


We use cookies to make interactions with our websites and services easy and meaningful, to better understand how they are used and to tailor advertising. You can read more ([https://www.salesforce.com/company/privacy/full\\_privacy.jsp#nav\\_info](https://www.salesforce.com/company/privacy/full_privacy.jsp#nav_info)) and make your cookie choices here ([https://www.salesforce.com/company/privacy/full\\_privacy.jsp#nav\\_info](https://www.salesforce.com/company/privacy/full_privacy.jsp#nav_info)). By continuing to use this site you are giving us your consent to do this.

✕

Language Support (/categories/language-support) > Python (/categories/python-support)

### This article was contributed by The MemCachier Add-on

MemCachier (<http://memcachier.com>) manages and scales clusters of memcache servers so you can focus on your app. Tell us how much memory you need and get started for free instantly. Add capacity later as you need it.

 follow @MemCachier on twitter (<http://twitter.com/MemCachier>).

## Scaling a Flask Application with Memcache

🕒 Last updated 22 August 2018

### ☰ Table of Contents

- Prerequisites
- Create a Flask application for Heroku
- Add task list functionality
- Deploy the task list app on Heroku
- Add caching to Flask
- Further reading & resources

Memcache is a technology that improves the performance and scalability of web apps and mobile app backends. You should consider using Memcache when your pages are loading too slowly or your app is having scalability issues. Even for small sites, Memcache can make page loads snappy and help future-proof your app.

This guide shows how to create a simple Flask 1.0 (<http://flask.pocoo.org/>) application, deploy it to Heroku, then add Memcache to alleviate a performance bottleneck.



The sample app in this guide can be seen running here (<https://memcachier-examples-flask.herokuapp.com/>). You can view the source code (<http://github.com/memcachier/examples-flask>) or deploy it with this Heroku Button:



Deploy to Heroku

([https://heroku.com/deploy?  
template=http://github.com/memcachier/examples-flask](https://heroku.com/deploy?template=http://github.com/memcachier/examples-flask))

## Prerequisites

Before you complete the steps in this guide, make sure you have all of the following:

- Familiarity with Python (and ideally some Flask)
- A Heroku user account (signup is free and instant (<https://signup.heroku.com/signup/dc>))
- Familiarity with the steps in Getting Started with Python on Heroku (<https://devcenter.heroku.com/articles/getting-started-with-python>)
- Python and the Heroku CLI (<https://devcenter.heroku.com/articles/heroku-cli>) installed on your computer

## Create a Flask application for Heroku

Flask is a minimalist framework that doesn't require an application skeleton. Simply create a Python virtual environment and install Flask like so:

```
$ mkdir flask_memcache
$ cd flask_memcache
$ python -m venv venv
$ source venv/bin/activate
(venv) $ pip install Flask
```

Now that we've installed the Flask framework, we can add our app code. Let's create a task list that allows you to add and remove tasks.

Flask is very flexible in the way you structure your application. Let's add a minimal skeleton to get started. First, create an app in `task_list/__init__.py`:

```
import os
from flask import Flask

def create_app():
    app = Flask(__name__)
    app.config.from_mapping(
        SECRET_KEY = os.environ.get('SECRET_KEY') or 'dev_key'
    )

    return app
```



This small sample app will not use the `SECRET_KEY`, but it's always a good idea to configure it. Larger projects almost always use it, and it is used by many Flask addons.

We also need set the `FLASK_APP` environment variable to let Flask know where to find the application. For local development, set all required environment variables in a `.env` file:

```
FLASK_APP=task_list  
FLASK_ENV=development
```

To make sure Flask picks up the variables defined in the `.env` file, install `python-dotenv` :

```
(env) $ pip install python-dotenv
```

Now you can run the app with `flask run` and visit it at `http://127.0.0.1:5000/` , but the app doesn't do anything yet.

## Create a Heroku app

Associate your Flask skeleton with a new Heroku app with the following steps:

1. Initialize a Git repository and commit the skeleton. Start by adding a `.gitignore` file to make sure you don't commit files you don't want to. Paste the following into it:

```
venv/  
.env  
  
*.pyc  
__pycache__/  
  
instance/
```

Now commit all files to the Git repository:

```
$ git init  
$ git add .  
$ git commit -m 'Flask skeleton'
```

2. Create a Heroku app:

```
$ heroku create
```

In addition to creating the actual Heroku application, this command adds the corresponding remote to your local Git repository.

We now have a Heroku app, but our Flask app is not yet ready to be deployed to Heroku. We will make a few necessary changes later, but first let's implement some task list functionality.

## Add task list functionality

Let's add a task list to the app that enables users to view, add, and delete tasks. To accomplish this, we need to:

1. Set up the database
2. Create a `Task` model
3. Create the view and controller logic

## Set up a PostgreSQL database

Before we can configure a database in Flask, we need to create the database. On Heroku, you can add a free development database to your app like so:

```
$ heroku addons:create heroku-postgresql:hobby-dev
```

This creates a PostgreSQL database for your app and adds a `DATABASE_URL` environment variable that contains its URL. To use our database, we need a few libraries to manage our database connection, models, and migrations:

```
(env) $ pip install flask-sqlalchemy flask-migrate psycopg2
```

Now we can configure our database in `task_list/__init__.py`:

```
import os
from flask import Flask
from flask_sqlalchemy import SQLAlchemy
from flask_migrate import Migrate

db = SQLAlchemy()
migrate = Migrate()

def create_app():
    app = Flask(__name__)
    app.config.from_mapping(
        SECRET_KEY = os.environ.get('SECRET_KEY') or 'dev_key',
        SQLALCHEMY_DATABASE_URI = os.environ.get('DATABASE_URL') or \
            'sqlite:/// ' + os.path.join(app.instance_path, 'task_list.sqlite'),
        SQLALCHEMY_TRACK_MODIFICATIONS = False
    )

    db.init_app(app)
    migrate.init_app(app, db)

    from . import models

    return app
```

This creates a `db` object that is now accessible throughout your Flask app. The database is configured via the `SQLALCHEMY_DATABASE_URI`, which uses the `DATABASE_URL` if available. Otherwise, it falls back to a local SQLite database. If you want to run the application locally using the SQLite database, you need to create an `instance` folder:

```
$ mkdir instance
```

The database is now ready to use. Save the changes with:

```
$ git commit -am 'Database setup'
```

Note that the snippet above imports database models with `from . import models`. However, the app doesn't have any models yet. Let's change that.

## Create the Task model

---

To create and store tasks, we need to do two things:

1. Create the Task model in `task_list/models.py` :

```
from task_list import db

class Task(db.Model):
    id = db.Column(db.Integer, primary_key=True)
    name = db.Column(db.Text(), nullable=False)

    def __repr__(self):
        return '<Task: {}>'.format(self.name)
```

This gives us a task table with two columns: `id` and `name` .

2. Initialize the database and create migrations:

```
(venv) $ flask db init
Creating directory .../flask_memcache/migrations ... done
Creating directory .../flask_memcache/migrations/versions ... done
Generating .../flask_memcache/migrations/env.py ... done
Generating .../flask_memcache/migrations/README ... done
Generating .../flask_memcache/migrations/alembic.ini ... done
Generating .../flask_memcache/migrations/script.py.mako ... done
(venv) $ flask db migrate -m "task table"
INFO [alembic.runtime.migration] Context impl SQLiteImpl.
INFO [alembic.runtime.migration] Will assume non-transactional DDL.
INFO [alembic.autogenerate.compare] Detected added table 'task'
Generating .../flask_memcache/migrations/versions/c90b05ec9bd6_task_table.py ... done
```

The new migration can be found in `migrations/versions/c90b05ec9bd6_task_table.py` (your filename's prefix will differ).

Save your changes so far:

```
$ git add .
$ git commit -m 'Task table setup'
```

## Create the task list application

---

The actual application consists of a view that is displayed in the front end and a controller that implements the functionality in the back end. Flask facilitates the organization of back-end controllers via blueprints that are registered in the main application.

1. Create a controller blueprint in `task_list/task_list.py` :

```

from flask import (
    Blueprint, flash, redirect, render_template, request, url_for
)

from task_list import db
from task_list.models import Task

bp = Blueprint('task_list', __name__)

@bp.route('/', methods=('GET', 'POST'))
def index():
    if request.method == 'POST':
        name = request.form['name']
        if not name:
            flash('Task name is required.')
        else:
            db.session.add(Task(name=name))
            db.session.commit()

    tasks = Task.query.all()
    return render_template('task_list/index.html', tasks=tasks)

@bp.route('/<int:id>/delete', methods=('POST',))
def delete(id):
    task = Task.query.get(id)
    if task != None:
        db.session.delete(task)
        db.session.commit()
    return redirect(url_for('task_list.index'))

```

This controller contains all functionality to:

- GET all tasks and render the `task_list` view
- POST a new task that will then be saved to the database
- Delete existing tasks

2. Register the blueprint in `task_list/__init__.py`:

```

# ...
def create_app():
    app = Flask(__name__)

    # ...

    from . import task_list
    app.register_blueprint(task_list.bp)

    return app

```

With the controller set up, we can now add the front end. Flask uses the Jinja templating language, which allows you to add Python-like control flow statements inside `{% %}` delimiters. For our task list view, we first create a base layout that includes boilerplate code for all views. We then create a template specific to the task list.

1. Create a base layout in `task_list/templates/base.html`:

```

<!DOCTYPE HTML>
<title>{% block title %}{% endblock %} - MemCachier Flask Tutorial</title>
<!-- Fonts -->
<link href="https://cdnjs.cloudflare.com/ajax/libs/font-awesome/4.4.0/css/font-awesome.min."
      rel='stylesheet' type='text/css' />
<!-- Bootstrap CSS -->
<link href="https://maxcdn.bootstrapcdn.com/bootstrap/4.0.0/css/bootstrap.min.css"
      rel="stylesheet" />

<section class="content">
  <div class="container">
    <header>
      {% block header %}{% endblock %}
    </header>
    {% for message in get_flashed_messages() %}
      <div class="alert alert-danger">
        <p class="lead">{{ message }}</p>
      </div>
    {% endfor %}
    {% block content %}{% endblock %}
  </div>
</section>

<!-- Bootstrap related JavaScript -->
<script src="https://code.jquery.com/jquery-3.2.1.slim.min.js"></script>
<script src="https://cdnjs.cloudflare.com/ajax/libs/popper.js/1.12.9/umd/popper.min.js"></script>
<script src="https://maxcdn.bootstrapcdn.com/bootstrap/4.0.0/js/bootstrap.min.js"></script>

```

2. Create a view that extends the base layout in `task_list/templates/task_list/index.html` :

```
{% extends 'base.html' %}

{% block header %}
  <h1 style="text-align:center">{% block title %}Task List{% endblock %}</h1>
{% endblock %}

{% block content %}
  <!-- New Task Card -->
  <div class="card">
    <div class="card-body">
      <h5 class="card-title">New Task</h5>

      <form method="POST">
        <div class="form-group">
          <input type="text" class="form-control" placeholder="Task Name"
            name="name" required>
        </div>
        <button type="submit" class="btn btn-default">
          <i class="fa fa-plus"></i> Add Task
        </button>
      </form>
    </div>
  </div>

  <!-- Current Tasks -->
  {% if tasks %}
    <div class="card">
      <div class="card-body">
        <h5 class="card-title">Current Tasks</h5>

        <table class="table table-striped">
          {% for task in tasks %}
            <tr>
              <!-- Task Name -->
              <td class="table-text">{{ task['name'] }}</td>
              <!-- Delete Button -->
              <td>
                <form action="{{ url_for('task_list.delete', id=task['id']) }}"
                  method="POST">
                  <button type="submit" class="btn btn-danger">
                    <i class="fa fa-trash"></i> Delete
                  </button>
                </form>
              </td>
            </tr>
          {% endfor %}
        </table>
      </div>
    </div>
  {% endif %}
{% endblock %}
```

The view consists of two cards: one that contains a form to create new tasks, and another that contains a table with existing tasks and a delete button associated with each task.

Our task list is now functional. Save the changes so far with:

```
$ git add .
$ git commit -m 'Add task list controller and views'
```

We are now ready to configure the app to deploy on Heroku.



## Deploy the task list app on Heroku

Deploying the Flask application on Heroku is easily done with the following steps:

1. Install the `gunicorn` server and freeze dependencies into `requirements.txt` :

```
(venv) $ pip install gunicorn
(venv) $ pip freeze > requirements.txt
```

2. To let Heroku know how to start up your app, you need to add a `Procfile` (<https://devcenter.heroku.com/articles/procfile>) to its root directory:

```
$ echo "web: flask db upgrade; gunicorn task_list:create_app()" > Procfile
```

The above command always runs any outstanding database migrations before starting up the application.

3. Set your Heroku app's required config vars:

```
$ heroku config:set FLASK_APP=task_list
$ heroku config:set SECRET_KEY="`< /dev/urandom tr -dc 'a-zA-Z0-9' | head -c16`"
```

4. Deploy the app to Heroku:

```
$ git add .
$ git commit -m 'Add Heroku related config'
$ git push heroku master
$ heroku open
```

Test the application by adding a few tasks. We now have a functioning task list running on Heroku. With this complete, we can learn how to improve its performance with Memcache.

## Add caching to Flask

Memcache is an in-memory, distributed cache. Its primary API consists of two operations: `SET(key, value)` and `GET(key)` . Memcache is like a hashmap (or dictionary) that is spread across multiple servers, where operations are still performed in constant time.

The most common use for Memcache is to cache the results of expensive database queries and HTML renders so that these expensive operations don't need to happen over and over again.

## Set up Memcache

To use Memcache in Flask, you first need to provision an actual Memcached cache. You can easily get one for free with the MemCachier add-on (<https://elements.heroku.com/addons/memcachier>):

```
$ heroku addons:create memcachier:dev
```

Then we need to configure the appropriate dependencies. We will use `Flask-Caching` (<https://github.com/sh4nks/flask-caching>) to use Memcache within Flask.

```
(venv) $ pip install Flask-Caching pylibmc
(venv) $ pip freeze > requirements.txt
```

Now we can configure Memcache for Flask in `task_list/__init__.py`:

```
# ...
from flask_caching import Cache

cache = Cache()
# ...

def create_app():
    app = Flask(__name__)

    # ...

    cache_servers = os.environ.get('MEMCACHIER_SERVERS')
    if cache_servers == None:
        cache.init_app(app, config={'CACHE_TYPE': 'simple'})
    else:
        cache_user = os.environ.get('MEMCACHIER_USERNAME') or ''
        cache_pass = os.environ.get('MEMCACHIER_PASSWORD') or ''
        cache.init_app(app,
            config={'CACHE_TYPE': 'saslmemcached',
                'CACHE_MEMCACHED_SERVERS': cache_servers.split(','),
                'CACHE_MEMCACHED_USERNAME': cache_user,
                'CACHE_MEMCACHED_PASSWORD': cache_pass,
                'CACHE_OPTIONS': { 'behaviors': {
                    # Faster IO
                    'tcp_nodelay': True,
                    # Keep connection alive
                    'tcp_keepalive': True,
                    # Timeout for set/get requests
                    'connect_timeout': 2000, # ms
                    'send_timeout': 750 * 1000, # us
                    'receive_timeout': 750 * 1000, # us
                    '_poll_timeout': 2000, # ms
                    # Better failover
                    'ketama': True,
                    'remove_failed': 1,
                    'retry_timeout': 2,
                    'dead_timeout': 30}}}})

    # ...

    return app
```

This configures `Flask-Caching` with `MemCachier`, which allows you to use your Memcache in a few different ways:

- Directly access the cache via `get`, `set`, `delete`, and so on
- Cache results of functions with the `memoize` decorator
- Cache entire views with the `cached` decorator
- Cache Jinja2 snippets

## Cache expensive database queries

---

Memcache is often used to cache expensive database queries. This simple example doesn't include any expensive queries, but for the sake of learning, let's assume that getting all tasks from the database is an expensive operation.

To cache the Task query ( `tasks = Task.query.all()` ), we change the controller logic in `task_list/task_list.py` like so:

```
# ...

from task_list import db, cache

#...

@bp.route('/', methods=('GET', 'POST'))
def index():
    # ...

    tasks = cache.get('all_tasks')
    if tasks == None:
        tasks = Task.query.all()
        cache.set('all_tasks', tasks)
    return render_template('task_list/index.html', tasks=tasks)

# ...
```

Deploy and test this new functionality:

```
$ git commit -am 'Add caching with MemCachier'
$ git push heroku master
$ heroku open
```

To see what's going on in your cache, open the MemCachier dashboard:

```
$ heroku addons:open memcachier
```

The first time you loaded your task list, you should have gotten an increase for the `get miss` and `set` commands. Every subsequent reload of the task list should increase `get hits` (refresh the stats in the dashboard).

Our cache is working, but there is still a major problem. Add a new task and see what happens. No new task appears on the current tasks list! The new task was created in the database, but the app is serving the stale task list from the cache.

## Clear stale data

---

When caching data, it's important to **invalidate** that data when the cache becomes stale. In our example, the cached task list becomes stale whenever a new task is added or an existing task is removed. We need to make sure our cache is invalidated whenever one of these two actions is performed.

We achieve this by deleting the `all_tasks` key whenever we create or delete a new task in `task_list/task_list.py`:

```
# ...

@bp.route('/', methods=('GET', 'POST'))
def index():
    if request.method == 'POST':
        name = request.form['name']
        if not name:
            flash('Task name is required.')
        else:
            db.session.add(Task(name=name))
            db.session.commit()
            cache.delete('all_tasks')

    # ...

@bp.route('/<int:id>/delete', methods=('POST',))
def delete(id):
    task = Task.query.get(id)
    if task != None:
        db.session.delete(task)
        db.session.commit()
        cache.delete('all_tasks')
    return redirect(url_for('task_list.index'))
```

Deploy the fixed task list:

```
$ git commit -am 'Clear stale data from cache'
$ git push heroku master
```

Now when you add a new task, all the tasks you've added since implementing caching will appear.

## Use the Memoization decorator

Our caching strategy above (try to obtain a cached value and add a new value to the cache if it's missing) is so common that `Flask-Caching` has a decorator for it called `memoize`. Let's change the caching code for our database query to use the `memoize` decorator.

First, we put the task query into its own function called `get_all_tasks` and decorate it with the `memoize` decorator. We always call this function to get all tasks.

Second, we replace the deletion of stale data with `cache.delete_memoized(get_all_tasks)`.

After making these changes, `task_list/task_list.py` should look as follows:

```
# ...

@bp.route('/', methods=('GET', 'POST'))
def index():
    if request.method == 'POST':
        name = request.form['name']
        if not name:
            flash('Task name is required.')
        else:
            db.session.add(Task(name=name))
            db.session.commit()

            cache.delete_memoized(get_all_tasks)

    tasks = get_all_tasks()
    return render_template('task_list/index.html', tasks=tasks)

@bp.route('/<int:id>/delete', methods=('POST',))
def delete(id):
    task = Task.query.get(id)
    if task != None:
        db.session.delete(task)
        db.session.commit()
        cache.delete_memoized(get_all_tasks)
    return redirect(url_for('task_list.index'))

@cache.memoize()
def get_all_tasks():
    return Task.query.all()
```

Deploy the memoized cache list and make sure the functionality has not changed:

```
$ git commit -am 'Cache data using memoize decorator'
$ git push heroku master
```



Because the `get_all_tasks` function doesn't take any arguments, you can also decorate it with `@cache.cached(key_prefix='get_all_tasks')` instead of `@cache.memoize()`. This is slightly more efficient.

## Cache Jinja2 snippets

With the help of `Flask-Caching`, you can cache Jinja snippets in Flask. This is similar to fragment caching in Ruby on Rails, or caching rendered partials in Laravel. If you have complex Jinja snippets in your application, it's a good idea to cache them, because rendering HTML can be a CPU-intensive task.



Do not cache snippets that include forms with CSRF tokens.

To cache a rendered set of task entries, we use a `{% cache timeout key %}` statement in `task_list/templates/task_list/index.html`:

```

<!-- ... -->

<table class="table table-striped">
  {% for task in tasks %}
    {% cache None, 'task-fragment', task['id']|string %}
    <tr>
      <!-- ... -->
    </tr>
    {% endcache %}
  {% endfor %}
</table>

<!-- ... -->

```

Here the timeout is `None` and the key is a list of strings that will be concatenated. As long as task IDs are never reused, this is all there is to caching rendered snippets. The PostgreSQL database we use on Heroku does not reuse IDs, so we're all set.

If you use a database that *does* reuse IDs (such as SQLite), you need to delete the fragment when its respective task is deleted. You can do this by adding the following code to the task deletion logic:

```

from flask_caching import make_template_fragment_key
key = make_template_fragment_key("task-fragment", vary_on=[str(task.id)])
cache.delete(key)

```

Let's see the effect of caching the Jinja snippets in our application:

```

$ git commit -am 'Cache task entry fragment'
$ git push heroku master

```

You should now observe an additional `get hit` for each task in your list whenever you reload the page (except the first reload).

## Cache entire views

We can go one step further and cache entire views instead of snippets. This should be done with care, because it can result in unintended side effects if a view frequently changes or contains forms for user input. In our task list example, both of these conditions are true because the task list changes each time a task is added or deleted, and the view contains forms to add and delete a task.

You can cache the task list view with the `@cache.cached()` decorator in `task_list/task_list.py`:

```

# ...

def is_post():
    return (request.method == 'POST')

@bp.route('/', methods=('GET', 'POST'))
@cache.cached(unless=is_post)
def index():
    # ...

# ...

```



The `@cache.cached()` decorator must be directly above the definition of the `index()` function (i.e., below the `@bp.route()` decorator).

Since we only want to cache the result of the `index()` function when we `GET` the view, we exclude the `POST` request with the `unless` parameter. We could also have separated the `GET` and `POST` routes into two different functions.

Because the view changes whenever we add or remove a task, we need to delete the cached view whenever this happens. By default, the `@cache.cached()` decorator uses a key of the form `'view/' + request.path`, which in our case is `'view/'`. Delete this key in the create and delete logic in `task_list/task_list.py` just after deleting the cached query:

```
# ...
cache.delete_memoized(get_all_tasks)
cache.delete('view/')
```

To see the effect of view caching, deploy your application:

```
$ git commit -am 'Cache task list view'
$ git push heroku master
```

On the first refresh, you should see the `get hit` counter increase according to the number of tasks you have, as well as an additional `get miss` and `set`, which correspond to the view that is now cached. Any subsequent reload will increase the `get hit` counter by just one, because the entire view is retrieved with a single `get` command.

Note that view caching does *not* obsolete the caching of expensive operations or Jinja snippets. It is good practice to cache smaller operations within cached larger operations, or smaller Jinja snippets within larger Jinja snippets. This technique (called Russian doll caching) helps with performance if a larger operation, snippet, or view is removed from the cache, because the building blocks do not have to be recreated from scratch.

## Using Memcache for session storage

On Heroku, it's not advisable to store session information on disk, because dynos have an ephemeral filesystem that doesn't persist across restarts.

Memcache works well for storing information for short-lived sessions that time out. However, because Memcache is a cache and therefore not persistent, long-lived sessions are better suited to permanent storage options, such as your database.

To store sessions in Memcache, you need Flask-Session (<https://pythonhosted.org/Flask-Session/>):

```
(env) $ pip install Flask-Session
(venv) $ pip freeze > requirements.txt
```

Then, configure Flask-Session in `task_list/__init__.py`:

```

import pylibmc
from flask_session import Session

# ...

def create_app():

    # ...

    if cache_servers == None:
        # ...

    else:
        # ...

        app.config.update(
            SESSION_TYPE = 'memcached',
            SESSION_MEMCACHED =
                pylibmc.Client(cache_servers.split(','), binary=True,
                               username=cache_user, password=cache_pass,
                               behaviors={
                                   # Faster IO
                                   'tcp_nodelay': True,
                                   # Keep connection alive
                                   'tcp_keepalive': True,
                                   # Timeout for set/get requests
                                   'connect_timeout': 2000, # ms
                                   'send_timeout': 750 * 1000, # us
                                   'receive_timeout': 750 * 1000, # us
                                   '_poll_timeout': 2000, # ms
                                   # Better failover
                                   'ketama': True,
                                   'remove_failed': 1,
                                   'retry_timeout': 2,
                                   'dead_timeout': 30,
                               })

        )
    Session(app)

    # ...

```

Our task list app does not have any use for sessions but you can now use sessions in your app like so:

```

from flask import session
session['key'] = 'value'
session.get('key', 'not set')

```

## Further reading & resources

- MemCachier Add-on Page (<https://elements.heroku.com/addons/memcachier>)
- MemCachier Documentation (<https://devcenter.heroku.com/articles/memcachier>)
- Advance Memcache Usage (<https://devcenter.heroku.com/articles/advanced-memcache>)
- Flask Caching Documentation (<https://flask-caching.readthedocs.io/en/latest/>)
- Heroku Python Guide (<https://devcenter.heroku.com/articles/getting-started-with-python>)
- Flask Documentation (<http://flask.pocoo.org/docs/1.0/>)
- Flask Mega-Tutorial (<https://blog.miguelgrinberg.com/post/the-flask-mega-tutorial-part-i-hello-world>)



