UNIT 1

FLASK WEB IMPLEMENTATION

Flask Web in Python

Flask is a lightweight WSGI web application framework. It is designed to make getting started quick and easy, with the ability to scale up to complex applications. It began as a simple wrapper around Werkzeug and Jinja and has become one of the most popular Python web application frameworks.

Basic Application Structure

Initiazation

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

Example hello.py: A complete Flask applicatio

```
from flask import Flask
app = Flask(__name__)
@app.route('/')
def index():
return '<h1>Hello World!</h1>'
```

Request Hooks

before_request

Registers a function to run before each request.

before_first_request

Registers a function to run only before the first request is handled. This can be a convenient way to add server initialization tasks.

after_request

Registers a function to run after each request, but only if no unhandled exceptions occurred.

teardown_request

Registers a function to run after each request, even if unhandled exceptions occurred.

Responses

When a view function needs to respond with a different status code, it can add the numeric code as a second return value after the response text. For example, the following view function returns a 400 status code, the code for a bad request

```
def index():
return '<h1>Bad Request</h1>', 400
```

Templates

In its simplest form, a Jinja2 template is a file that contains the text of a response. Example
shows a Jinja2 template that matches the response of the index() view

fple templates/index.html: Jinja2 template
<h1>Hello World!</h1>

Control Structure

```
 {% if user %}
Hello, {{ user }}!
{% else %}
Hello, Stranger!
{% endif %}
```

Links

Any application that has more than one route will invariably need to include links that connect the different pages, such as in a navigation bar.

Example

templates/base.html

```
{% block head %}
{{ super() }}
<link rel="shortcut icon" href="{{ url_for('static', filename='favicon.ico') }}"
type="image/x-icon">
<link rel="icon" href="{{ url_for('static', filename='favicon.ico') }}"
type="image/x-icon">
{% endblock %}
```

Web Form

Example hello.py: Flask-WTF configuration

```
app = Flask(__name__)
app.config['SECRET_KEY'] = 'hard to guess string'
```

Form Classes

When using Flask-WTF, each web form is represented in the server by a class that inherits from the class FlaskForm.The class defines the list of fields in the form

Example hello.py: form class definition

```
from flask_wtf import FlaskForm
from wtforms import StringField, SubmitField
from wtforms.validators import DataRequired
class NameForm(FlaskForm):
    name = StringField('What is your name?', validators=[DataRequired()])
    submit = SubmitField('Submit')
```

Databases

A database stores application data in an organized way. The application then issues inherits from the class FlaskForm.The class defines the list of fields in the form.

Python Database Frameworks

Python has packages for most database engines, both open source and commercial. Flask puts no restrictions on what database packages can be used, so we can work with MySQL, Postgres, SQLite, Redis, MongoDB, CouchDB, or DynamoDB if any of these is your favorite.

Factors

Ease of Use
Performance
Portability
Flask integration

Model Definition

Example hello.py

```
class Role(db.Model):
    __tablename__ = 'roles'
        id = db.Column(db.Integer, primary_key=True)
       name = db.Column(db.String(64), unique=True)
      def __repr__(self):
        return '<Role %r>' % self.name
class User(db.Model):
       __tablename__ = 'users'
        id = db.Column(db.Integer, primary_key=True)
        username = db.Column(db.String(64), unique=True, index=True)
       def __repr__(self):
            return '<User %r>' % self.username
```

Relationships

Relational databases establish connections between rows in different tables through the use of relationships.

```
class Role(db.Model):
# ...
   users = db.relationship('User', backref='role')
   class User(db.Model):
# ...
   role_id = db.Column(db.Integer, db.ForeignKey('roles.id'))
```

Email

Email Support with Flask-Mail

Install: (venv) $ pip install flask-mail

Flask-Mail initialization

Example hello.py

```
from flask_mail import Mail
mail = Mail(app)
```

Example hello.py: email example

```
# ...
app.config['FLASKY_ADMIN'] = os.environ.get('FLASKY_ADMIN')
# ...
@app.route('/', methods=['GET', 'POST'])
def index():
form = NameForm()
if form.validate_on_submit():
user = User.query.filter_by(username=form.name.data).first()
if user is None:
user = User(username=form.name.data)
db.session.add(user)
session['known'] = False
if app.config['FLASKY_ADMIN']:
send_email(app.config['FLASKY_ADMIN'], 'New User',
'mail/new_user', user=user)
else:
session['known'] = True
session['name'] = form.name.data
form.name.data = ''
return redirect(url_for('index'))
return render_template('index.html', form=form, name=session.get('name'),
known=session.get('known', False))
```

this is for application email merged with Flask Web

A Social Blogging Application

User Authentication

password hashing in the User model

app/models.py:

```
from werkzeug.security import generate_password_hash, check_password_hash
class User(db.Model):
# ...
password_hash = db.Column(db.String(128))
@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)
```

The password hashing function is implemented through a write-only property called password .When this property is set, the setter method will call Werkzeug’s generate_password_hash() function and write the result to the password_hash field.Attempting to read the password property will return an error, as clearly the original Attempting to read the password property will return an error, as clearly the original.

Creating an Authentication Blueprint

Blueprint is a way to define routes in the global scope
after the creation of the application was moved into a factory function.

Example app/auth/__init__.py

```
rom flask import Blueprint
auth = Blueprint('auth', __name__)
from . import views
```

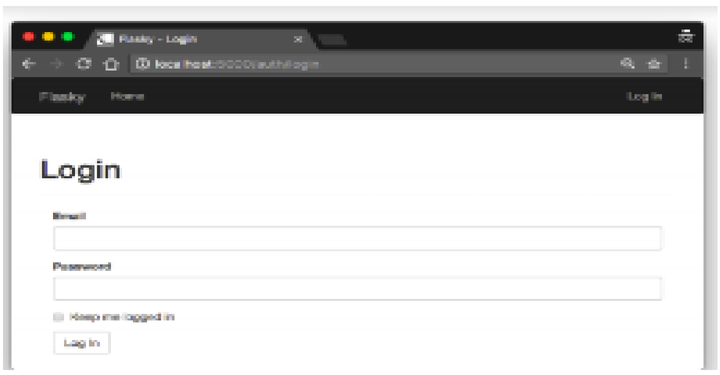
Adding a Login Form

The login form that will be presented to users has a text field for the email address, a password field, a “remember me” checkbox, and a submit button.

Example

app/auth/forms.py

```
from flask_wtf import FlaskForm
from wtforms import StringField, PasswordField, BooleanField, SubmitField
from wtforms.validators import DataRequired, Length, Email
class LoginForm(FlaskForm):
email = StringField('Email', validators=[DataRequired(), Length(1, 64),
Email()])
password = PasswordField('Password', validators=[DataRequired()])
remember_me = BooleanField('Keep me logged in')
submit = SubmitField('Log In')
```

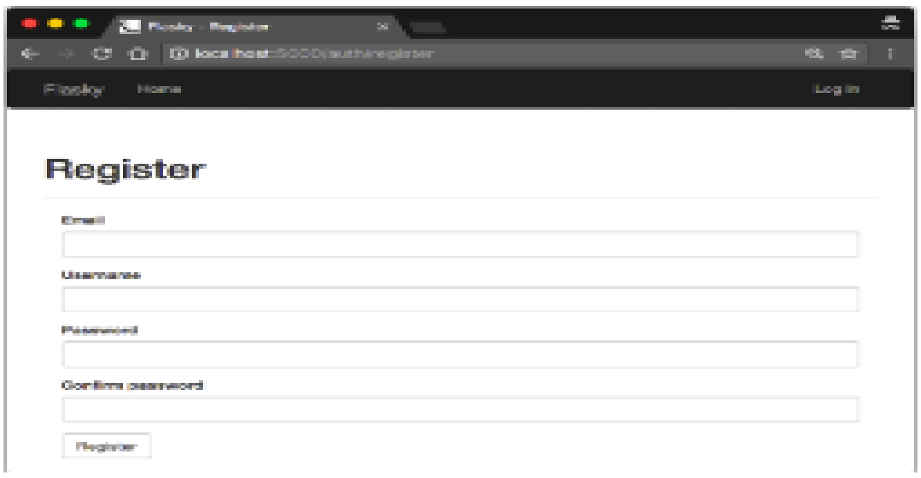


Adding a User Registration Form

The form that will be used in the registration page asks the user to enter an email address, username, and password.

Example app/auth/forms.py

```
from flask_wtf import FlaskForm
from wtforms import StringField, PasswordField, BooleanField, SubmitField
from wtforms.validators import DataRequired, Length, Email, Regexp, EqualTo
from wtforms import ValidationError
from ..models import User
class RegistrationForm(FlaskForm):
email = StringField('Email', validators=[DataRequired(), Length(1, 64),
Email()])
```



User Roles

The user role implementation presented in this chapter is a hybrid between discrete roles and permissions. Users are assigned a discrete role, but each role defines what actions it allows its users to perform through a list of permissions.

Application Programming Interfaces

Function is to provide the client application with data retrieval and storage services. In this model, the server becomes a web service or application programming interface (API).