ├── __init__.py
│   │   ├── tokens.py
│   │   ├── messages.py
│   │   └── users.py
│   ├── stats.py
│   ├── templates/
│   │   └── index.html
│   └── static/
│       └── client-side js and css files
├── config.py
├── manage.py
├── tests/
│   ├── __init__.py
│   └── tests.py
└── requirements.txt
```

What's Next?

- Refactoring as shown can go on as the application continues to evolve
- Examples:
  - models.py can become a package, with a module per model inside.
  - The api package can have sub-packages with different API versions.
  - The client side application can be moved into a separate project.

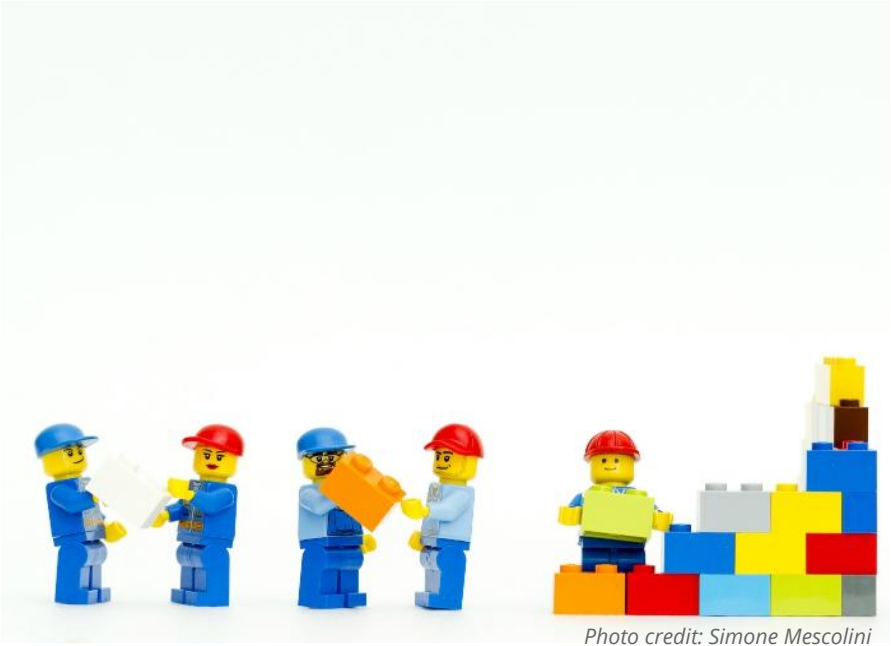


Photo credit: Simone Mescolini

Part II
Production Scaling

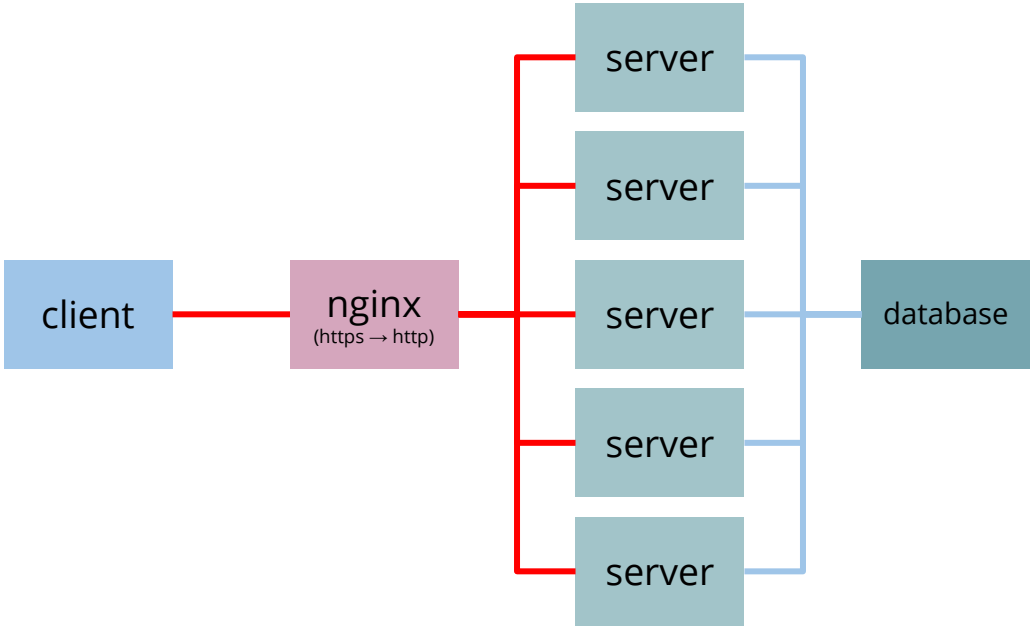
Scaling Web Servers

- Multiple threads
  - Limited use of multiple CPUs due to the GIL.
  - Application might need to synchronize access to shared resources.
- Multiple processes
  - Great way to take advantage of multiple CPUs.
  - Synchronization problems are less common than with threads.
- Green threads/coroutines (eventlet, gevent)
  - Extremely lightweight; hundreds/thousands of threads have small impact.
  - Cooperative multitasking makes synchronization much easier to manage.
  - Non-blocking I/O and threading functions.
  - I/O and threading functions in the standard library are incompatible.

Using Production Web Servers v0.12

- Gunicorn
  - Written in Python, fairly robust, easy to use.
  - Supports multiple processes, and eventlet or gevent green threads.
  - Limited load balancer
- Uwsgi
  - Written in C, very fast, extensive and somewhat hard to configure.
  - Supports multiple threads, multiple processes and gevent green threads.
- Nginx
  - Written in C, very fast.
  - Ideal to serve static files in production, bypassing Python and Flask.
  - Great as reverse proxy and load balancer in front of gunicorn/uwsgi servers.

Scaling with nginx



Bottlenecks: I/O-Bound vs. CPU-Bound

- I/O Bottlenecks
  - Flack example: scraping of links included in posts.
  - Solutions
    - Concurrent request handlers through multiple threads, processes or green threads.
    - Make I/O heavy requests asynchronous.
- CPU Bottlenecks
  - Flack example: markdown rendering of posts.
  - Solutions
    - Make CPU intensive requests asynchronous and offload the CPU heavy tasks to auxiliary threads or processes to keep the server unblocked.

Asynchronous HTTP Requests

- The request should start the actual task in the background and return.
- The status code in the response should be 202 (Accepted).
- The Location header should include a URL where the client can ask for status for the asynchronous task.
- Requests sent to the status URL should continue to return 202 while the background task is still in progress. The response body can include progress updates if desired.
- After the background task is finished, the status URL should return the response from the task, as it would have been returned by a synchronous version of the request.

Asynchronous Flask Requests v0.13

- The simplest approach is to run lengthy tasks in a background thread.
- An awesome decorator can be built to do this transparently for Flask.

synchronous...

```
@api.route('/messages', methods=['POST'])
@token_auth.login_required
def new_message():
    # ...
```

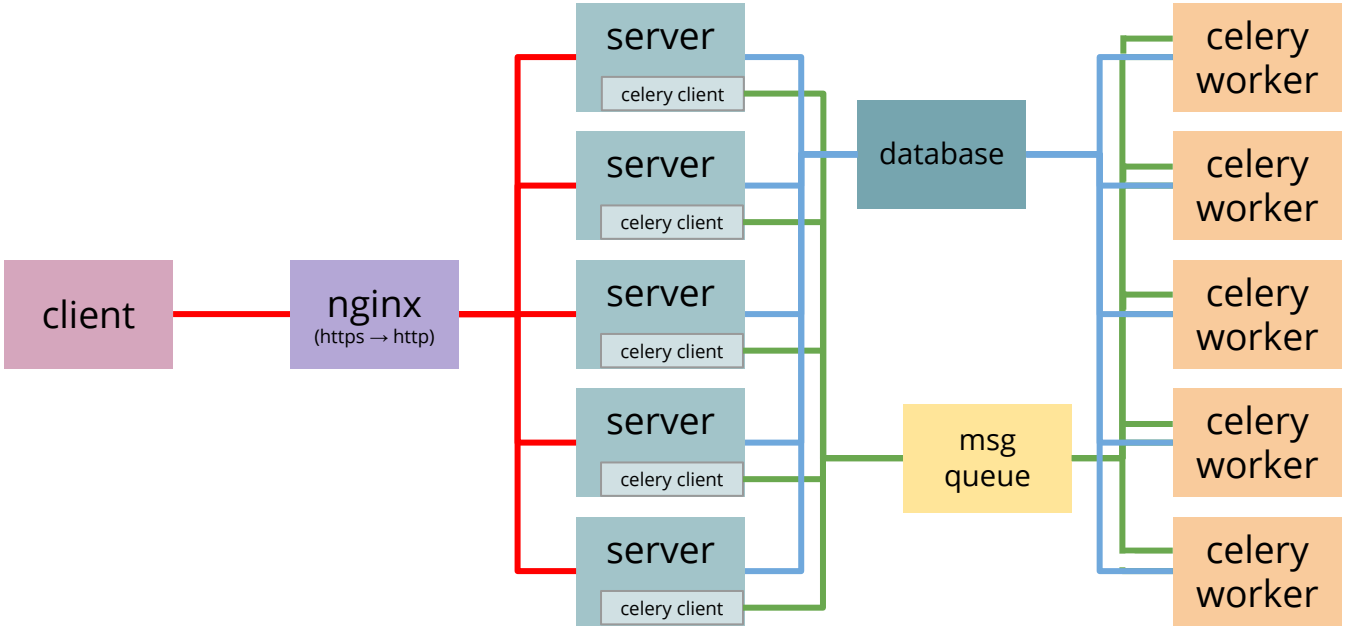
asynchronous!!!

```
@api.route('/messages', methods=['POST'])
@token_auth.login_required
@async
def new_message():
    # ...
```

Celery Workers v0.14

- Sometimes it is desirable to have a fixed pool of workers dedicated to running asynchronous tasks.
- Celery runs a pool of worker processes that listen for tasks provided by the main process. The processes communicate through a message queue (Redis, RabbitMQ, etc.).
- The `async` decorator can be modified to send tasks to Celery. No code changes to the application required!
- To start the celery worker processes, use `./manage.py celery`

Scaling with nginx and Celery



Battling Request/Response “Churn”

- With REST, clients are forced to poll to stay updated, adding extra load.
- Switching to a “server-push” model can help.
  - Option #1: Streaming
  - Option #2: Long-polling
  - Option #3: WebSocket
  - Option #4: Socket.IO (long-polling + WebSocket)

Socket.IO Server v0.15

- Server-push with Socket.IO

Server (Python)

```python
def push_model(model):
    socketio.emit('updated_model', {
        'class': model.__class__.__name__,
        'model': model.to_dict()
    })
```

Client (JavaScript)

```javascript
socket.on('updated_model', function(data) {
    if (data['class'] == 'User') {
        updateUser(data.model);
    }
    else if (data['class'] == 'Message') {
        updateMessage(data.model);
    }
});
```

Socket.IO Server v0.15

- Clients can push to the server too!

Client (JavaScript)

```
socket.emit('post_message',
            {source: args.message},
            token)
```

Server (Python)

```
@socketio.on('post_message')
def on_post_message(data, token):
    verify_token(token, add_to_session=True)

    msg = Message.create(data)
    # … write message to the database

    push_model(msg)
```

Socket.IO Server v0.15

- No need to poll to find disconnected users!
- To identify the user we use the Flask user session.

Server (Python)

```
@socketio.on('disconnect')
def on_disconnect():
    nickname = session.get('nickname')
    if nickname:
        user = User.query.filter_by(nickname=nickname).first()
        user.online = False
        # … write user to the database

        push_model(user)
```

Socket.IO + Celery v0.16

- Like request handlers, Socket.IO event handlers cannot be CPU heavy.
- Celery saves the day again!

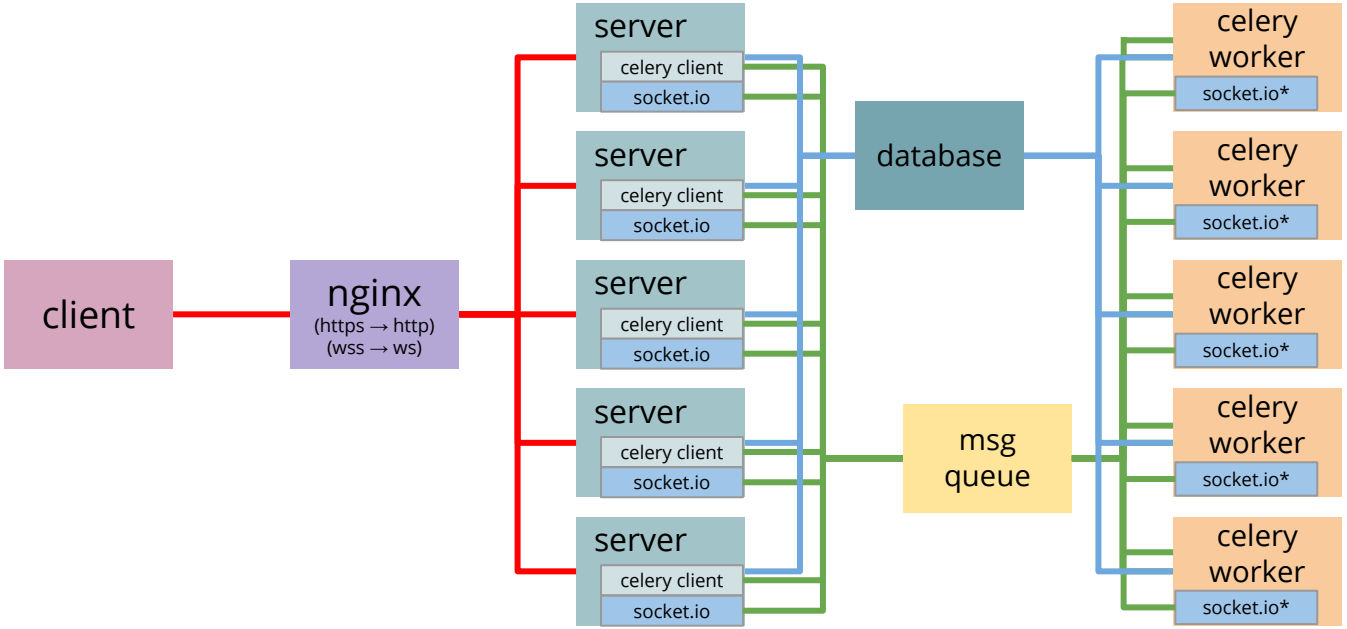
Socket.IO event handler

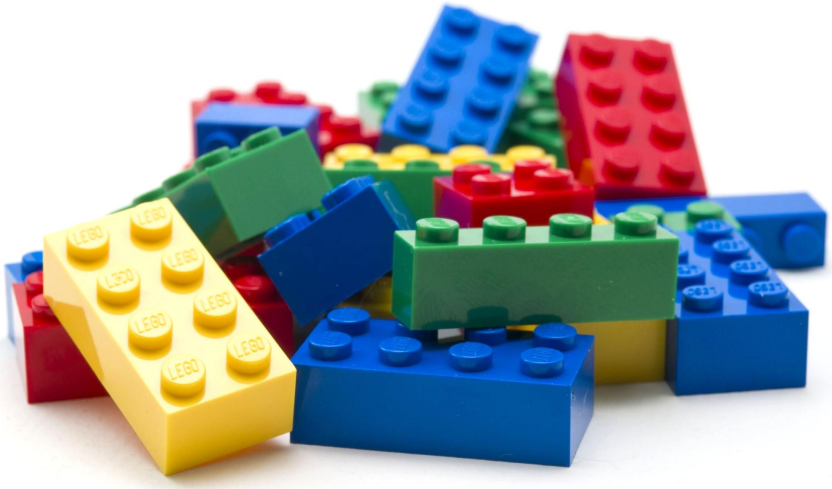
```
@socketio.on('post_message')
def on_post_message(data, token):
    verify_token(token)

    if g.current_user:
        post_message.apply_async(
            args=(g.current_user.id, data))
```

Celery task

```
@celery.task
def post_message(user_id, data):
    from .wsgi_aux import app
    with app.app_context():
        u = User.query.get(user_id).first()
        msg = Message.create(data, u)
        # … write message to the database
        push_model(msg)
        if msg.expand_links():
            push_model(msg)
```

Scaling with nginx, Celery and Flask-SocketIO





Thank You!

@miguelgrinberg