How to set up a production-grade flask application using Application Factory Pattern and Celery

A highly adaptable and scalable workflow for structuring and configuring a production-grade flask application using flask blueprints, the application factory pattern and Celery.

Ange Uwase May 16 · 21 min read ★

Photo by Will Porada on Unsplash

Disclaimer

This is not a beginner tutorial. It is assumed that the reader is experienced with the flask web application framework, its commonly used libraries and celery.

Groundwork

1. Create a github repository for your project, and initialize it with a README and a python .gitignore file.

2. Clone the git repo onto your local machine

```
$ git clone <repo link>
```

3. Create a virtual environment and install flask and any other libraries you will need

```
$ pip install flask python-dotenv flask-mail celery redis
$ pip freeze > requirements.txt
```

Project Structure: Flask Blueprints

What are blueprints?

Flask is a very flexible web development framework. You as a developer are given full autonomy to decide how to structure your web application. If you are building a small project, there is no harm in having all of your code in a single module. However, the commonly accepted pattern for large projects is to break your project into multiple packages using Flask Blueprints.

A module is a single `.py` python file. A package is a folder that contains one or more modules along with an `__init__.py` file. The `__init__.py` file is what distinguishes a package from a standard folder: if it is present in a folder then that folder is a package; if it is not present then that folder is just a normal storage folder.

A blueprint is a package that encapsulates one specific piece of functionality in your application. You should think of a flask application structured using blueprints as several key pieces of functionality working together to deliver the complete web application.

A flask application is made up of blueprints (Author's own)

Before you start coding your project, it is a good idea to first give thought to what blueprints you can divide the application into. My personal approach is to use two blueprints, `auth` and `main`. The `auth` blueprint handles all the functionality related to users- registration, login, logout, password reset and account confirmation. The `main` blueprint handles the functionality and features unique to the application. You could also add a third blueprint, `api`, for handling programmatic access to the web application resources.

You should create the following project structure for a project with two blueprints:

```
|-application.py
|-config.py
|-.env
|-.gitignore
|-readme.md
|-requirements.txt
|-celery_worker.py
|-Dockerfile
|-docker-compose.yml
|-pytest.ini
|-env/
|-tests/
    |-conftest.py
    |-test_main.py
    |-test_auth.py
    |-test_models.py
|-app/
    |-__init__.py
    |-models.py
    |-forms.py
    |-tasks.py
    |-static/
    |-templates/
        |-base.html
        |-400.html
        |-403.html
        |-404.html
        |-405.html
```

```
        |-views.py
        |-forms.py
        |-templates/auth
            |-register.html
    |-main/
        |-__init__.py
        |-views.py
        |-forms.py
        |-templates/mai
            |-index.html
```

The table below gives a rundown of the various components:

File or Folder	Description
application.py	The file that contains the flask application instance for running the web application
config.py	The file that contains the configuration variables the application needs
requirements.txt	The file that contains a list of all the libraries and their dependecies your web application uses
.env	The file that contains all your application environment variables
celery_worker.py	The file that contains a starter script for launching a Celery worker
pytest.ini	Pytest configuration file
/tests/	The folder where the test modules are stored
/tests/test_models.py	Test module that contains tests for the models
/tests/test_main.py	Test module that contains the functional and unit tests for the main blueprint
/tests/test_auth.py	Test module that contains the functional and unit tests for the auth blueprint
/tests/conftest.py	This is the file where the pytest fixtures are defined
/app/	The package that contains your web application
/app/__init__.py	The file that contains theflask application factory function
/app/models.py	The file that contains the class definitions for the various models used in the application
/app/forms.py	The file that contains the class definitions of all forms used in the application
/app/tasks.py	The file that contains the background Celery task functions
/app/static/	The folder that contains the static content of the application
/app/templates/	The folder that contains the templates that are used by all the blueprints
/app/auth/	The package that contains the code related to user management functionality
/app/main/	The package that contains the code related to the main application functionality
/app/auth/__init__.py	The file in which the blueprint object gets instantiated
/app/auth/views.py	The file that contains the code related to the blueprint
/app/auth/templates/	The folder that contains the templates that belong to the auth blueprint

Common files in a flask application (Author's own)

When you use blueprints, the view functions that handle requests won't all be in one file; they are going to be split across different files. Each blueprint will have its own `views.py` file containing the code belonging to it. Instead of the routes having an `@app` decorator,

Once you have organised your project into blueprints, instead of the server passing requests to the flask application instance to be handled, the requests are pushed to the appropriate blueprint and the blueprint handles them. For the flask application instance to know the blueprints in the project and the routes that belong to it, the blueprints have to "register" with the flask application instance.

Configuration Management

Configuring a flask application is the process of specifying the value of key parameters used to control the behavior of the flask application instance and its extensions. The official flask documentation on this topic provides a nice list of all the built-in flask variables that can be configured to suit your needs.

Configuration methods

Flask provides 4 main ways you can configure a flask application: environment variables, config attribute of the flask app instance, a CFG file and an object.

***Environment variables**

When you have one or two configuration parameters you want to set, using environment variables is the easiest configuration method. To set the parameters you run the `set` (for Windows) or `export` (for Linux) command. Flask recommends that you set the `FLASK_APP` and `FLASK_ENV` environment variables using this method to be able to start up the flask development server in debug mode using the `flask run` command.

```
$set FLASK_APP = app.py
$set FLASK_ENV = development
```

Using environment variables for configuring key parameters is simple but doesn't scale well when you have many parameters or when you want to use multiple environments

Config Attribute of the flask app instance

The flask application instance exposes a dictionary-like object called `config`, accessed via dot notation, that allows you to set configuration variables to a desired value the same way you would set a value for a dictionary key. When this object is used, all the configuration code is written immediately after the instantiation of the flask object. This is because the configuration settings need to be available upon the instantiation of the flask application so that it can be properly configured before it is run.

```
app = Flask(__name__)
app.config['SECRET_KEY'] = 'hard to guess
app.config['TESTING'] = True
app.config['MAIL_SERVER'] = 'smtp.googlemail.com'
app.config['MAIL_PORT'] = 465
app.config['MAIL_USE_TLS'] = True
app.config['MAIL_USERNAME'] = os.environ.get('MAIL_USERNAME')
app.config['MAIL_PASSWORD'] = os.environ.get('MAIL_PASSWORD')
```

Having all your configuration code in the script where you instantiate the flask object is not flexible though. When the flask object gets instantiated it gets configured based on those settings right away, and there is no way to change the configuration of the flask instance after it has already been instantiated.

This becomes a problem when you want to have different “versions” of your application based on what you are doing. That is, you want a flask app configured for development when you are in development phase, a flask app configured for testing when you want to do testing and a flask app configured for production when you are ready to deploy.

Perhaps you used SQLite during development but want to use a postgreSQL database in production. Or you want to use a different SQLite database file for development, testing and production to keep things separate so that they don’t interfere with each other. The approach of being able to configure a flask application differently for different environments is the only way you can accomplish this.

*A CFG file

pull the configuration settings from that file using the `from_pyfile()` method it exposes. The argument that this method takes is the path to the file, which can be a path relative to the project root directory or an absolute path to the file. This method scales well but doesn't allow for specifying different configuration settings for different environments.

An example of the content of a configuration file called **config.cfg** could be:

```
DEBUG = False
SECRET_KEY = 'a bad secret key'
```

To configure the flask application using the **config.cfg** file stored in the root project directory:

```
app = Flask(__name__)
app.config.from_pyfile('config.cfg')
```

***An object**

The method of configuring a flask application that both scales well and allows you to specify different configuration settings for different environments is to define the configuration settings in a set of python classes.

In a python script file called **settings.py** or **config.py** located in the project root directory, you define 4 classes:

- class Config(object), which is the base configuration class that contains configurations settings that apply to all the environments. All the other 3 classes inherit from this class.
- class DevelopmentConfig(Config), which contains the configuration settings for the development environment.

- class ProductionConfig(Config), which contains the configuration settings for the production environment.

To configure a flask application instance using the config.py/ settings.py file you then use the `from_object()` method the config object exposes. The argument that this method takes indicates the name of the configuration script and the class from which the configuration data should be loaded. Suppose your configuration script is called config.py. During the development stage you would want to load the configuration data from the DevelopmentConfig class. The way to do this is shown below:

```
app = Flask(__name__)
app.config.from_object('config.DevelopmentConfig')
```

The production-grade way flask apps are configured

In practice, there are certain parameters such as the secret key, mail server username and password, and many others whose value you wouldn't want to explicitly hardcode in the config.py file for security reasons.

What lands up happening is that flask applications get configured using a combination of the environment variables method and object method. The sensitive parameters get set in the environment using the environment variables method and are then imported into the config.py file using the `os.environ.get()` method. A default value is provided in case the environment does not provide a value for the parameter. The non-sensitive parameters get explicitly defined in the config.py file.

If you don't want to set the environment variables using the terminal as explained earlier, there is a nifty little python package called `python-dotenv` . Once the package is installed, you create a `.env` file in your project root directory and define all your environment variables in it. You then tell your config.py file to load the environment configuration settings from the `.env` file by calling the `load_dotenv()` method. If you

In your `.env` file:

```
FLASK_APP = application.py
FLASK_ENV = development
CONFIG_TYPE = config.DevelopmentConfig
SECRET_KEY = 'A terrible secret key'
MAIL_USERNAME = username
MAIL_PASSWORD = password
CELERY_BROKER_URL = redis://localhost:6379
RESULT_BACKEND = redis://localhost:6379
```

.env hosted with ♡ by GitHub view raw

In your `config.py` file:

```
#!/usr/bin/env python

import os
from dotenv import load_dotenv
load_dotenv()


# Find the absolute file path to the top level project directory
basedir = os.path.abspath(os.path.dirname(__file__))

class Config:
    """
    Base configuration class. Contains default configuration settings + configuration settings a
    """
    # Default settings
    FLASK_ENV = 'development'
    DEBUG = False
    TESTING = False
    WTF_CSRF_ENABLED = True

    # Settings applicable to all environments
    SECRET_KEY = os.getenv('SECRET_KEY', default='A very terrible secret key.')

    MAIL_SERVER = 'smtp.googlemail.com'
```

```
    MAIL_USERNAME = os.getenv('MAIL_USERNAME', default='')
    MAIL_PASSWORD = os.getenv('MAIL_PASSWORD', default='')
    MAIL_DEFAULT_SENDER = os.getenv('MAIL_USERNAME', default='')
    MAIL_SUPPRESS_SEND = False

    SQLALCHEMY_TRACK_MODIFICATIONS = False

    CELERY_BROKER_URL = os.getenv('CELERY_BROKER_URL ')
    RESULT_BACKEND = os.getenv('RESULT_BACKEND')


class DevelopmentConfig(Config):
    DEBUG = True
    SQLALCHEMY_DATABASE_URI = "sqlite:///" + os.path.join(basedir, 'dev.db')

class TestingConfig(Config):
    TESTING = True
    WTF_CSRF_ENABLED = False
    MAIL_SUPPRESS_SEND = True
    SQLALCHEMY_DATABASE_URI = "sqlite:///" + os.path.join(basedir, 'test.db')

class ProductionConfig(Config):
    FLASK_ENV = 'production'
    SQLALCHEMY_DATABASE_URI = os.getenv('PROD_DATABASE_URI', default="sqlite:///" + os.path.join
```

config.py hosted with ♡ by GitHub view raw

Define Your Blueprints

Each blueprint must have an `__init__.py` file. In this file, you define the blueprint by instantiating an instance of the Blueprint class. The arguments passed to the class constructor are the name of the blueprint and the name of the folder containing the templates belonging to the blueprint. You then need to import the routes associated with

In `app/auth/__init__.py`:

```
#This bluepint will deal with all user management functionality

from flask import Blueprint

main_blueprint = Blueprint('main', __name__, template_folder='templates')

from . import views
```

__init__.py hosted with ♡ by GitHub view raw

In `app/main/__init__.py`:

```
#This bluepint will deal with all user management functionality

from flask import Blueprint

main_blueprint = Blueprint('main', __name__, template_folder='templates')

from . import views
```

__init__.py hosted with ♡ by GitHub view raw

You will need to write some initial code for testing purposes to ensure that everything is setup correctly.

When working with blueprints:

- Make sure that the decorator used to define any route uses the blueprint object.
- Make sure that the `render_template()` function argument takes the form `blueprint_name/template_name.html` . This is done to reflect the fact that a blueprint can only render templates that belong to it.

blueprints. `url_for(auth.login)`.

- Any reference you would make to the `app` object needs to be replaced by the `current_app` object. This is because when you are working with blueprints, you no longer have access to the flask application instance directly. You can only get access to it via its proxy, `current_app`.
- If you specified a prefix for a blueprint in the blueprint registration, the route in the view function decorator doesn't get that prefix (that is, the decorator is not `@auth_blueprint.route('/users/login')` as you would have thought). However, a request recieved for that route from a client must contain the prefix otherwise the server will return a 404 error.
- In the blueprint's directory, you need to create a templates folder, and then inside that templates folder another folder named after the blueprint. The templates associated with the blueprint are stored in the directory with the name of the blueprint. This approach is the advised way of storing templates associated with the view functions of a blueprint.

```
|project
    |-auth
        |-templates
            |-auth
```

In `app/auth/views.py`:

```
from . import main_blueprint
from flask import render_template, request, redirect, url_for

@main_blueprint.route('/')
def index():
    return render_template('main/index.html')
```

views.py hosted with ♡ by GitHub view raw

```
<h1>Hello world from the main blueprint!</h1>
```

register.html hosted with ♡ by GitHub view raw

In `app/main/views.py`:

```
from . import main_blueprint
from flask import render_template, request, redirect, url_for

@main_blueprint.route('/')
def index():
    return render_template('main/index.html')
```

views.py hosted with ♡ by GitHub view raw

In `app/main/templates/main/index.html`:

```
<h1>Hello world from the main blueprint!</h1>
```

register.html hosted with ♡ by GitHub view raw

Application Factory Pattern

Typically, a flask application is instantiated with a global scope. What this means is that when you run the flask application script, the flask app gets instantiated and configured right away. And unfortunately, once it is running, there is no way to change its configuration settings.

This is not ideal behavior. You want to be able to create different versions of the same flask application that are configured for different environments (development, testing or production).

A solution is to move the instantiation and configuration of the flask application into a function. The approach of using an application factory function and then calling it when

of configuring flask applications enables you to define the configuration of the flask application before you instantiate it. Essentially everything you need to setup for a flask application gets defined in the application factory function so that it returns a flask application instance that is fully set up and configured to your liking.

The function where the instantiation of the flask app is defined is called an application factory function (because it is literary used to produce manyflask applications, just like a factory is used to produce many products). It is defined in the project package's `app/__init__.py` module. It is then instantiated in the project's `application.py` module.

A summary of all the things that get defined in the application factory function:

1. Flask application instantiation
2. Flask application configuration
3. Flask extensions instantiation and initialization
4. Registration of blueprints
5. Registration of request callbacks
6. Registration of application-wide error handlers
7. Configuration of logging

Registering the blueprints

Blueprints are registered by passing the blueprint object to the `register_blueprint()` method the flask application instance exposes. The method takes a second optional parameter, which is a prefix that all URLs associated with the blueprint should start with. In the example below the prefix for the auth blueprint is `/users` . This means that to access the login page, for example, a user would have to request the `/users/login` route.

```
1
2  #register blueprints
3  def register_blueprints(app):
```

```
    app.register_blueprint(auth_blueprint, url_prefix='/users')
    app.register_blueprint(main_blueprint)
```

__init__.py hosted with ♡ by GitHub view raw

In `app/__init__.py`:

```
"""
This contains the application factory for creating flask application instances.
Using the application factory allows for the creation of flask applications configured
for different environments based on the value of the CONFIG_TYPE environment variable
"""

import os
from flask import Flask
from flask_mail import Mail

### Flask extension objects instantiation ###
mail = Mail()

### Application Factory ###
def create_app():

    app = Flask(__name__)

    # Configure the flask app instance
    CONFIG_TYPE = os.getenv('CONFIG_TYPE', default='config.DevelopmentConfig')
    app.config.from_object(CONFIG_TYPE)


    # Register blueprints
    register_blueprints(app)

    # Initialize flask extension objects
    initialize_extensions(app)

    # Configure logging
    configure_logging(app)

    # Register error handlers
    register_error_handlers(app)
```

```
37
38
39  ### Helper Functions ###
40  def register_blueprints(app):
41      from app.auth import auth_blueprint
42      from app.main import main_blueprint
43
44      app.register_blueprint(auth_blueprint, url_prefix='/users')
45      app.register_blueprint(main_blueprint)
46
47  def initialize_extensions(app):
48      mail.init_app(app)
49
50  def register_error_handlers(app):
51    pass
52
53
54  def configure_logging(app):
55    pass
```

__init__.py hosted with ♡ by GitHub view raw

In `application.py`:

```
1  from app import create_app
2
3  application = create_app()
4
5  if __name__ == '__main__':
6    application.run()
```

application.py hosted with ♡ by GitHub view raw

To test everything out to make sure your blueprints are setup correctly, run the following command in the terminal:

```
$ flask run
```

When you navigate to http://127.0.0.1:5000/users/register/you-email-address@gmail.com you should see “Hello, your-email-address@gmail.com, from the auth blueprint!”

Logging

What is logging and why is it important?

Logging is the process of recording information about your application. It is used for recording events as they occur and is a great tool for debugging any issues and gaining insight into how your application is working.

Things you should log include:

- User-related events, such as registration, sign-in, sign-out, incorrect password attempts
- Events specific to the functionality of your application (for example, for a blog it could be posting a new blog or a comment being added)
- Errors, both applicaton-specific errors as well as database operation errors.

You should not store any sensitive data in logs, since they are usually stored as text files and are therefore not secure.

The logging module

Flask works with the standard python logging module. The logging module has 4 submodules, which can be accessed via dot notation: loggers, handlers, filters and formatters.

Loggers are the objects that create log messages. When you create a log message you have to specify its criticality by using the function associated with the criticality level.

- Debug → 10 → debug()
- Info → 20 → info()
- Warning → 30 → warning()
- Error → 40 → error()
- Critical → 50 → critical()

There is a default logger object, which can be accessed and used without needing to configure anything. Every flask instance exposes it via the `app.logger` object. If you are working with blueprints, then you have to access it via the `current_app` object, the proxy for the flask application instance.

In your `app/main/views.py` :

```
from . import main_blueprint
from flask import render_template, request, redirect, url_for, current_app

@main_blueprint.route('/')
def index():
    current_app.logger.info("Index page loading")
    return render_template('main/index.html')
```

views.py hosted with ♡ by GitHub view raw

Unfortunately, the default logger only prints to the console. So if you want to log to a file, you need to configure a new logger instance. The default logger will still continue logging, but you can disable it if you choose:

```
from flask.logging import default_handler

app.logger.removeHandler(default_handler)
```

higher will be logged and any log messages with a criticality value lower than it will not be logged. This is useful in situations where you want to reduce the volume of log messages without deleting log calls from your source code. You can increase the minimal log level of messages to be written to the log, for example: ERROR messsages and above.

Handlers are the objects that direct the log messages to the right destination. The default handler, called a stream handler, sends log messages to the terminal. You can create different handler objects to route log messages to various destinations.

- To log to a file you would use a FileHandler. To send log messages as email you would use a SMTPHandler.
- The FileHandler method takes in the path of the log file you want to write to, including its name, and instantiates a FileHandler object that will send log messages to that file.
- The instance folder is generally used to store files that run at runtime (logs and database files). This folder needs to be added to your .gitignore file so that it is not tracked by version control.

```
file_handler = logging.FileHandler('instance/my-app-log.log')

app.logger.addHandler(file_handler)
```

- The FileHandler object writes the log messages to a single log file. This can lead to the log file becoming large very quickly. A better approach is to use the RotatingFileHandler object. It also writes log messages to a file, but it creates a new log file whenever the current log file exceeds a specified file size (maxBytes). It will create a new file up to a specified number of files (backupCount) before it starts overwriting the existing files.

```
from logging.handlers import RotatingFileHandler
```

Filters are used to add contextual information to log messages. For example, when logging requests, you can create a filter that adds the remote IP address where the request came from.

Log formatters are used to specify the format of the log messages. Each log message is a LogRecord object. Log formatters are used to specify which attributes of LogRecords you want shown and in what order they should appear.

Common attributes of LogRecords include:

- `%(asctime)s` - datetime when the LogRecord was created
- `%(filename)s` - filename portion of pathname
- `%(funcName)s` - name of function containing the logging call
- `%(levelname)s` - logging level for the message
- `%(lineno)d` - line number of source code where the logging call was issued (if available)
- `%(message)s` - the logged message
- `%(module)s` - module from which the logging call was issued

How to configure logging

You should configure logging before creating the flask application instance, otherwise it will use the default handler which writes log messages to the console. This is why configuring logging is done in the application factory function.

The steps required to configure logging:

1. Import the logging library. Optional: import `default_handler` from `logging.handlers`
2. Instantiate a handler object.

4. Instantiate a Formatter object.
5. Add the Formatter object to the handler object using the `setFormatter()` method the handler object exposes.
6. Add the handler object to the logger using the `addHandler()` method it exposes.
7. Deactivate default handler

In `app/__init__.py`:

```
def configure_logging(app):
    import logging
    from flask.logging import default_handler
    from logging.handlers import RotatingFileHandler

    # Deactivate the default flask logger so that log messages don't get duplicated
    app.logger.removeHandler(default_handler)

    # Create a file handler object
    file_handler = RotatingFileHandler('flaskapp.log', maxBytes=16384, backupCount=20)

    # Set the logging level of the file handler object so that it logs INFO and up
    file_handler.setLevel(logging.INFO)

    # Create a file formatter object
    file_formatter = logging.Formatter('%(asctime)s %(levelname)s: %(message)s [in %(filename)s:

    # Apply the file formatter object to the file handler object
    file_handler.setFormatter(file_formatter)

    # Add file handler object to the logger
    app.logger.addHandler(file_handler)
```

__init__.py hosted with ♡ by GitHub view raw

Restart the flask development server and navigate to the index page.

```
2021-05-14 16:14:31,910 INFO: Index page loading [in views.py: 6]
```

Custom Error Handling

HTTP Status Codes

When a client makes a request to a web server, the response message includes an HTTP status code. It's a 3-digit number that indicates the result of processing the request. Status codes are grouped into 5 catagories based on the first digit, and each category represents a type of response:

1. 1xx — informational response
2. 2xx — successful eg

- 200 (OK), for successful processing of the request

3. 3xx — redirection

- 302 (Found), for successfuly redirecting the client to a new URL

4. 4xx — client error

- 400 (Bad Request): when the client makes a request that the server can't understand or doesn't allow.
- 403 (Forbidden): when the client tries to access a restricted resource and doesn't have authorization to do so.
- 404 (Not Found): when a client requests a URL that the server does not recognise. The error message given should be something along the lines of "Sorry, what you are looking for just isn't there!".

along the lines of “Sorry, the method requested is not supported by this resource!”.

5. 5xx — server error

- 500 (Internal Server Error): Usually occurs due to programming errors or the server getting overloaded.

How to create a custom error page

You usually want to define your own custom error pages for 403, 404, 405 and 500 errors so that:

- the error pages can have the same look as the rest of your web application. By using a template that extends base.html the error pages will have the same navbar and footer as all your other webpages.
- you can provide navigation links to make it easier for the user to navigate back to the application.

Step 1: Define a template file for the error code, and save it in the templates folder where you save your base.html file.

An example custom error template for Page Not Found is shown below. You will need to define the custom error handler templates for 400, 403, 404, 405 and 500.

In your `app/templates/base.html` file:

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

{% block content %}
<h1 class="errorpage-title">Page Not Found (404)</h1>
<div class="errorpage-section">
  <h4>What you were looking for is just not there!</h4>
  <h4><a href="{{ url_for('main.index') }}">Flask Application</a></h4>
</div>
{% endblock %}
```

404.html hosted with ♡ by GitHub view raw

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

{% block content %}
<h1 class="errorpage-title">Page Not Found (404)</h1>
<div class="errorpage-section">
  <h4>What you were looking for is just not there!</h4>
  <h4><a href="{{ url_for('main.index') }}">Flask Application</a></h4>
</div>
{% endblock %}
```

404.html hosted with ♡ by GitHub view raw

Step 2: Define an error handler for each error code.

The error handler is the function that will render the error template when the error occurs. You can name it whatever you want, it must be decorated with the `@app.errorhandler()` decorator. This decorator takes as an argument the status code of the error it must handle.

The syntax for a custom error handler is shown below:

```
@app.errorhandler(status_code)
def function_name(error):
    # You can log the error or send an email here
    return render_template("status_code.html"), status_code
```

When you are using the application factory method, all the error handler functions get defined inside a helper function, `register_error_pages()`. This function gets called in the application factory function and registers all the error handlers at once. It is useful for defining the error handlers outside the `create_app()` function to keep it clean but still be able to make the flask application instance aware of the error handlers.

In your `app/__init__.py` file:

```
def register_error_handlers(app):

```

```
        return render_template('400.html'), 400

    # 403 - Forbidden
    @app.errorhandler(403)
    def forbidden(e):
        return render_template('403.html'), 403

    # 404 - Page Not Found
    @app.errorhandler(404)
    def page_not_found(e):
        return render_template('404.html'), 404

    # 405 - Method Not Allowed
    @app.errorhandler(405)
    def method_not_allowed(e):
        return render_template('405.html'), 405

    # 500 - Internal Server Error
    @app.errorhandler(500)
    def server_error(e):
        return render_template('500.html'), 500

```

__init__.py hosted with ♡ by GitHub view raw

This setup registers the custom error handlers with the flask application instance. Blueprints support the `errorhandler()` decorator as well, which means that you can register error handlers with a blueprint. However, this way of registerming custom error handlers is not advised, because it can't be used by other blueprints. It is better to register error handlers with the application instance so that they can view used by all view functions in all blueprints.

To test the custom error handlers, Flask provides an `abort()` function that can be used to manually throw an error. It takes as an argument the status code of the error you want to throw. A default error page for that error will be displayed but it's quite plain.

In your `app/main/views.py` :

```
@main_blueprint.route('/')
def index():
    current_app.logger.info("Index page loading")
    return render_template('main/index.html')

@main_blueprint.route('/admin')
def admin():
    abort(400)
```

views.py hosted with ♡ by GitHub view raw

If an error handler for that error has been registered, it will be called to display the custom error page instead of the default page.

Restart the flask development server. Navigate to http://127.0.0.1:5000/admin and you should see the custom error message. Change the argument of the `abort()` function to 403, 404, 405 and finally 500 to make sure all the custom error pages are being rendered correctly.

Celery

What is celery?

Celery is an open source asynchronous task queue based on distributed message passing. Although it supports scheduling, it is most commonly used for asynchronous task processing. When your application has long-running tasks such as processing data, generating reports or network-related tasks like sending emails, it is preferable that you run these tasks in the background instead of running them in the same process as a request.

A distributed messaging system has three components:

- a publisher (celery client that issues background jobs. The celery client runs with the flask application)

background tasks)

Distributed message passing basically means that Celery relies on a middleman called a broker to act as an intermediary between it and the task producer. To initiate a task the client sends a message to the message queue. The message queue then passes the message along to the worker and does whatever is required. The worker returns the status or result of the job to the client via the message broker. At no point do the client and worker communicate with each other directly. They are, for all intents and purposes, not even aware of each other. They each only deal with the message queue.

The main advantages of using a distributed messaging architecture are:

1. scalability: each component can scale individually as needed without affecting the others
2. loose-coupling: the worker only cares about the message it receives, not who produced it
3. the message queue provides real time messaging to the worker. The worker doesn't need to constantly poll the queue to see if there is a message

Using Celery

Installation

If you have not done so already, install celery and the package for the message broker of your choice. I have chosen to use Redis.

```
$ pip install celery redis
```

Configuration

To configure Celery you need to define two parameters, `CELERY_BROKER_URL` and `RESULT_BACKEND`. The values for these parameters is the URL of the broker you have chosen. It essentially tells Celery where the broker is running so that Celery will be able

The `RESULT_BACKEND` parameter defines where the task results get stored when you run a task that returns something. If you are running background tasks that don't return a result or status then you don't need to defune this parameter. If you do want to return results your message broker can also be your result backend. That is why I have given them the same URL.

1. Define the URL of the redis client in the `.env` file:

```
CELERY_BROKER_URL = redis://localhost:6379

RESULT_BACKEND = redis://localhost:6379
```

2. Configure Celery in the `Config` class in `config.py`:

```
CELERY_BROKER_URL = os.getenv('CELERY_BROKER_URL ')

RESULT_BACKEND = os.getenv('RESULT_BACKEND')
```

Celery Setup

1. Instantiate a Celery object in `app/__init__.py`:

```
import os
from flask import Flask, render_template
from flask_mail import Mail
from celery import Celery  # NEW!!!!!
from config import Config  # NEW!!!!!
import logging
from flask.logging import default_handler
from logging.handlers import RotatingFileHandler

### Flask extension objects instantiation ###
mail = Mail()

```

```python
### Application Factory ###
def create_app():

    app = Flask(__name__)

    # Configure the flask app instance
    CONFIG_TYPE = os.getenv('CONFIG_TYPE', default='config.DevelopmentConfig')
    app.config.from_object(CONFIG_TYPE)

    # Configure celery
    celery.conf.update(app.config)             # NEW!!!!!

    # Register blueprints
    register_blueprints(app)

    # Initialize flask extension objects
    initialize_extensions(app)

    # Configure logging
    configure_logging(app)

    # Register error handlers
    register_error_handlers(app)

    return app
```

__init__.py hosted with ♡ by GitHub view raw

In the code above we instantiate a new Celery object by calling the Celery class object. The arguments it expects are:

- the name of the module where the celery object is being instantiated
- the URL for the broker, which tells celert where the broker service is running
- the URL of the result backend

```
celery = Celery(__name__, broker=Config.CELERY_BROKER_URL,
result_backend=Config.RESULT_BACKEND)
```

Any other configurations for Celery are passed to the Celery object from the flask application's configurations inside the application factory function:

```
celery.conf.update(app.config)
```

2. Define tasks you want to run in the background

For this example we will setup Celery to send emails in the background.

The `flask-mail` extension has already been configured.

Background tasks are defined as functions. Each function needs to be decorated with the `@celery.task` decorator. It's a good idea to define all the tasks in a separate module.

In `app/tasks.py` create a task for sending emails:

```
from flask_mail import Message
from . import mail, celery
from flask import current_app

@celery.task(name='app.tasks.send_celery_email')
def send_celery_email(message_data):
    app = current_app._get_current_object()
    message = Message(subject=message_data['subject'],
                      recipients=[message_data['recipients']],
                      body=message_data['body'],
                      sender=app.config['MAIL_DEFAULT_SENDER'])

    mail.send(message)
```

tasks.py hosted with ♡ by GitHub view raw

```python
from . import auth_blueprint
from flask import render_template, request, redirect, url_for
from ..tasks import send_celery_email

@auth_blueprint.route('/register/<string:email>')
def register(email):
    message_data={
        'subject': 'Hello from the flask app!',
        'body': 'This email was sent asynchronously using Celery.',
        'recipients': email,

    }
    send_celery_email.apply_async(args=[message_data])
    return render_template('auth/register.html', email=email)
```

views.py hosted with ♡ by GitHub view raw

To call the task I used the `apply_async()` method and pass it the arguments that the task function needs using the `args` keyword.

3. Write a script for launching the celery worker.

In `celery_worker.py`:

```python
#!/usr/bin/env python
import os
#from app import create_app, celery
from app import create_app

app = create_app()
app.app_context().push()

from app import celery
```

celery_worker.py hosted with ♡ by GitHub view raw

The code above creates a flask application, and then uses it to establish an application context for the celery instance to run in. The application context is necessary for Celery

It is necessary to import the celery instance because Celery will need it to execute the background tasks.

Testing everything out

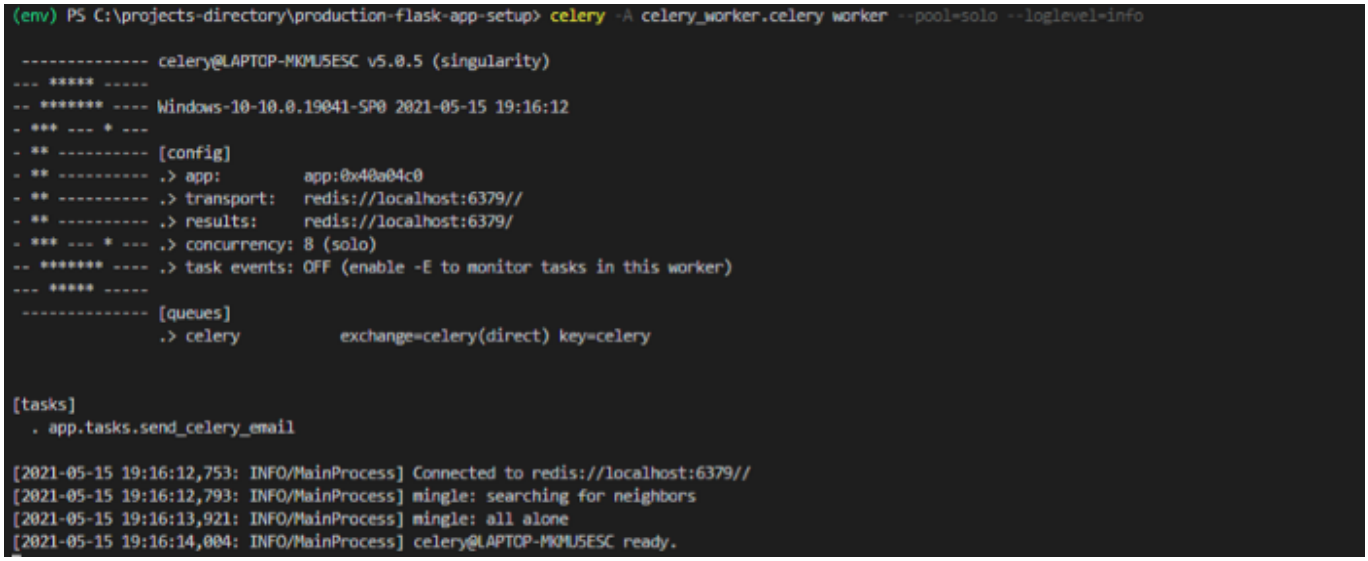
1. Open up docker and run a Redis container.

```
$ docker run -d -p 6379:6379 redis
```

2. Open up a new terminal window, activate the virtual environment and start a celery client:

```
$ celery -A celery_worker.celery worker --pool=solo --loglevel=info
```

You should see the clery client start up. It will show you that it has connected to the redis client, and also show you the tasks that have been created for Celery to run in the background.



Celery client terminal output (Author's own)

If you experience any issues, uninstall and reinstall celery and try again. Also play around with different execution pools as I have found that that can also make a difference.

If you check the terminal where the Celery client is running, you should see the following to know that the task has been successfully executed:

Celery executing the email sending background task (Author's own)

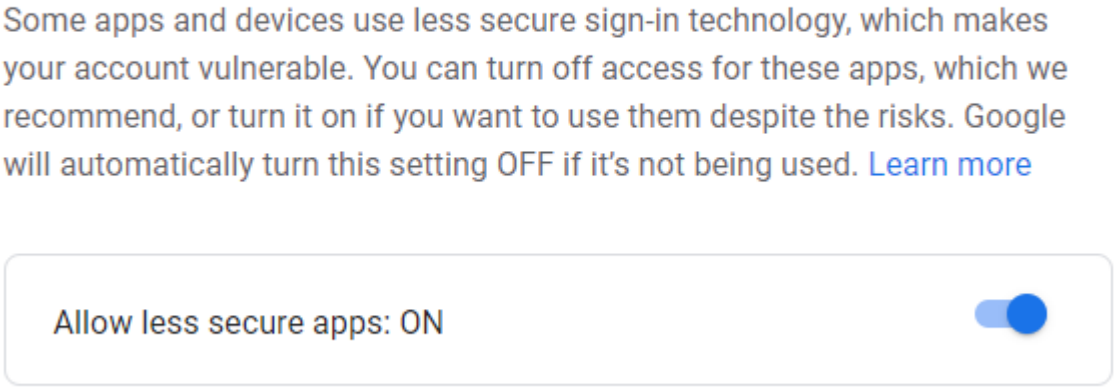
The last line of the output:

```
[2021-05-15 19:36:39,651: INFO/MainProcess] Task
app.tasks.send_celery_email[59e09990-ed19-4209-8afb-3bdd87ec80c6]
succeeded in 3.1569999999919673s: None
```

You should receive an email from the mail delivery subsystem stating that the `your-email-address@gmail.com` address could not be found.

Replace the `your-email-address` part of the URL with a valid email address and you should receive an email with the subject and body as defined in `app/views.py`.

Remember that for the flask application to send emails using the google SMTP server on your behalf you need to have allowed less secure apps access.



Allow less secure apps (Author's own)

Conclusion

That is it! Once this setup is completed you can proceed to develop your web application.

You can find this repo on my GitHub. You are more than welcome to fork the repo and use it as a start for a fully setup and configured flask web application. If you do so remember to add a `.env` file will all your environment variables as that file is not uploaded to Github.