Flask By Example

Miguel Grinberg
PyCon 2014

Flask and Me

- My blog **http://blog.miguelgrinberg.com** is powered by Flask.
- I wrote the Flask Mega-Tutorial 18-part series.
- I wrote several articles on API development with Flask.
- My most popular Flask extensions:
    - Flask-SocketIO (WebSocket communication)
    - Flask-Migrate (Database migrations with Alembic)
    - Flask-HTTPAuth (RESTful authentication)
    - Flask-PageDown (Live Markdown editor)
    - Flask-Moment (Rendering of dates and times)
- I wrote the book "**Flask Web Development**" for O'Reilly, in bookstores in May 2014.

About This Tutorial

About This Tutorial

Tutorial Pre-requisites

- Previous Python coding experience
- Basic knowledge of HTML and CSS
- A bit of JavaScript will definitely not hurt

Software Requirements

- Python 2.7 or 3.3+ on any supported OS
- virtualenv (or pyvenv on Python 3.4)
- git
- Network connection (only to install the application)

About This Tutorial (cont'd)

Your homework:

- Today:
  - Watch me build an application
  - Ask questions
- Later:
  - Hack on the example application, take what you want to seed your own project!
  - Complete your knowledge with the documentation
  - Ask questions
  - Show me what you build!

Talks

The Example Application

Talks

Features:

- One or more speakers can publish their talks.
- For each talk the slides and/or video can be embedded.
- The home page shows a timeline of talks by all speakers.
- Each speaker has a profile page with information and a list of talks.
- Users can write questions or comments using Markdown syntax.
- Speakers moderate comments written on their talks.
- Administrators moderate comments written on all talks.
- Email notifications are sent when new comments are written.
- Lists of talks and comments are paginated.
- Check out the live app at **http://talks.miguelgrinberg.com**.

Talks (cont'd)

Project structure:

- Available on github: **https://github.com/miguelgrinberg/flask-pycon2014**.
- There are 24 incremental versions tagged `v0.1` to `v0.24`.
- You can checkout any version to run or study in detail.
- Click the commit on github to see a detailed list of changes that went into a particular feature.

Application Setup

Clone the git repository:

```shell
$ git clone https://github.com/miguelgrinberg/flask-pycon2014.git
$ cd flask-pycon2014
```

Create a virtual environment (Linux/OSX):

```shell
$ virtualenv venv
$ source venv/bin/activate
(venv) $ pip install -r requirements.txt
```

Create a virtual environment (Windows):

```shell
> virtualenv venv
> venv\Scripts\activate
(venv) > pip install -r requirements.txt
```

Application Setup (cont'd)

Register a user:

```shell
(venv) $ python manage.py adduser --admin <your-email-address> <your-username>
Password: <your-password>
Confirm: <your-password>
User <your-username> was registered successfully.
```

Configure a gmail account as email server (Linux/OSX):

```shell
(venv) $ export MAIL_USERNAME=<your-gmail-username>
(venv) $ export MAIL_PASSWORD=<your-gmail-password>
```

Configure a gmail account as email server (Windows):

```shell
(venv) > set MAIL_USERNAME=<your-gmail-username>
(venv) > set MAIL_PASSWORD=<your-gmail-password>
```

Application Setup (cont'd)

Start the web server:

```shell
(venv) $ python manage.py runserver
 * Running on http://127.0.0.1:5000/
 * Restarting with reloader
```

Type `http://localhost:5000` in your browser's address bar!

Flask from Scratch

Up and Running

Step 1: Install Flask in a virtual environment

```shell
$ virtualenv venv
$ source venv/bin/activate
(venv) $ pip install flask
```

Up and Running (cont'd)

Step 2: Create an application instance

hello.py

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

Up and Running (cont'd)

Step 3: Define routes

hello.py

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

@app.route('/')
def index():
    return '<h1>Hello World!</h1>'

@app.route('/user/<name>')
def user(name):
    return '<h1>Hello, {0}!</h1>'.format(name)
```

Up and Running (cont'd)

Step 4: Start the development web server

hello.py

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

@app.route('/')
def index():
    return '<h1>Hello World!</h1>'

@app.route('/user/<name>')
def user(name):
    return '<h1>Hello, {0}!</h1>'.format(name)

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

Up and Running (cont'd)

Step 5: Run as a normal Python script!

```shell
(venv) $ python hello.py
 * Running on http://127.0.0.1:5000/
 * Restarting with reloader
```

Decorators

- Decorators are used extensively by the framework and extensions to register application provided functions as callbacks.
- Useful decorators:
  - `route` registers functions to handle routes.
  - `before_request` registers a function to run before request handlers.
  - `before_first_request` is similar, but only once at the start.
  - `after_request` registers a function to run after request handlers run.
  - `teardown_request` registers a function to run after request handlers run, even if they throw an exception.
  - `errorhandler` defines a custom error handler.
- Many Flask extensions define their own decorators as well.

Context Globals

- Context globals avoid the need to pass important variables to request handlers.
- Flask's application context defines the following context globals:
    - `current_app` is the application instance.
    - `g` is a global dictionary for request data storage.
- Flask's request context defines the following context globals:
    - `request` is the request being processed by the thread.
    - `session` is the user session storage.
- Some Flask extensions define their own context globals as well.

Helper functions

- Flask provides several auxiliary functions for , among them:
  - `url_for()` generates links to routes or static files.
  - `render_template()` renders Jinja2 templates.
  - `redirect()` generates a redirect response.
  - `jsonify()` generates a JSON response.
  - `abort()` generates an error response (throws an exception).
  - `flash()` registers a message to display to the user.

Flask Has Awesome Documentation

- Read it!

Version 0.1

Basic Project Structure

Structure for Larger Projects

```
|-talks                   <-- project folder
   |-app/                 <-- application package
   |-tests/               <-- unit tests package
   |-venv/                <-- virtual environment
   |-requirements.txt     <-- dependencies
   |-config.py            <-- configurations
   |-manage.py            <-- launch script
```

This structure is not set in stone. Customize as you see fit!

The Application Package

- Templates, static files get each a dedicated folder.
- Blueprints are implemented in sub-packages, and also get a sub-folder inside templates.
- Common functionality such as models are implemented as modules.

```
|-app/                  <-- application package
  |-templates/          <-- base template folder
    |-talks/            <-- main blueprint templates
  |-static/             <-- static files
  |-talks/              <-- application blueprint
  |-__init__.py         <-- application factory
  |-models.py           <-- database models
```

Configuration

The base configuration class holds common settings, overloaded in subclasses as necessary.

config.py

```python
import os

class Config:
    SECRET_KEY = os.environ.get('SECRET_KEY')

class DevelopmentConfig(Config):
    DEBUG = True
    SECRET_KEY = os.environ.get('SECRET_KEY') or 't0p s3cr3t'

class TestingConfig(Config):
    TESTING = True

class ProductionConfig(Config):
    pass

config = {
    'development': DevelopmentConfig,
    'testing': TestingConfig,
    'production': ProductionConfig,

    'default': DevelopmentConfig
}
```

Application Factory Pattern

- The application instance is created and configured at run-time.
- Routes are imported from a blueprint.

app/__init__.py

```
from flask import Flask
from config import config

def create_app(config_name):
    app = Flask(__name__)
    app.config.from_object(config[config_name])

    from .talks import talks as talks_blueprint
    app.register_blueprint(talks_blueprint)

    return app
```

Blueprints

- Blueprints are containers for routes, static files and/or templates.
- A blueprint becomes part of the application when it is registered with it.

app/talks/__init__.py

```python
from flask import Blueprint
talks = Blueprint('talks', __name__)
from . import routes
```

app/talks/routes.py

```python
from flask import render_template
from . import talks

@talks.route('/')
def index():
    return render_template('talks/index.html')

@talks.route('/user/<username>')
def user(username):
    return render_template('talks/user.html', username=username)
```

Templates

- Templates help separate logic and presentation.
- The default template engine for Flask is Jinja2, by the same developer.
- Dynamic parts are specified with placeholder variables.
- A wide array of control directives are available in templates.

app/templates/talks/index.html

```html
<h1>Hello, World!</h1>
```

app/templates/talks/user.html

```html
<h1>Hello, {{ username }}!</h1>
```

Launch script

Flask-Script is an extension that adds command line options to Flask.

shell

```
(venv) $ pip install flask-script
```

manage.py

```
#!/usr/bin/env python
import os
from app import create_app
from flask.ext.script import Manager

app = create_app(os.getenv('FLASK_CONFIG') or 'default')
manager = Manager(app)

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

shell

```
(venv) $ python manage.py runserver
```

Avoiding Circular Dependencies

- Frequent problem with Flask applications that are split in modules.
- Most common instance:
  - Blueprint instance is created in `__init__.py`
  - Routes are defined in `routes.py` and need to import blueprint instance to get the `route` decorator.
  - `__init__.py` needs to import the routes to register them with the blueprint.
- Two tricks that help avoid this problem:
  - If routes are imported at the bottom of `__init__.py` then the circular dependency remains, but it does not cause errors.
  - Importing symbols inside the function that needs them can sometimes avoid circular dependencies.

Version 0.2

Twitter Bootstrap Integration

Flask-Bootstrap

- Flask extension that provides the base HTML document with Bootstrap libraries imported.
- The official Bootstrap documentation has lots of copy/paste ready examples: **http://getbootstrap.com**.

shell

```
(venv) $ pip install flask-bootstrap
```

app/__init__.py

```
from flask.ext.bootstrap import Bootstrap
bootstrap = Bootstrap()

def create_app(config_name):
    # ...
    bootstrap.init_app(app)
    # ...
    return app
```

Flask-Bootstrap (cont'd)

Jinja2's template inheritance feature is used to greatly simplify the integration with the Bootstrap libraries.

app/templates/talks/index.html

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

{% block title %}Talks{% endblock %}

{% block navbar %}
    ...
{% endblock %}

{% block content %}
<div class="container">
    <h1>Hello, World!</h1>
</div>
{% endblock %}
```

Version 0.3

Advanced Template Inheritance

Template Inheritance

An additional level of template inheritance is used to eliminate duplication of markup.

app/templates/base.html

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

{% block title %}Talks{% endblock %}

{% block navbar %}
    ...
{% endblock %}

{% block content %}
<div class="container">
    {% block page_content %}{% endblock %}
</div>
{% endblock %}
```

Template Inheritance (cont'd)

The application's templates inherit from the new base template.

app/templates/talks/index.html

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

{% block page_content %}
<h1>Hello, World!</h1>
{% endblock %}
```

app/templates/talks/user.html

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

{% block page_content %}
<div class="page-header">
    <h1>Hello, {{ username }}!</h1>
</div>
{% endblock %}
```

Version 0.4

Links

Links

Flask's `url_for()` function is used to generate application links.

app/templates/base.html

```
...
<a class="navbar-brand" href="{{ url_for('talks.index') }}">Talks</a>
...
<ul class="nav navbar-nav">
    <li><a href="{{ url_for('talks.index') }}">Home</a></li>
    <li><a href="{{ url_for('talks.user', username='miguel') }}">Profile</a></li>
</ul>
...
```

- For routes created with the `app.route` decorator the function name is used.
  - Example: `url_for('index')`
- For blueprint routes the blueprint name and the function name are separated by a dot.
  - Example: `url_for('talks.index')`
- If the blueprint name is not given it is taken from the running context.
  - Example: `url_for('.index')`
- Static files can be referenced with endpoint name `'static'` and a `filename` argument.
  - Example: `url_for('static', filename='styles.css')`

Version 0.5

Database

Flask-SQLAlchemy

- Flask-SQLAlchemy nicely integrates SQLAlchemy with Flask applications. Documentation links:
  - **http://pythonhosted.org/Flask-SQLAlchemy/**
  - **http://docs.sqlalchemy.org/**

shell

```
(venv) $ pip install flask-sqlalchemy
```

config.py

```
basedir = os.path.abspath(os.path.dirname(__file__))
class DevelopmentConfig(Config):
    # ...
    SQLALCHEMY_DATABASE_URI = os.environ.get('DEV_DATABASE_URL') or \
        'sqlite:///' + os.path.join(basedir, 'data-dev.sqlite')

class TestingConfig(Config):
    # ...
    SQLALCHEMY_DATABASE_URI = os.environ.get('TEST_DATABASE_URL') or \
        'sqlite:///' + os.path.join(basedir, 'data-test.sqlite')

class ProductionConfig(Config):
    # ...
    SQLALCHEMY_DATABASE_URI = os.environ.get('DATABASE_URL') or \
        'sqlite:///' + os.path.join(basedir, 'data.sqlite')
```

Flask-SQLAlchemy (cont'd)

app/__init__.py

```python
from flask.ext.sqlalchemy import SQLAlchemy
db = SQLAlchemy()

def create_app(config_name):
    # ...
    db.init_app(app)
    # ...
    return app
```

Model Definition

Models are defined as Python classes.

app/models.py

```python
from datetime import datetime
from . import db

class User(db.Model):
    __tablename__ = 'users'
    id = db.Column(db.Integer, primary_key=True)
    email = db.Column(db.String(64), nullable=False, unique=True, index=True)
    username = db.Column(db.String(64), nullable=False, unique=True, index=True)
    is_admin = db.Column(db.Boolean)
    password_hash = db.Column(db.String(128))
    name = db.Column(db.String(64))
    location = db.Column(db.String(64))
    bio = db.Column(db.Text())
    member_since = db.Column(db.DateTime(), default=datetime.utcnow)
    avatar_hash = db.Column(db.String(32))
```

Database creation

- The database can be created and destroyed from a Python shell.

shell

```
(venv) $ python manage.py shell
>>> from app import db
>>> db.create_all()
>>> db.drop_all()
```

- For medium to large applications it is strongly recommended to use a schema migration framework such as Flask-Migrate (Alembic).

Version 0.6

Password Hashing

Password Hashing

- Password hashing is very hard to get right if you do it on your own!
- Werkzeug provides secure password hashing and verification functions that use a unique random salt per password and PBKDF2 key derivation.

app/models.py

```python
from werkzeug.security import generate_password_hash, check_password_hash

class User(db.Model):
    # ...

    @property
    def password(self):
        raise AttributeError('password is not a readable attribute')

    @password.setter
    def password(self, password):
        self.password_hash = generate_password_hash(password)

    def verify_password(self, password):
        return check_password_hash(self.password_hash, password)
```

Version 0.7

Authentication Blueprint

More on Blueprints

Applications can have multiple blueprints.

app/auth/__init__.py

```python
from flask import Blueprint
auth = Blueprint('auth', __name__)
from . import routes
```

app/auth/routes.py

```python
from flask import render_template
from . import auth

@auth.route('/login')
def login():
    return render_template('auth/login.html')
```

More on Blueprints (cont'd)

Blueprints can be registered with a URL prefix.

app/__init__.py

```
def create_app(config_name):
    # ...
    from .auth import auth as auth_blueprint
    app.register_blueprint(auth_blueprint, url_prefix='/auth')
    # ...
    return app
```

Version 0.8

User Registration

User Registration

In this application users are registered from the command line with a custom Flask-Script command.

manage.py

```python
from app import db
from app.models import User

# ...

@manager.command
def adduser(email, username, admin=False):
    """Register a new user."""
    from getpass import getpass
    password = getpass()
    password2 = getpass(prompt='Confirm: ')
    if password != password2:
        import sys
        sys.exit('Error: passwords do not match.')
    db.create_all()
    user = User(email=email, username=username, password=password, is_admin=admin)
    db.session.add(user)
    db.session.commit()
    print('User {0} was registered successfully.'.format(username))
```

Custom Flask-Script Commands

Flask-Script uses introspection to generate the command line help messages.

shell

```
(venv) $ python manage.py adduser --help
usage: manage.py adduser [-h] [-a] email username

Register a new user.

positional arguments:
  email
  username

optional arguments:
  -h, --help   show this help message and exit
  -a, --admin
```

Example usage:

shell

```
(venv) $ python manage.py adduser john@example.com john
Password: <type password>
Confirm: <type password again>
User john was registered successfully.
```

Version 0.9

Login Form

Web Forms

- The Flask-WTF extension provides an excellent object-oriented abstraction for working with web forms.
  - The `Form` class represents a web form.
  - Subclasses of `Field` represent the form fields.
  - Validators can be applied to form fields.

shell

```
(venv) $ pip install flask-wtf
```

Typical form workflow:

pseudo-code

```
@route('/some-url', methods=['GET', 'POST'])
def foo():
    form = MyForm()
    if form.validate_on_submit():
        # process form data
        # values are in form.<field>.data
        return redirect(url_for('...'))
    # initialize form fields here
    # form.<field>.data = value
    return render_template('template.html', form=form)
```

Web Forms (cont'd)

Form definition:

app/auth/forms.py

```python
from flask.ext.wtf import Form
from wtforms import StringField, PasswordField, BooleanField, SubmitField
from wtforms.validators import Required, Length, Email

class LoginForm(Form):
    email = StringField('Email', validators=[Required(), Length(1, 64), Email()])
    password = PasswordField('Password', validators=[Required()])
    remember_me = BooleanField('Keep me logged in')
    submit = SubmitField('Log In')
```

Form usage:

app/auth/routes.py

```python
@auth.route('/login', methods=['GET', 'POST'])
def login():
    form = LoginForm()
    if form.validate_on_submit():
        pass
    return render_template('auth/login.html', form=form)
```

Rendering Forms with Flask-Bootstrap

Flask-Bootstrap includes Jinja2 macros that simplify the rendering of Flask-WTF forms.

app/templates/auth/login.html

```
{% import "bootstrap/wtf.html" as wtf %}

{% block page_content %}
...
{{ wtf.quick_form(form) }}
{% endblock %}
```

Version 0.10

Logging In

Flask-Login

- Flask-Login keeps track of the logged in user in the user session.
- It makes no assumptions about how users are represented, stored or logged in.

shell

```
(venv) $ pip install flask-login
```

app/__init__.py

```python
from flask.ext.login import LoginManager
login_manager = LoginManager()
login_manager.login_view = 'auth.login'

def create_app(config_name):
    # ...
    login_manager.init_app(app)
    # ...
    return app
```

Flask-Login (cont'd)

- The user class needs to inherit from `UserMixin`, or else implement the following methods:
  - `is_authenticated()`
  - `is_active()`
  - `is_anonymous()`
  - `get_id()`
- The application must register a user loader callback.
- Request handlers can be protected with the `login_required` decorator.

app/models.py

```python
from flask.ext.login import UserMixin
from . import db, login_manager

class User(UserMixin, db.Model):
    # ...

@login_manager.user_loader
def load_user(user_id):
    return User.query.get(int(user_id))
```

Logging Users In

Step 1: Load user by the email address given in the form.

app/models.py

```python
from flask import render_template, redirect, request, url_for, flash
from flask.ext.login import login_user
from ..models import User

@auth.route('/login', methods=['GET', 'POST'])
def login():
    form = LoginForm()
    if form.validate_on_submit():
        user = User.query.filter_by(email=form.email.data).first()
        if user is None or not user.verify_password(form.password.data):
            flash('Invalid email or password.')
            return redirect(url_for('.login'))
        login_user(user, form.remember_me.data)
        return redirect(request.args.get('next') or url_for('talks.index'))
    return render_template('auth/login.html', form=form)
```

Logging Users In (cont'd)

Step 2: Verify that the email and password given in the form are valid.

app/models.py

```python
from flask import render_template, redirect, request, url_for, flash
from flask.ext.login import login_user
from ..models import User

@auth.route('/login', methods=['GET', 'POST'])
def login():
    form = LoginForm()
    if form.validate_on_submit():
        user = User.query.filter_by(email=form.email.data).first()
        if user is None or not user.verify_password(form.password.data):
            flash('Invalid email or password.')
            return redirect(url_for('.login'))
        login_user(user, form.remember_me.data)
        return redirect(request.args.get('next') or url_for('talks.index'))
    return render_template('auth/login.html', form=form)
```

Logging Users In (cont'd)

Step 3: Log the user in and redirect to the home page.

app/models.py

```
from flask import render_template, redirect, request, url_for, flash
from flask.ext.login import login_user
from ..models import User

@auth.route('/login', methods=['GET', 'POST'])
def login():
    form = LoginForm()
    if form.validate_on_submit():
        user = User.query.filter_by(email=form.email.data).first()
        if user is None or not user.verify_password(form.password.data):
            flash('Invalid email or password.')
            return redirect(url_for('.login'))
        login_user(user, form.remember_me.data)
        return redirect(request.args.get('next') or url_for('talks.index'))
    return render_template('auth/login.html', form=form)
```

Logging Users In (cont'd)

Step 4: Redirect to the "next" page if available.

app/models.py

```
from flask import render_template, redirect, request, url_for, flash
from flask.ext.login import login_user
from ..models import User

@auth.route('/login', methods=['GET', 'POST'])
def login():
    form = LoginForm()
    if form.validate_on_submit():
        user = User.query.filter_by(email=form.email.data).first()
        if user is None or not user.verify_password(form.password.data):
            flash('Invalid email or password.')
            return redirect(url_for('.login'))
        login_user(user, form.remember_me.data)
        return redirect(request.args.get('next') or url_for('talks.index'))
    return render_template('auth/login.html', form=form)
```

Flashed Messages

app/templates/base.html

```
...
{% for message in get_flashed_messages() %}
<div class="alert alert-warning">
    <button type="button" class="close" data-dismiss="alert">×</button>
    {{ message }}
</div>
{% endfor %}
...
```

Access to the logged-in user

- The currently logged in user can be accessed through the `current_user` context global.
- Navigation bar links can be specifically created for the current user:
  - A "Profile" link points to the logged-in user's profile page.
  - A "Presenter Login" link is shown if there is no logged-in user.

app/templates/base.html

```
{% if current_user.is_authenticated() %}
<li>
    <a href="{{ url_for('talks.user', username=current_user.username) }}">Profile</a>
</li>
{% endif %}
...
{% if not current_user.is_authenticated() %}
<li><a href="{{ url_for('auth.login') }}">Presenter Login</a></li>
{% endif %}
```

Access to the logged-in user (cont'd)

Templates can also access the logged-in user:

app/templates/talks/index.html

```
{% if current_user.is_authenticated() %}
<h1>Hello, {{ current_user.username }}!</h1>
{% endif %}
```

User profile route now works with real users:

app/talks/routes.py

```
from ..models import User

@talks.route('/user/<username>')
def user(username):
    user = User.query.filter_by(username=username).first_or_404()
    return render_template('talks/user.html', user=user)
```

app/templates/talks/user.html

```
<h1>Hello, {{ user.username }}!</h1>
```

Version 0.11

Logging Out

Logging Out

The `login_required` decorator prevents access to anonymous users.

app/auth/routes.py

```
from flask.ext.login import login_user, logout_user, login_required

@auth.route('/logout')
@login_required
def logout():
    logout_user()
    flash('You have been logged out.')
    return redirect(url_for('talks.index'))
```

The logged-in state can be used to show log in or out links.

app/templates/base.html

```
...
{% if not current_user.is_authenticated() %}
<li><a href="{{ url_for('auth.login') }}">Presenter Login</a></li>
{% else %}
<li><a href="{{ url_for('auth.logout') }}">Logout</a></li>
{% endif %}
...
```

Version 0.12

User Avatars

Gravatar Mini-Reference

- The Gravatar service is the easiest way to include user avatar images.
- Given `HASH=md5(email_address)`, the URL for the avatar image for the email address is:
  - **http://www.gravatar.com/avatar/HASH** (normal version)
  - **https://secure.gravatar.com/avatar/HASH** (secure version)
- The query string can include optional arguments:
  - `s=N` where N is the size of the image in pixels.
  - `d=D` where D is the name of an image generator to be used for users that don't have an avatar registered.
  - `r=R` where R is the image rating (g, pg, etc.)

Example avatar markup:

```
<img src="http://www.gravatar.com/avatar/205e460b479e2e5b48aec07710c08d50?s=128&r=g">
```

User Avatars

Avatar URL generation is encapsulated in the `User` model.

app/models.py

```
class User(UserMixin, db.Model):
    # ...
    def __init__(self, **kwargs):
        super(User, self).__init__(**kwargs)
        if self.email is not None and self.avatar_hash is None:
            self.avatar_hash = hashlib.md5(
                self.email.encode('utf-8')).hexdigest()

    def gravatar(self, size=100, default='identicon', rating='g'):
        if request.is_secure:
            url = 'https://secure.gravatar.com/avatar'
        else:
            url = 'http://www.gravatar.com/avatar'
        hash = self.avatar_hash or \
               hashlib.md5(self.email.encode('utf-8')).hexdigest()
        return '{url}/{hash}?s={size}&d={default}&r={rating}'.format(
            url=url, hash=hash, size=size, default=default, rating=rating)
```

User Avatars (cont'd)

Gravatar URLs can be requested directly from templates.

app/templates/talks/user.html

```
...
<div class="page-header user-profile">
    <div class="user-avatar"><img src="{{ user.gravatar(128) }}"></div>
    <h1>{{ user.username }}</h1>
</div>
...
```

Version 0.13

User Profile Editor

Edit Profile Route

- The `ProfileForm` class defines the three user editable fields.
- The `login_required` decorator prevents access to regular users.
- Database session needs the "real" `current_user` object, not the context global proxy.

app/talks/routes.py

```python
@talks.route('/profile', methods=['GET', 'POST'])
@login_required
def profile():
    form = ProfileForm()
    if form.validate_on_submit():
        current_user.name = form.name.data
        current_user.location = form.location.data
        current_user.bio = form.bio.data
        db.session.add(current_user._get_current_object())
        db.session.commit()
        flash('Your profile has been updated.')
        return redirect(url_for('talks.user', username=current_user.username))
    form.name.data = current_user.name
    form.location.data = current_user.location
    form.bio.data = current_user.bio
    return render_template('talks/profile.html', form=form)
```

Version 0.14

Adding Talks

Talk Model

SQLAlchemy allows relationships between models to be easily defined.

app/models.py

```python
class Talk(db.Model):
    __tablename__ = 'talks'
    id = db.Column(db.Integer, primary_key=True)
    title = db.Column(db.String(128), nullable=False)
    description = db.Column(db.Text)
    slides = db.Column(db.Text())
    video = db.Column(db.Text())
    venue = db.Column(db.String(128))
    venue_url = db.Column(db.String(128))
    date = db.Column(db.DateTime())
    user_id = db.Column(db.Integer, db.ForeignKey('users.id'))

class User(UserMixin, db.Model):
    # ...
    talks = db.relationship('Talk', backref='author', lazy='dynamic')
```

Talk Form

The `Optional()` validator enables validators to work on fields that are allowed to be empty.

app/talks/forms.py

```python
class TalkForm(Form):
    title = StringField('Title', validators=[Required(), Length(1, 128)])
    description = TextAreaField('Description')
    slides = StringField('Slides Embed Code (450 pixels wide)')
    video = StringField('Video Embed Code (450 pixels wide)')
    venue = StringField('Venue', validators=[Required(), Length(1, 128)])
    venue_url = StringField('Venue URL',
                            validators=[Optional(), Length(1, 128), URL()])
    date = DateField('Date')
    submit = SubmitField('Submit')
```

Add Talk Route

The relationship handles the foreign keys automatically.

app/talks/routes.py

```
@talks.route('/new', methods=['GET', 'POST'])
@login_required
def new_talk():
    form = TalkForm()
    if form.validate_on_submit():
        talk = Talk(title=form.title.data,
                    description=form.description.data,
                    slides=form.slides.data,
                    video=form.video.data,
                    venue=form.venue.data,
                    venue_url=form.venue_url.data,
                    date=form.date.data,
                    author=current_user)
        db.session.add(talk)
        db.session.commit()
        flash('The talk was added successfully.')
        return redirect(url_for('.index'))
    return render_template('talks/edit_talk.html', form=form)
```

Version 0.15

Timelines

Database Queries

- Queries are used to obtain data from the database.
- The one-to-many relationship between users and talks is used to obtain list of talks by a user to show in the profile page.

app/talks/routes.py

```python
@talks.route('/')
def index():
    talk_list = Talk.query.order_by(Talk.date.desc()).all()
    return render_template('talks/index.html', talks=talk_list)

@talks.route('/user/<username>')
def user(username):
    user = User.query.filter_by(username=username).first_or_404()
    talk_list = user.talks.order_by(Talk.date.desc()).all()
    return render_template('talks/user.html', user=user, talks=talk_list)
```

Sub-Templates

- Sub-templates are used to avoid repetition.

app/templates/talks/{index|user}.html

```
{% include "talks/_talks.html" %}
```

app/templates/talks/_talks.html

```
<ul class="talk-list">
{% for talk in talks %}
    <li class="talk">{% include "talks/_talk_header.html" %}</li>
{% endfor %}
</ul>
```

app/templates/talks/_talk_header.html

```
<div class="talk-header">
    <h2>{{ talk.title }}</h2><h3>{{ talk.description }}</h3>
    <p>
        <a href="{{ url_for('talks.user', username=talk.author.username) }}">{{ talk.author.username }}</a>
        at
        {% if talk.venue_url %}
            <a href="{{ talk.venue_url }}">{{ talk.venue }}</a>
        {% else %}
            {{ talk.venue }}
        {% endif %}
    </p>
</div>
```

Version 0.16

Date Handling

Rendering Dates and Times

Flask-Moment is a convenient extension that simplifies the use of   to render dates in the browser.

shell

```
(venv) $ pip install flask-moment
```

app/__init__.py

```
from flask.ext.moment import Moment
moment = Moment()

def create_app(config_name):
    # ...
    moment.init_app(app)
    # ...
    return app
```

app/templates/base.html

```
{% block scripts %}
{{ super() }}
{{ moment.include_moment() }}
{% endblock %}
```

app/templates/talks/_talk_header.html

```
...
on {{ moment(talk.date, local=True).format('LL') }}.
```

Version 0.17

Talk Page

Talk Route

- The route simply loads the requested talk from the database and passes it to the template for rendering.
- The `get_or_404()` method of Flask-SQLAlchemy will automatically return a response with 404 status code to the client if the requested talk is not found.

app/talks/routes.py

```python
@talks.route('/talk/<int:id>')
def talk(id):
    talk = Talk.query.get_or_404(id)
    return render_template('talks/talk.html', talk=talk)
```

Talk Templates

- The talk header template from the talk timelines is reused here.
- The `safe` Jinja2 filter suppresses escaping. Only use for trusted content!

app/templates/talks/talk.html

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

{% block page_content %}
<div class="page-header">
    {% include "talks/_talk_header.html" %}
</div>
<div class="talk-body">
    {% if talk.video %}
        <div class="talk-video">
            {{ talk.video | safe }}
        </div>
    {% endif %}
    {% if talk.slides %}
        <div class="talk-slides">
            {{ talk.slides | safe }}
        </div>
    {% endif %}
</div>
{% endblock %}
```

Talk Templates (cont'd)

The talk header template is updated to display the talk title as a link to the corresponding talk page.

app/templates/talks/_talk_header.html

```
<h2><a href="{{ url_for('talks.talk', id=talk.id) }}">{{ talk.title }}</a></h2>
<h3>{{ talk.description }}</h3>
```

Version 0.18

Talk Editor

Talk Editor Routes

- Additional validation is done to ensure that only authorized people edit talks.
- Flask's `abort()` function is used to return an error response.
- Data is moved to and from the form using helper methods.

app/talks/routes.py

```python
@talks.route('/edit/<int:id>', methods=['GET', 'POST'])
@login_required
def edit_talk(id):
    talk = Talk.query.get_or_404(id)
    if not current_user.is_admin and talk.author != current_user:
        abort(403)
    form = TalkForm()
    if form.validate_on_submit():
        form.to_model(talk)
        db.session.add(talk)
        db.session.commit()
        flash('The talk was updated successfully.')
        return redirect(url_for('.talk', id=talk.id))
    form.from_model(talk)
    return render_template('talks/edit_talk.html', form=form)
```

Model/Form Data Exchange

app/talks/forms.py

```python
class TalkForm(Form):
    # ...

    def from_model(self, talk):
        self.title.data = talk.title
        self.description.data = talk.description
        self.slides.data = talk.slides
        self.video.data = talk.video
        self.venue.data = talk.venue
        self.venue_url.data = talk.venue_url
        self.date.data = talk.date

    def to_model(self, talk):
        talk.title = self.title.data
        talk.description = self.description.data
        talk.slides = self.slides.data
        talk.video = self.video.data
        talk.venue = self.venue.data
        talk.venue_url = self.venue_url.data
        talk.date = self.date.data
```

Version 0.19

User Comments

Markdown Support

- Comments are entered using Markdown syntax.
- Markdown and Bleach are used for Markdown rendering on the server.
- Flask-PageDown renders Markdown live in the browser.
- For security reasons the browser only sends Markdown source to the server.

shell

```
(venv) $ pip install flask-pagedown markdown bleach
```

app/__init__.py

```
from flask.ext.pagedown import PageDown
pagedown = PageDown()

def create_app(config_name):
    # ...
    pagedown.init_app(app)
    # ...
    return app
```

app/templates/talks/talk.html

```
{% block scripts %}
{{ super() }}
{{ pagedown.include_pagedown() }}
{% endblock %}
```

Comment Model

- For each comment the Markdown source and the rendered HTML are stored.
- The `Comment` model has two one-to-many relationships from `User` and `Talk`.

app/models.py

```python
class Comment(db.Model):
    __tablename__ = 'comments'
    id = db.Column(db.Integer, primary_key=True)
    body = db.Column(db.Text)
    body_html = db.Column(db.Text)
    timestamp = db.Column(db.DateTime, index=True, default=datetime.utcnow)
    author_id = db.Column(db.Integer, db.ForeignKey('users.id'))
    author_name = db.Column(db.String(64))
    author_email = db.Column(db.String(64))
    notify = db.Column(db.Boolean, default=True)
    approved = db.Column(db.Boolean, default=False)
    talk_id = db.Column(db.Integer, db.ForeignKey('talks.id'))

class User(UserMixin, db.Model):
    # ...
    comments = db.relationship('Comment', lazy='dynamic', backref='author')

class Talk(db.Model):
    # ...
    comments = db.relationship('Comment', lazy='dynamic', backref='talk')
```

Server-Side Markdown

- A SQLAlchemy change event triggers the Markdown rendering.
- The Markdown text is rendered in three steps:
    - The text is rendered to HTML.
    - Bleach is used to filter any HTML tags not in the white list.
    - Bleach's `linkify()` function is used to convert any plain URLs to links.

app/models.py

```python
from markdown import markdown
import bleach

class Comment(db.Model):
    # ...
    @staticmethod
    def on_changed_body(target, value, oldvalue, initiator):
        allowed_tags = ['a', 'abbr', 'acronym', 'b', 'blockquote', 'code',
                        'em', 'i', 'li', 'ol', 'pre', 'strong', 'ul',
                        'h1', 'h2', 'h3', 'p']
        target.body_html = bleach.linkify(bleach.clean(
            markdown(value, output_format='html'),
            tags=allowed_tags, strip=True))

db.event.listen(Comment.body, 'set', Comment.on_changed_body)
```

Comment Forms

- Two forms are used, one for presenters and administrators, another for regular users.
- Flask-PageDown's `PageDownField` is used in place of a regular text area field.

app/talks/forms.py

```python
from flask.ext.pagedown.fields import PageDownField

class PresenterCommentForm(Form):
    body = PageDownField('Comment', validators=[Required()])
    submit = SubmitField('Submit')

class CommentForm(Form):
    name = StringField('Name', validators=[Required(), Length(1, 64)])
    email = StringField('Email', validators=[Required(), Length(1, 64), Email()])
    body = PageDownField('Comment', validators=[Required()])
    notify = BooleanField('Notify when new comments are posted', default=True)
    submit = SubmitField('Submit')
```

Talk Route with Comment Support

- The appropriate comment form for the current user is used.
- On form submission a comment is created with its `approved` field set to `False` for regular users.

app/talks/routes.py

```
@talks.route('/talk/<int:id>', methods=['GET', 'POST'])
def talk(id):
    talk = Talk.query.get_or_404(id)
    comment = None
    if current_user.is_authenticated():
        form = PresenterCommentForm()
        if form.validate_on_submit():
            comment = Comment(body=form.body.data, talk=talk,
                              author=current_user,
                              notify=False, approved=True)
    else:
        form = CommentForm()
        if form.validate_on_submit():
            comment = Comment(body=form.body.data, talk=talk,
                              author_name=form.name.data,
                              author_email=form.email.data,
                              notify=form.notify.data, approved=False)
    # ...
```

Talk Route with Comment Support (cont'd)

- The flashed message is different for approved or not approved messages.
- A non-existant fragment is used in the redirect to reset scroll position of the page.
- The comment list is sorted by date in ascending order and sent to the template.

app/talks/routes.py

```python
@talks.route('/talk/<int:id>', methods=['GET', 'POST'])
def talk(id):
    # ...
    if comment:
        db.session.add(comment)
        db.session.commit()
        if comment.approved:
            flash('Your comment has been published.')
        else:
            flash('Your comment will be published after it is reviewed by the presenter.')
        return redirect(url_for('.talk', id=talk.id) + '#top')
    comments = talk.comments.order_by(Comment.timestamp.asc()).all()
    return render_template('talks/talk.html', talk=talk, form=form, comments=comments)
```

Comment Templates

- The list of comments is rendered below the talk slides and/or video.
- A sub-template organization similar to the one for tasks is used.
- A form to enter a new comment is rendered below the comment list.
- A fragment is given as a form action, so that the scroll position is preserved.

app/templates/talks/talk.html

```
{% import "bootstrap/wtf.html" as wtf %}
...
{% if comments %}
    <h3>Comments</h3>
    {% include "talks/_comments.html" %}
{% endif %}
<h3 id="comment-form">Write a comment</h3>
{{ wtf.quick_form(form, action='#comment-form') }}
...
```

Comment Templates (cont'd)

- Jinja2's `set` directive is used to pass a variable to the included template.
- The loop index is available as `loop.index`

app/templates/talks/_comments.html

```
<ul class="comment-list">
{% for comment in comments %}
    {% set index = loop.index %}
    <li class="comment">
        {% include "talks/_comment.html" %}
    </li>
{% endfor %}
</ul>
```

Comment Templates (cont'd)

- The commenter's email address is shown only to the talk author or administrator.
- Flask-Moment is used to render the comment timestamp in "time ago" mode.

app/templates/talks/_comment.html

```
<p>
    {% if comment.author %}
        <span class="label label-primary">#{{ index }}</span>
        <a href="{{ url_for('.user', username=comment.author.username) }}">
            {{ comment.author.username }}
        </a>
    {% else %}
        <span class="label label-default">#{{ index }}</span>
        <b>{{ comment.author_name }}</b>
        {% if current_user.is_authenticated() and
                 (talk.author == current_user or current_user.is_admin) %}
        (<a href="mailto:{{ comment.author_email }}">{{ comment.author_email }}</a>)
        {% endif %}
    {% endif %}
    commented {{ moment(comment.timestamp).fromNow() }}:
</p>
<div class="comment-body">
    {{ comment.body_html | safe }}
</div>
```

Version 0.20

Comment Moderation

APIs Mini-Reference

- The REpresentational State Transfer (REST) model is typically used for Web application APIs due to its close ties to the HTTP protocol.
- Request URLs name resources, the items of interest in the application's domain.
- The request method determines the action to carry out:
  - `POST`: create a new resource. This is the C in CRUD.
  - `GET`: read a resource or collection of resources. This is the R in CRUD.
  - `PUT`: update a resource. This is the U in CRUD.
  - `DELETE`: delete a resource. This is the D in CRUD.
- Resources are serialized and sent in the bodies of requests and responses as needed. JSON and XML are common serialization formats.
- Flask has native support for RESTful APIs through its request routing.

API Blueprint

- The API endpoints are defined in a separate blueprint.
- The blueprint is versioned, to leave room for expansion.
- Each route implements a resource and method combination.
- This API implements "approve" and "delete" comment moderation operations.

app/api_1_0/__init__.py

```python
from flask import Blueprint
api = Blueprint('api', __name__)
from . import comments, errors
```

app/__init__.py

```python
def create_app(config_name):
    # ...
    from .api_1_0 import api as api_blueprint
    app.register_blueprint(api_blueprint, url_prefix='/api/1.0')
    # ...
    return app
```

API endpoints

- API routes return a JSON response using `jsonify()`.

app/api_1_0/comments.py

```python
from flask import jsonify
from . import api
from .errors import bad_request

@api.route('/comments/<int:id>', methods=['PUT'])
def approve_comment(id):
    comment = Comment.query.get_or_404(id)
    # TODO: ensure user has permission to approve comment
    if comment.approved:
        return bad_request('Comment is already approved.')
    comment.approved = True
    db.session.add(comment)
    db.session.commit()
    return jsonify({'status': 'ok'})

@api.route('/comments/<int:id>', methods=['DELETE'])
def delete_comment(id):
    # ...
```

Error Handling

- Helper functions are implemented for error responses.
- All responses return JSON.

app/api_1_0/errors.py

```python
from flask import jsonify

def bad_request(message):
    response = jsonify({'status': 'bad request', 'message': message})
    response.status_code = 400
    return response

def unauthorized(message):
    response = jsonify({'status': 'unauthorized', 'message': message})
    response.status_code = 401
    return response

def forbidden(message):
    response = jsonify({'status': 'forbidden', 'message': message})
    response.status_code = 403
    return response

def not_found(message):
    response = jsonify({'status': 'not found', 'message': message})
    response.status_code = 404
    return response
```

Error Handling (cont'd)

- A custom error handler is implemented to catch 404 exceptions thrown by Flask-SQLAlchemy.
- Other exceptions can be handled in the same way.

app/api_1_0/errors.py

```
# ...

@api.errorhandler(404)
def not_found_handler(e):
    return not_found('resource not found')
```

Authentication

- All API requests must come with a valid authentication token.
- Tokens are generated and validated in the `User` model.
- Package itsdangerous is used to generate cryptographically secure tokens.

app/models.py

```python
from itsdangerous import TimedJSONWebSignatureSerializer as Serializer

class User(UserMixin, db.Model):
    # ...
    def get_api_token(self, expiration=300):
        s = Serializer(current_app.config['SECRET_KEY'], expiration)
        return s.dumps({'user': self.id}).decode('utf-8')

    @staticmethod
    def validate_api_token(token):
        s = Serializer(current_app.config['SECRET_KEY'])
        try:
            data = s.loads(token)
        except:
            return None
        id = data.get('user')
        if id:
            return User.query.get(id)
        return None
```

Authentication (cont'd)

- Token verification happens in a `before_request` handler for the blueprint.
- The user is obtained from the token and recorded in the `g` context global.

app/api_1_0/__init__.py

```python
from flask import request, g
from . import errors

@api.before_request
def before_api_request():
    if request.json is None:
        return errors.bad_request('Invalid JSON in body.')
    token = request.json.get('token')
    if not token:
        return errors.unauthorized('Authentication token not provided.')
    user = User.validate_api_token(token)
    if not user:
        return errors.unauthorized('Invalid authentication token.')
    g.current_user = user
```

Authentication (cont'd)

- API routes access the user making the requests as `g.current_user`.

app/api_1_0/comments.py

```
@api.route('/comments/<int:id>', methods=['PUT'])
def approve_comment(id):
    comment = Comment.query.get_or_404(id)
    if comment.talk.author != g.current_user and \
            not g.current_user.is_admin:
        return forbidden('You cannot modify this comment.')
    # ...

@api.route('/comments/<int:id>', methods=['DELETE'])
def delete_comment(id):
    comment = Comment.query.get_or_404(id)
    if comment.talk.author != g.current_user and \
            not g.current_user.is_admin:
        return forbidden('You cannot modify this comment.')
    # ...
```

Moderation Queries

- The models have helper methods for common queries needed to perform comment moderation.
- A database join operation is performed to obtain all the comments to be moderated in all the talks that belong to a speaker.

app/models.py

```python
class Talk(db.Model):
    # ...
    def approved_comments(self):
        return self.comments.filter_by(approved=True)

class Comment(db.Model):
    # ...
    @staticmethod
    def for_moderation():
        return Comment.query.filter(Comment.approved == False)

class User(UserMixin, db.Model):
    # ...
    def for_moderation(self, admin=False):
        if admin and self.is_admin:
            return Comment.for_moderation()
        return Comment.query.join(Talk, Comment.talk_id == Talk.id).\
            filter(Talk.author == self).filter(Comment.approved == False)
```

Moderation Routes

- Moderation for speakers and admins are handled separately.
  - Speakers can only moderate comments for their talks.
  - Admins can moderate comments for all talks.

app/talks/routes.py

```python
@talks.route('/moderate')
@login_required
def moderate():
    comments = current_user.for_moderation().order_by(Comment.timestamp.asc())
    return render_template('talks/moderate.html', comments=comments)

@talks.route('/moderate-admin')
@login_required
def moderate_admin():
    if not current_user.is_admin:
        abort(403)
    comments = Comment.for_moderation().order_by(Comment.timestamp.asc())
    return render_template('talks/moderate.html', comments=comments)
```

Moderation Template

- The JavaScript API client is included in the moderation and talk pages so that comments can be moderated in both.
- The client-side implementation is available on the github repository.

app/templates/talks/moderate.html

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

{% block page_content %}
<h2>Comment moderation</h2>
<ul class="comment-list">
{% for comment in comments %}
    {% set talk = comment.talk %}
    <li class="comment">
        <p>In <a href="{{ url_for('talks.talk', id=talk.id) }}">{{ talk.title }}</a></p>
        {% include "talks/_comment.html" %}
    </li>
{% endfor %}
</ul>
{% endblock %}

{% block scripts %}
{{ super() }}
{% include "_api_client.html" %}
{% endblock %}
```

Version 0.21

Pagination

Pagination

- Page sizes are specified as configuration options.
- Page number is given as a query string argument.
- Flask-SQLAlchemy's `paginate()` is applied to database queries.
- The `Pagination` object is passed to the template.

config.py

```
class Config:
    # ...
    TALKS_PER_PAGE = 50
    COMMENTS_PER_PAGE = 100
```

app/talks/routes.py

```
@talks.route('/')
def index():
    page = request.args.get('page', 1, type=int)
    pagination = Talk.query.order_by(Talk.date.desc()).paginate(
        page, per_page=current_app.config['TALKS_PER_PAGE'],
        error_out=False)
    talk_list = pagination.items
    return render_template('talks/index.html', talks=talk_list, pagination=pagination)
```

Pagination (cont'd)

- Bootstrap pager markup is used for "next" and "previous" links.
- Flask-SQLAlchemy's pagination object has the previous and next page numbers.

app/templates/talks/index.html

```html
<ul class="pager">
    {% if pagination.has_prev %}
    <li class="previous">
        <a href="{{ url_for('talks.index', page=pagination.prev_num) }}">
            ← Newer
        </a>
    </li>
    {% else %}
    <li class="previous disabled"><a href="#">← Newer</a></li>
    {% endif %}
    {% if pagination.has_next %}
    <li class="next">
        <a href="{{ url_for('talks.index', page=pagination.next_num) }}">
            Older →
        </a>
    </li>
    {% else %}
    <li class="next disabled"><a href="#">Older →</a></li>
    {% endif %}
</ul>
```

Version 0.22

Bonus Feature: Emails

Emails

- The Flask-Mail extension is used to send emails to users.
- The default configuration uses a gmail account to send email. This is sufficient for development, but a dedicated email server must be used in production.
- Two helper functions are added:
    - `send_author_notification()` sends an email to the author of a talk when new comments require moderation.
    - `send_comment_notification()` sends an email to all the previous commenters in the talk.
- To avoid sending too many emails a queue collects pending emails.
- A background thread flushes the email queue at regular intervals.
- Emails to commenters include an link with an unsubscribe token. When clicked, the comment for that user is flagged so that no new comment notifications are sent to that address.
- The changes for this feature are on github.

Version 0.23

Unit Tests

Unit Tests

- Business logic should be in models or service classes and tested outside of a running application.
- Small and focused unit tests are easier to maintain than large end-to-end tests.
- Write tests that are simple and straightforward, you do not want to have bugs in your tests!
- Test APIs with the Flask test client.
- Only use end-to-end testing tools such as Selenium for tests that cannot be implemented with simpler methods.
- Only test your application code, assume the libraries that you use are well tested.
- Use code coverage to find out what your tests are missing.

A custom Flask-Script command can run all the tests and generate a coverage report:

shell

```
(venv) $ pip install nose coverage
```

manage.py

```
@manager.command
def test():
    from subprocess import call
    call(['nosetests', '-v',
          '--with-coverage', '--cover-package=app', '--cover-branches',
          '--cover-erase', '--cover-html', '--cover-html-dir=cover'])
```

Version 0.24

Production Mode

Logging

- Flask's logger does not have any handlers in production mode.
- `logging.SMTPHandler` is added to send application errors by email to a designated administrator. Users see a status 500 error page.
- `logging.SysLogHandler` is added to send regular logs to syslog (for Unix servers)
- On Windows `NTEventLogHandler` or `FileHandler` can be used instead.

Environment Variables

- Import environment variables from `.env` file, if present.
- Due to the sensitive information, this file needs to be created manually for each deployed system, do not put it under version control.
- Do not use gmail as email server, install sendmail, postfix, etc. or use a third party service.
- Example:

.env

```
FLASK_CONFIG=production
SECRET_KEY=you-will-never-guess!
MAIL_USERNAME=<your-smtp-username>
MAIL_PASSWORD=<your-smtp-password>
MAIL_SENDER=admin@yourdomain.com
MAIL_ERROR_RECIPIENT=errors@yourdomain.com
DATABASE_URL=mysql://user:password@yourdomain/talks
```

We are done!

Questions?

Thank You!

twitter	@miguelgrinberg
www	miguelgrinberg.com
github	github.com/miguelgrinberg