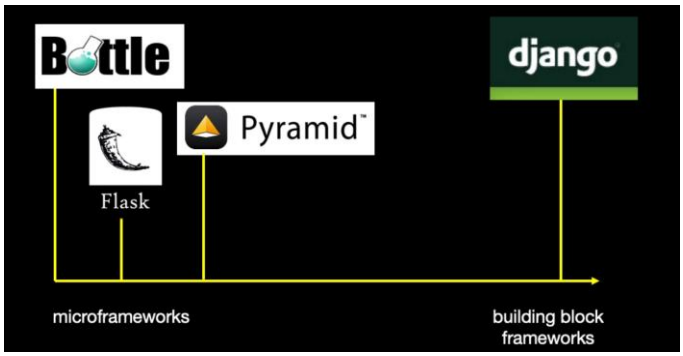
Building Data-driven web apps with Flask and SQLAlchemy

Basics:

Flask is a form of microframework, meaning that any libraries or packages can be used without the framework forcing a certain standard upon the programmer

On the other end, other frameworks come with packages all installed and they are easier to set up, however it restricts customisations



Routes- map URL patterns to views

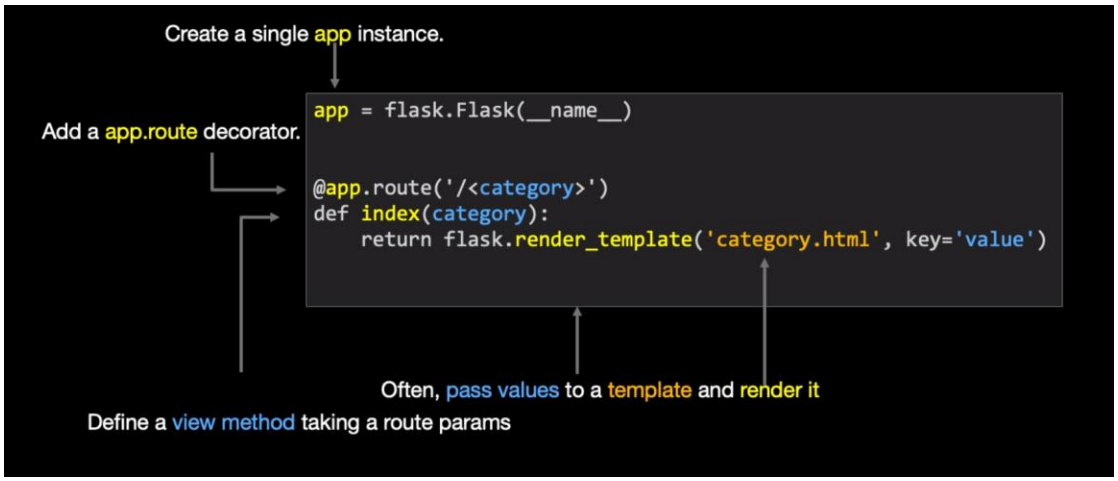
View methods (controllers)- process incoming requests

Templates (aka views)- Dynamic HTML

Models- data and behaviour passed to view

Static content- rich support for cached assests

Basic view template:



More realistic view:

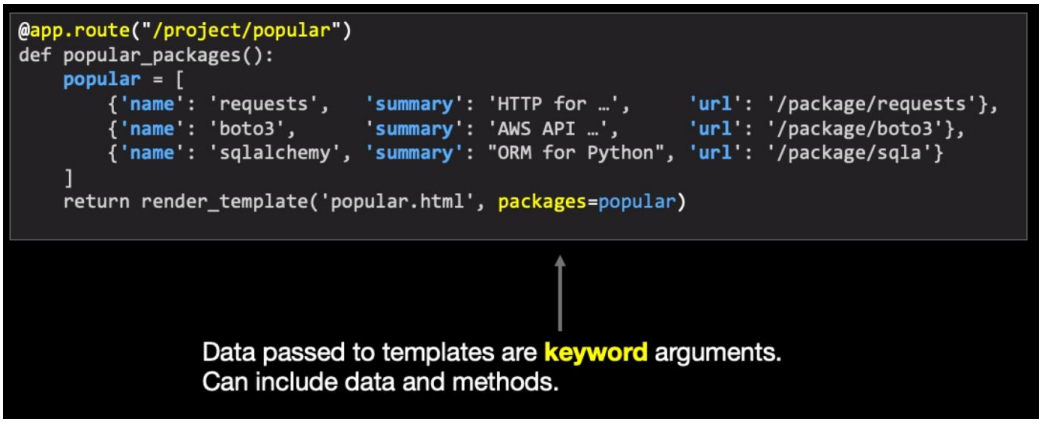
```python
@app.route('/account/register', methods=['POST'])
def register():
    # collect data from params, args, form, headers, etc.
    email = request.form.get('email')
    password = request.form.get('password')

    # process request
    if not repository.create_account(email, password):
        return render_template('register.html', error="Cannot...")

    # return data to template engine
    return render_template('register.html',
                           error=None,
                           msg="Welcome to our community")
```

A view will execute if a route is mapped to the view and is requested by the user, the logic inside of the view will be executed on execution

Methods is used to state that this view will be executed when a post has been submitted

For routes, app.route is used as a decorator that maps URLs to views with a unique url pattern, optional HTTP verb and route data

Data can be passed to a template via keyword arguments- these can also include methods:



Templates can be expressed as “HTML with python where dynamic content is needed”, these have the following format:

```
<!DOCTYPE html>
<html lang="us-en">
<body>
<div>
    {% for p in packages %}
        <div>
            <a href="/project/{{ p.id }}">{{ p.id }}</a>
            {% if p.summary %}
                <span>
                    {{ p.summary }}
                </span>
            {% endif %}
        </div>
    {% endfor %}
</div>
</body>
</html>
```

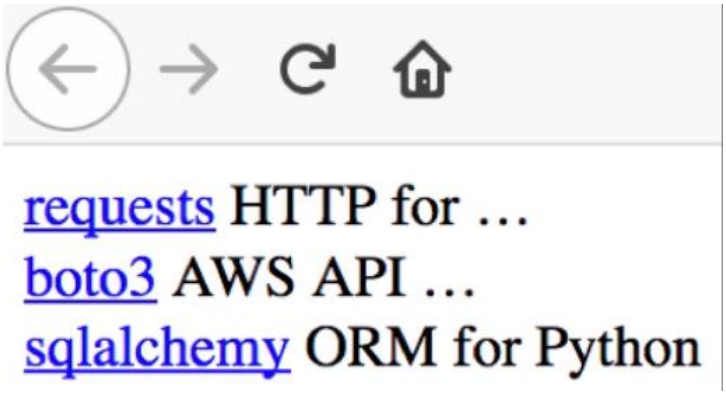
Loops are written via {% for %} blocks on iterables

Conditions are written via {% if %} blocks

Both of these must be contained within an endif and endfor statements

Values are written in double brackets

This produces the following webpage:



Setup:

CLI

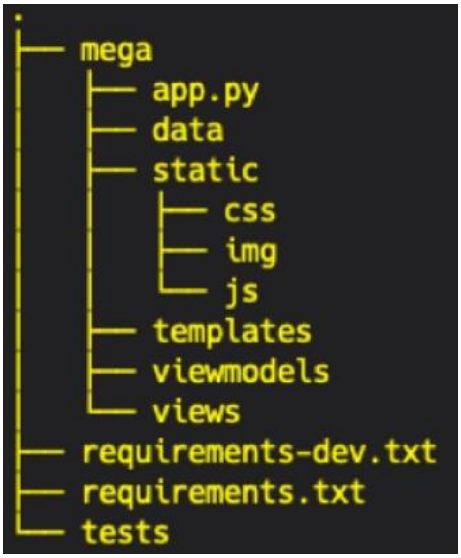
1. Create a virtual environment- this is so that all settings are set local to the project and it is easier managing packages for that specific project

   This is done by first entering the location to start the server and entering:
   - ‘Virtualenv flask’- this creates a virtual machine that will create its own version of python modules such that python will search this file when it is executed when the virtual machine is run

- 'Source flask/bin/activate'- this starts the virtual machine and sets all commands to execute inside of it (the name of the virtual environment is now prepended to the command prompt)
- At this point, all modules inside of the virtual machine can be executed
- 'deactivate'- used to close down the virtual machine
- Install all libraries needed:
    - 'flask/bin/pip install flask'
    - 'flask/bin/pip install flask-login'
    - 'flask/bin/pip install flask-mail'
    - 'flask/bin/pip install flask-sqlalchemy'
    - 'flask/bin/pip install migrate'
    - 'flask/bin/pip install whooshalchemy'
    - 'flask/bin/pip install flask-wtf'
    - 'flask/bin/pip install flask-babel'
    - 'flask/bin/pip install coverage'

  These can be used when the virtual environment is reactivated

2. Create the folder structure:



The requirements text file is used to set the exact requirements so that the server can be recreated if needed. EG:

```
# requirements.txt
flask
sqlalchemy
passlib
```

```
# requirements-dev.txt
-r requirements.txt

pytest
pytest-cov
webtest
```

!The names of the folders must not be capitalised or Flask will not be able to detect the correct files!

3. Create a files:

   Define an app.py, used to start and run the server and a requirements.txt

   App.py should be set up to output some sort of website

4. Start the server- run app.py in the terminal by entering 'python app.py' (make sure VM is activated)

Jinja Templates:

Jinja templates are essentially HTML files with supporting Python

Templates should share the same name as their view methods

To connect views with their templates, the method: flask.render_template('template path')

Data can be passed into the template via keyword arguments, these are then referenced in the templates. These are referenced via double curly brackets

Comparisons and iterations are formatted between curly brackets with a percent sign at either side

Example:

App.py:

```
app = flask.Flask(__name__)


def get_latest_packages():
    return [
        {'name': 'flask', 'version': '1.2.3'},
        {'name': 'sqlalchemy', 'version': '2.2.0'},
        {'name': 'passlib', 'version': '3.0.0'},
    ]


@app.route('/')
def index():
    test_packages = get_latest_packages()
    return flask.render_template('index.html', packages=test_packages)


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

Jinja-Template:

```
</style>

<h1>Python Package Index</h1>

<h2>Packages</h2>

{% for p in packages %}
<div>
    <span class="title">
        {{ p.name.upper() }}
    </span>
    <span class="version">
        {{ p.version }}
    </span>
</div>


{% endfor %}

</body>
</html>
```

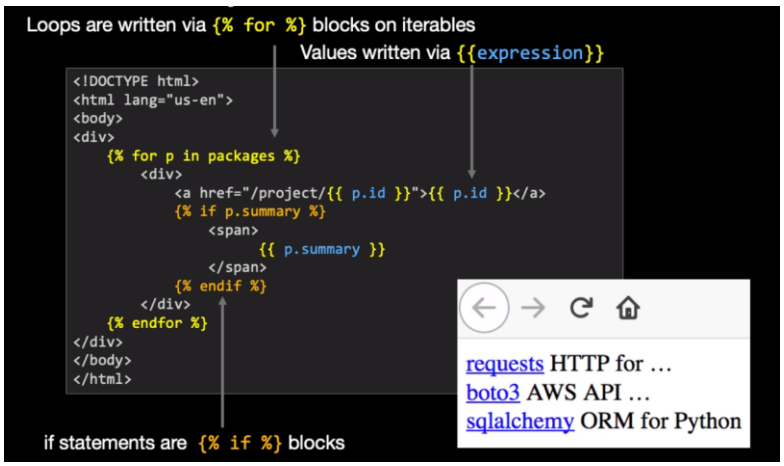
Website:

Python Package Index

Packages

FLASK 1.2.3
SQLALCHEMY 2.2.0
PASSLIB 3.0.0

Basic template:



Having a consistent layouts state the following:

- Consistent meta info
- Consistent CSS and JavaScript files
- Consistent analytics
- Controlled Add-on CSS and JavaScript

To include a global layout page in many other pages, the following syntax is used:

Layout page:

```
<div class="main_content">
    {% block main_content %}{% endblock %}
</div>
```

Normal page:

```
{% extends "_layout.html" %}
{% block main_content %}
    <h1>Python Package Index</h1>

    <h2>About</h2>

    <div>
        This is our demo app for our Flask course.
    </div>
{% endblock %}
```

There can be multiple blocks that can be added into the layout page- eg: {% block Additional_CSS %}{% endblock %}, this allows the user to insert extra CSS if needed; it can be left blank

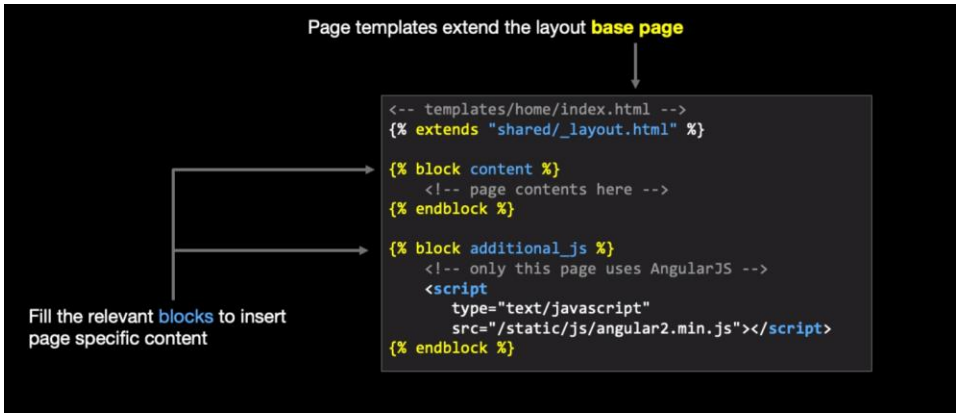
Now that blocks have been introduced, the content should be split into folders which properly represent their correct purpose- inside the template folder the following folders should be created:

- Home- normal home page webpages
- Shared- a folder which contains content which is shared between many different webpages

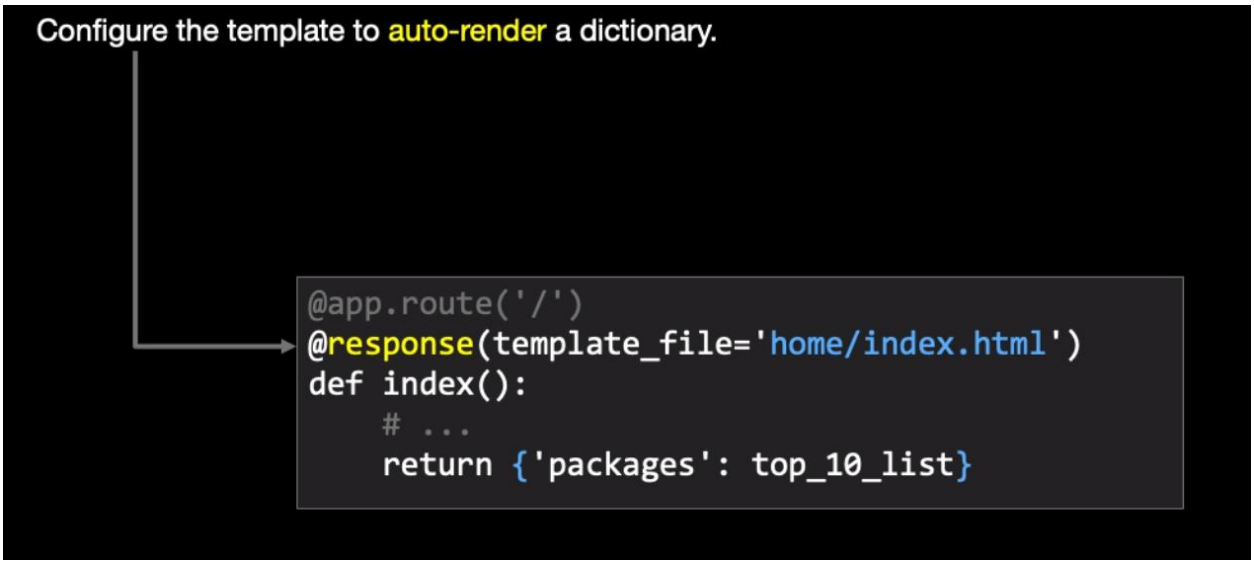
Standard layout html:

More blocks is better as it means more content can be added

This can be used as follows:



@response(template_file=”[directory-path]”)- this is another way of returning a response to a user’s request.



A response decorator needs to be defined under the following code:

```python
# pypi_org/infrastructure/view_modifiers.py
def response(*, mimetype: str = None, template_file: str = None):
    def response_inner(f):
        # print("Wrapping in response {}".format(f.__name__), flush=True)

        @wraps(f)
        def view_method(*args, **kwargs):
            response_val = f(*args, **kwargs)
            if isinstance(response_val, flask.Response):
                return response_val

            if template_file and not isinstance(response_val, dict):
                raise Exception(
                    "Invalid return type {}, we expected a dict as the return value.".format(type(response_val)))

            if template_file:
                response_val = flask.render_template(template_file, **response_val)

            resp = flask.make_response(response_val)
            if mimetype:
                resp.mimetype = mimetype

            return resp

        return view_method

    return response_inner
```

Routing and URLs:

Having a professional URL hierarchy is beneficial for the following reasons:

- Users can more easily navigate your website
- Search engine optimisation

Routing in Flask has the following structure:

Flask will pass the URL into the view method

The arguments of the URL are defined as request.args, which is a global dictionary

All the viewmethods should not be stored in one Python file, hence these are normally split into many different Python scripts

A new folder should be created that is used for providing useful functions for the webserver, this is called ‘services’ and is stored as one of the top directories (next to views, templates, ext…)

Flask blueprints allows for apps to be declared and not defined in multiple python files so that it can be referenced before the server is started and the app is created, this can be done using the Blueprint method, which requires three things:

- A name which represents the view (‘EG. Home’)
- The name of the file: __name__ (this is for when the blueprint is imported)
- Template folder (EG. templates)

Next all the app.route() snippets should be replaced with the name of the blueprint object:

Blueprint = flask.Blueprint(‘home’,__name__,’templates’)

Blueprint.route(‘/’)

Example:

```
import flask

from pypi_org.infrastructure.view_modifiers import response
import pypi_org.services.package_service as package_service

blueprint = flask.Blueprint('home', __name__, template_folder='templates')


@blueprint.route('/')
@response(template_file='home/index.html')
def index():
    test_packages = package_service.get_latest_packages()
    return {'packages': test_packages}
    # return flask.render_template('home/index.html', packages=test_packages)


@blueprint.route('/about')
@response(template_file='home/about.html')
def about():
    return {}
```

Next the app has to be aware of each of the blueprint views that have been created, hence the app.py should be rearranged to form the following format:

```python
import flask

app = flask.Flask(__name__)


def main():
    register_blueprints()
    app.run(debug=True)


def register_blueprints():
    from pypi_org.views import home_views
    app.register_blueprint(home_views.blueprint)


if __name__ == '__main__':
    main()
```

For URLs where one base URL can have multiple descendants, hence a template should be created that allows for multiple of these pages to be referenced from a parent page, this has the following syntax:

```python
@blueprint.route('/project/<package_name>')
#@response(template_file='packages/details.html')
def package_details(package_name: str):
    return "Package details for {}".format(package_name)
```

Views and templates should be organised in directories

Basic format:

```python
# views/packages.py
import flask

blueprint = flask.Blueprint('home', __name__, template_folder='templates')


@blueprint.route('/')
def index():
    # ...
```

```python
# app.py
import pypi_org.views.packages

app = flask.Flask(__name__)

app.register_blueprint(pypi_org.views.packages.blueprint)

app.run()
```

Flask can take in many integer URLs and produce pages that are related to that input, EG. returning ranks for webpages on the website. The code is as follows:

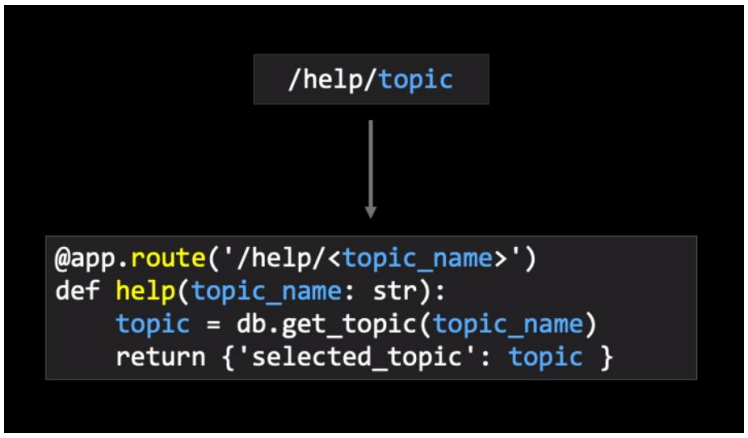
```
@blueprint.route('/<int:rank>')
def popular(rank: int):
    return "The details for the {}th most popular package".format(rank)
```

Setting the route as int:, results in only integers being captured and returned as that result. This function automatically returns the URL, as an integer called rank. This will output on a webpage as:

Different HTML requests should be handled in different functions in the views:

```
@blueprint.route('/account/register', methods=['GET'])
@response(template_file='account/register.html')
def register_get():
    return {}


@blueprint.route('/account/register', methods=['POST'])
@response(template_file='account/register.html')
def register_post():
    return {}
```

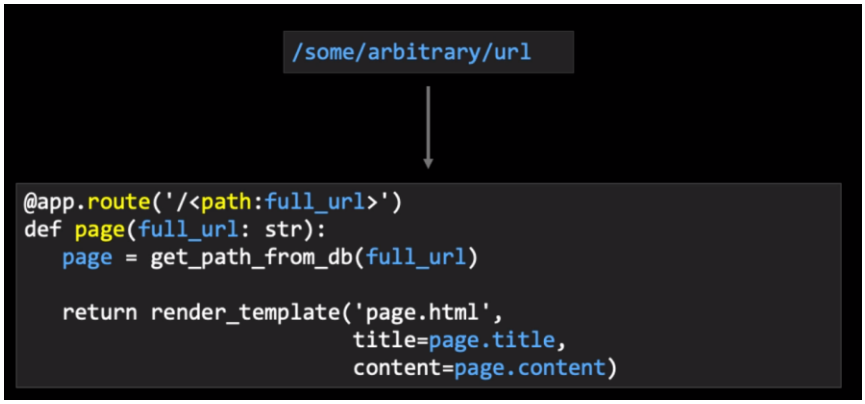
Routes can be used for returning many different webpages for the same parent URL:



Flask.abort(404)- used for returning that a page was not found

To route the whole path in flask and pass it to a view, the '<path:full_url>' syntax is used, this will return the entire URL (eg. a/b/c/d)

Routing can be used to create a custom CMS, here is a snippet of how this can be implemented:



CSS front-end framework:

Bootstrap employs reset-CSS, this means that every property is automatically set to a default value. This is beneficial as if the user was to view the website from a different browser, the webpage will look consistent across all browsers

Referencing the bootstrap CDN will allow bootstrap to be used

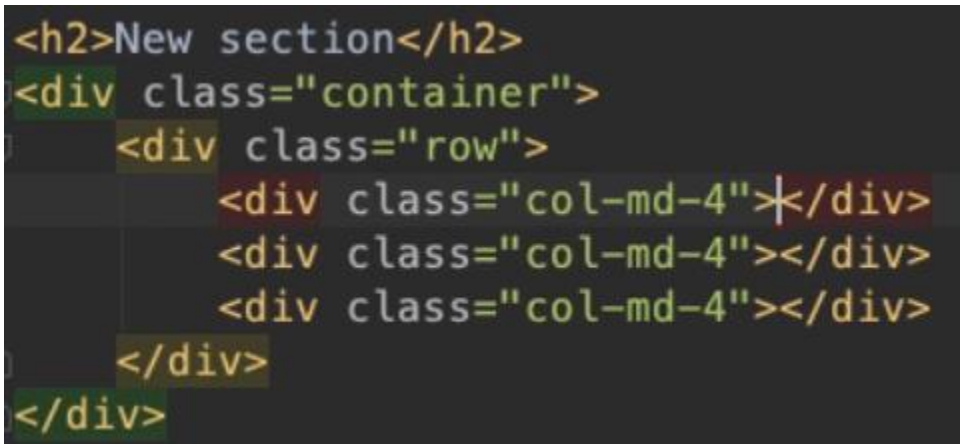
Pycharm allows HTML to be run which results in the file being run on a server

Bootstrap data will stay in column form until the browser size is smaller than specified and it will then be represented in tabular form

Bootstrap data must be inside of a container, which has a series of rows, which then have a series of responsive columns

Pycharm has shorthand for creating bootstrap elements:

div.container>div.row>div.col-md-4*3



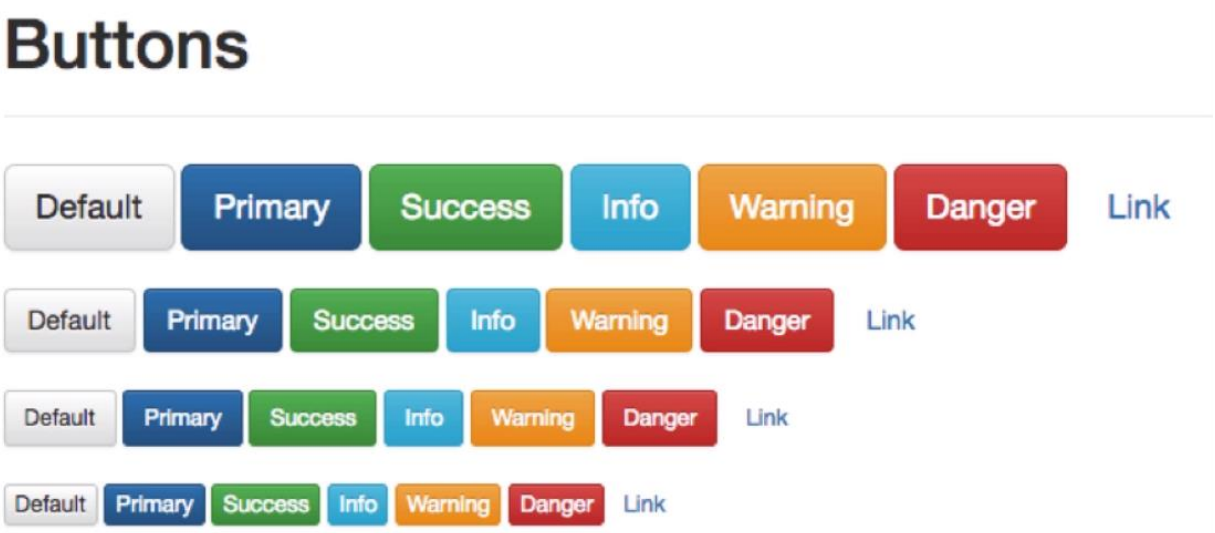
Bootstrap columns have the following sizes:

Size	Width
Extra small	<576px
Small	≥576px
Medium	≥768px
Large	≥992px
Extra large	≥1200px

Links and buttons can be set under the class button to be represented as a button

"form-control"- bootstrap class for form elements which makes them more recognisable and professional

Button types:



Bootstrap themes can be used to increase the professionalism of a website by giving it a set theme- two websites for this are "startbootstrap.com" and "wrapbootstrap"s

Unsplash- royalty free image websites

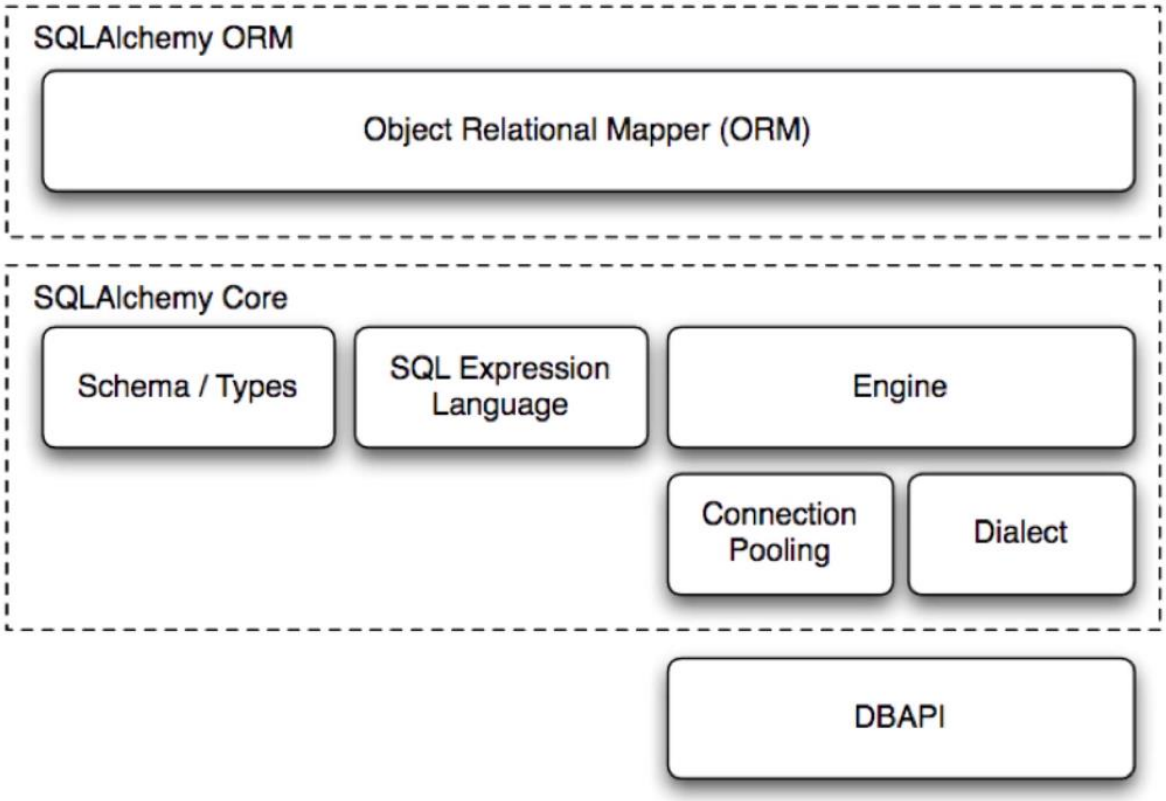
Display is used to set how the data inside of a div is represented. Inline-block is used to set the elements to be displayed horizontally

SQLAlchemy:

Is a data access layer, as it allows you to communicate with many relational database management software- benefits:

- No ORM required
- Mature, high performing
- DBA approved- ability to swap out generated SQL with hand-optimising statements
- Unit of work- organises pending insert/update/delete operations into queues and flushes them in one batch
- Many database supports
- Eager-loading- allows entire graphs of objects linked by collections and references to be loaded with few or just one query

Architecture:



SQL-alchemy files should be stored in their own directory, which is normally named 'data'

Each SQL-alchemy class should be stored in its own class file

SQL-alchemy is normally imported as 'import sqlalchemy as sa'

All SQLAlchemy classes must be derived from a base class, which is defined as:

All classes on this database must derive from this class

The class properties of the class should be set to the columns of the table, these are defined as:

Variable = sa.column(sa.datetype)

The primary key can be set by passing the keyword argument as: primary_key=True, and autoincrement can be set the same way

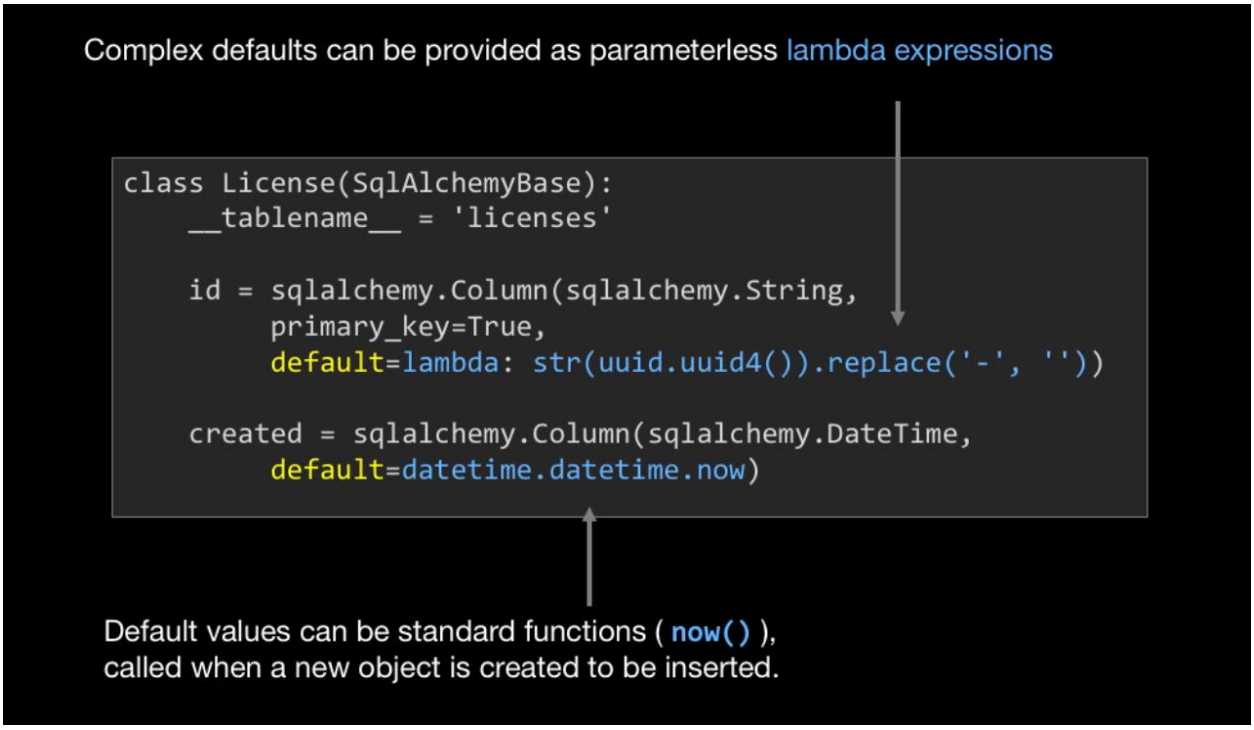
To change the table name, the variable '__tablename__' is referenced

For debugging, __repr__(self) should return the key column of the database

Here is the format for creating tables:

```
class Package(SqlAlchemyBase):
    __tablename__ = 'packages'

    id = sqlalchemy.Column(sqlalchemy.Integer,
                primary_key=True, autoincrement=True)

    summary = sqlalchemy.Column(sqlalchemy.String)
    size = sqlalchemy.Column(sqlalchemy.Integer)
    home_page = sqlalchemy.Column(sqlalchemy.String)

    releases = sqlalchemy.orm.relationship("Release", ...)
```

Default values can be set as the following:



Default values can be created using any sort of function

Indexes help speed up search results on databases for more important fields, however they do increase write time

Keys and indexes are defined as following:

Relationships can be defined using the following format- in the primary table a relationship is created as such:

```
# releases relationship
releases = orm.relation("Release", order_by=[
    Release.major_ver.desc(),
    Release.minor_ver.desc(),
    Release.build_ver.desc(),
], back_populates='package')
```

Meanwhile in the foreign table a relationship is created as:

```
# Package relationship
package_id = sqlalchemy.Column(sqlalchemy.String, sqlalchemy.ForeignKey("packages.id"))
package = orm.relation('Package')
```

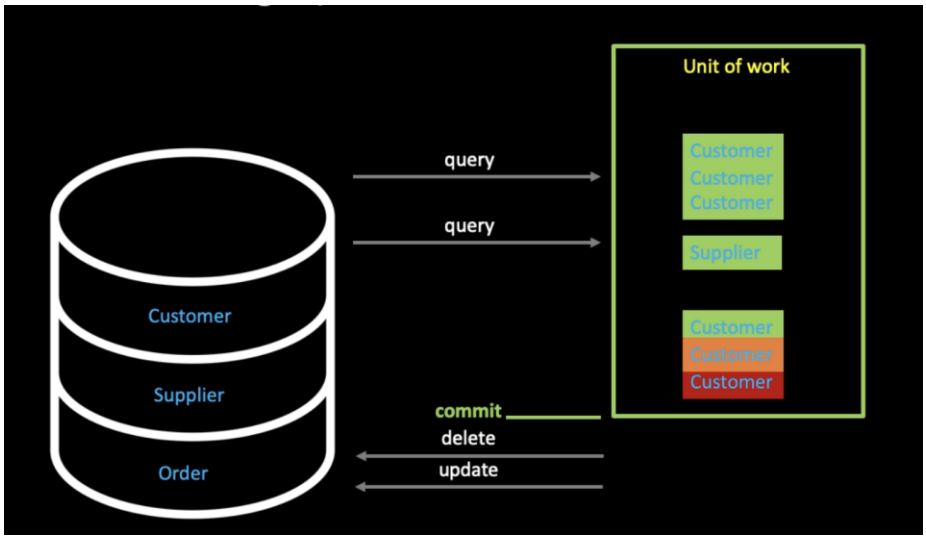
Note that the table name is referenced as the foreign key, rather than the class name

Finally, the database can be created using the following code:

Traverses all derived classes, creates a table for each, won't update existing.

```
# import all SqlAlchemyBase derived classes
import packages
import releases ...

# run this code only once per process assuming 1 database
conn_str = 'sqlite:///' + db_file

engine = sqlalchemy.create_engine(conn_str, echo=False)
SqlAlchemyBase.metadata.create_all(engine)

session_factory = sqlalchemy.orm.sessionmaker(bind=engine)
```

Unit of work design, this is a SQL-Alchemy Session:

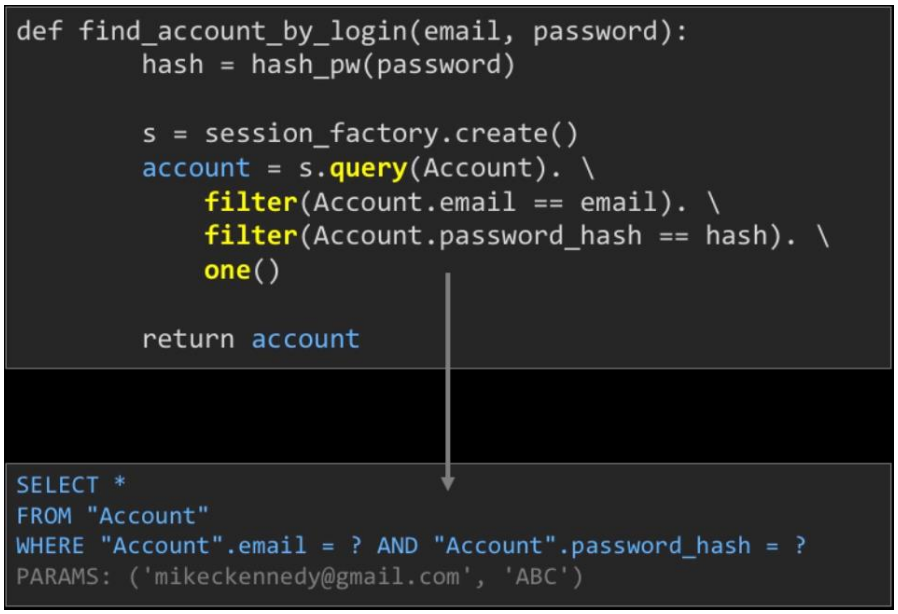


This is represented by the following code:

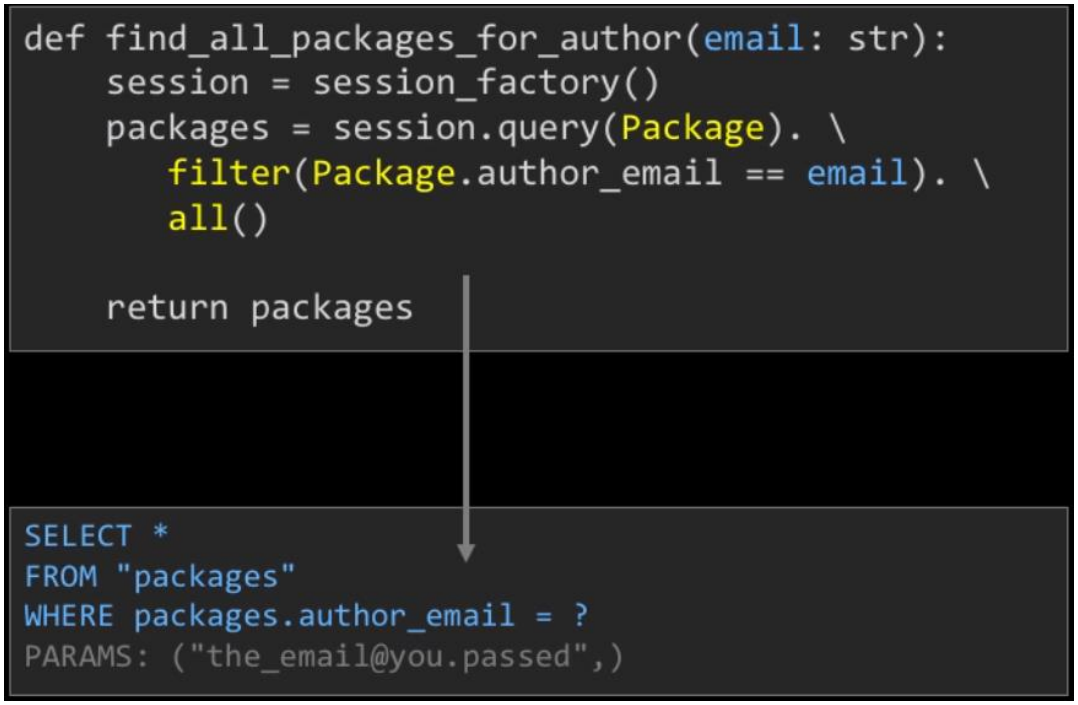
```
# run this code only once per process assuming 1 database
conn_str = 'sqlite:///' + db_file
engine = sqlalchemy.create_engine(conn_str, echo=False)
session_factory = sqlalchemy.orm.sessionmaker(bind=engine)

# run this code once per unit of work (think transaction)
# session is the Unit of Work in SQLAlchemy
session = session_factory()

session.add(...)
session.query(...)
session.add(...)

session.commit()
```

Querying a single record:

This searches an account where the email and password match the function arguments, and it only returns one object as you cannot have more than one account:



Querying multiple records:



SQL equivalents:

Operation	ORM Syntax
EQUALS	`query.filter(User.name == 'ed')`
NOT EQUAL	`query.filter(User.name != 'ed')`
LIKE	`query.filter(User.name.like('%ed%'))`
IN	`query.filter(User.name.in_(['ed', 'wendy', 'jack']))`
NOT IN	`query.filter(~User.name.in_(['ed', 'wendy', 'jack']))`
NULL	`query.filter(User.name == None)`
AND	`query.filter(User.name == 'ed').filter(User.age == 27)`
OR	`query.filter(or_(User.name == 'ed', User.name == 'wendy'))`

Ordering:

```
def find_all_packages():
    session = session_factory()
    packages = session.query(Package). \
       order_by(Package.created.desc()). \
       all()

    return packages
```

Update:

```
def set_package_author(new_email, new_name, package_id):
    session = session_factory()
    package = session.query(Package). \
       filter(Package.id == package_id). \
       one()

    package.author = new_name
    package.author_email = new_email

    session.commit()
```

Retrieve one or more objects, update, call commit().

SQL-alchemy will track the package that has been changed, so when it is committed it can easily be updated in the database

Relationships:

Back_populates means that the relationship works both ways

```
class Package(SqlAlchemyBase):
    id = sqlalchemy.Column(sqlalchemy.String, ...)

    releases = orm.relationship("Release", order_by=[
            Release.major_ver.desc(),
            Release.minor_ver.desc(),
            Release.build_ver.desc()
        ], back_populates="package")
```

```
class Release(SqlAlchemyBase):
    id = ...

    package_id = sqlalchemy.Column(sqlalchemy.String, sqlalchemy.ForeignKey('packages.id'))
    package = orm.relationship("Package", back_populates="releases")
```

Another example is the following code:

```
from flask import Flask
from flask_sqlalchemy import SQLAlchemy

app = Flask(__name__)
app.config['SQLALCHEMY_DATABASE_URI'] = 'sqlite:///relationships.db'
db = SQLAlchemy(app)

class Person(db.Model):
    id = db.Column(db.Integer, primary_key=True)
    name = db.Column(db.String(20))
    pets = db.relationship('Pet', backref='owner')

class Pet(db.Model):
    id = db.Column(db.Integer, primary_key=True)
    name = db.Column(db.String(20))
    owner_id = db.Column(db.Integer, db.ForeignKey('person.id'))
```

This code means that from person the variable ‘pets’ can be referenced, which will return a list of all the pets that share the same owner_id as the person primary key, and referencing owner from pets will return the owner for that pet. This also means that relationships can easily be created when a person/pet is created, this is done by using the relationship as a field:

```
>>> anthony = Person(name='Anthony')
>>> db.session.add(anthony)
>>> db.session.commit()
>>> michelle = Person(name='Michelle')
>>> db.session.add(michelle)
>>> db.session.commit()
>>> spot = Pet(name='Spot', owner=anthony)
>>> db.session.add(spot)
>>> db.session.commit()
```

Inserting data:

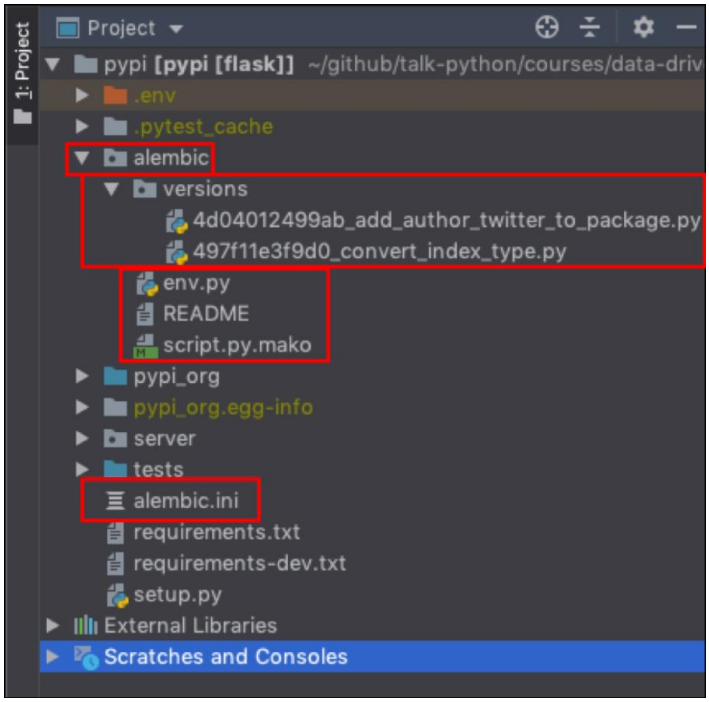
```
session = session_factory()

package = Package()
package.id = 'sqlalchemy'
package.author = 'Mike Bayer'

release1 = Release()
release1.package = package
release1...

release2 = Release()
release2.package = package
release2...

session.add(package)

session.commit()
```

Alembic- database migrations:

Database migration tool- this is used for updating database via scripts rather than editing the tables manually, this also means that the table can be downgraded if need be. Alembic is created as follows:

```
(env) $ pip install alembic
... installed alembic 1.11.0, ...

(env) $ alembic init alembic
Creating directory ./alembic ... done
  Creating directory . /alembic/versions ... done
  Generating ./alembic/script.py.mako ... done
  Generating ./alembic/env.py ... done
  Generating ./alembic/README ... done
  Generating ./alembic.ini ... done
  Edit configuration/connection/logging settings in './alembic.ini' before proceeding.
```

New structure:



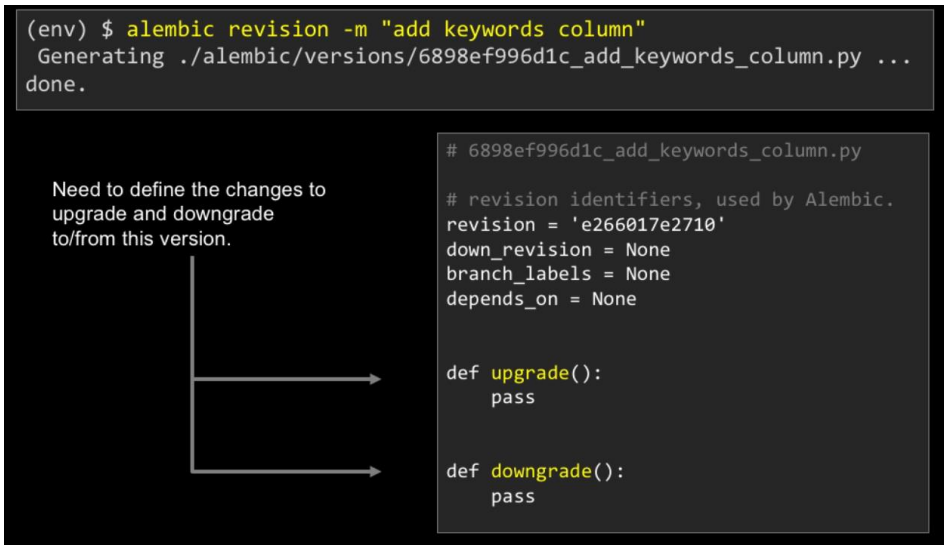
As the table is changed, more version scripts are added

Finally, the main change is to set the path to the database in alembic.ini:

```
# alembic.ini

# lots of optional settings ...

sqlalchemy.url = sqlite:///./pypi_web/db/pypi_dev.sqlite
```

There are two ways to deal with database migrations with alembic, manually creating the scripts and using SQLAlchemy changes to generate scripts

Manual creation can be done as follows:



The change is then defined:

```
# 6898ef996d1c_add_keywords_column.py

# revision identifiers, used by Alembic.
revision = 'e266017e2710'
down_revision = None
branch_labels = None
depends_on = None


def upgrade():
    op.add_column('packages', sa.Column('keywords', sa.String, nullable=True))


def downgrade():
    op.drop_column('packages', sa.Column('keywords'))
```

A safe way of checking that a column doesn't already exist is by using this function:

```
# IMPORTANT: Name: alembic_helpers.py, place in alembic folder.

from alembic import op
from sqlalchemy import engine_from_config
from sqlalchemy.engine import reflection


def table_has_column(table, column):
    config = op.get_context().config
    engine = engine_from_config(
        config.get_section(config.config_ini_section), prefix='sqlalchemy.')
    insp = reflection.Inspector.from_engine(engine)
    has_column = False
    for col in insp.get_columns(table):
        if column not in col['name']:
            continue
        has_column = True
    return has_column
```

This is implemented as:

```
# 6898ef996d1c_add_keywords_column.py

# revision identifiers, used by Alembic.
…

import imp
import os
alembic_helpers = imp.load_package('alembic_helpers',
   os.path.abspath(os.path.join(os.path.dirname(__file__), '..', 'alembic_helpers.py')))


def upgrade():
    if not alembic_helpers.table_has_column('packages', 'keywords'):
        op.add_column('packages', sa.Column('keywords', sa.String, nullable=True))

...
```

Finally the database is upgraded:

```
(env) $ alembic upgrade head
INFO  [alembic.runtime.migration] Context impl SQLiteImpl.
INFO  [alembic.runtime.migration] Will assume non-transactional DDL.
INFO  [alembic.runtime.migration] Running upgrade 078e6813b9c1 -> 6898ef996d1c, add keywords column
```

The other way is by using auto-generated changes. To do this, the target_metadata must be set to the base class:

```
# env.py

import pypi_web
from pypi_web.data import *


target_metadata = pypi_web.data.modelbase.SqlAlchemyBase.metadata
```

Next a change in the ORM models are made and a upgrade script is created:

```
(env) $ alembic revision --autogenerate -m "add last login column"
 Detected added column 'users.last_login'
 Generating ./alembic/versions/e266017e2710_add_last_login_column.py ...
done.
```

```
# e266017e2710_addlast_login_column.py

# revision identifiers, used by Alembic.
revision = 'e266017e2710'
down_revision = '6898ef996d1c'
branch_labels = None
depends_on = None


def upgrade():

    # ### commands auto generated by Alembic - please adjust! ###
    op.add_column('users', sa.Column('last_login', sa.String, nullable=True))
    # ### end Alembic commands ###
```

Finally the database can be updated

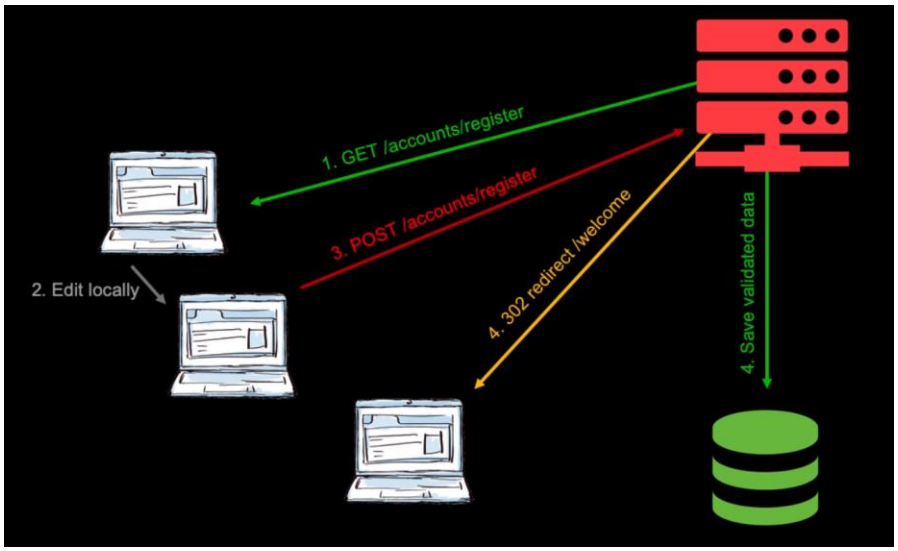
```
(env) $ alembic upgrade head
INFO  [alembic.runtime.migration] Context impl SQLiteImpl.
INFO  [alembic.runtime.migration] Will assume non-transactional DDL.
INFO  [alembic.runtime.migration] Running upgrade 6898ef996d1c -> e266017e2710, add last login
```

User input with HTML forms:

Having an empty action results in the page refresh

POST cannot be cached, meaning it is more secure

All links and normal pages are retrieved using GET, while sensitive data is sent via POST:



Form data can be captured using the following method:

```python
@blueprint.route('/account/register', methods=['POST'])
@response(template_file='account/register.html')
def register_post():
    r = flask.request

    name = r.form.get('name')
    email = r.form.get('email', '').lower().strip()
    password = r.form.get('password', '').strip()

    if not name or not email or not password:
        return {
            'error': "Some required fields are missing."
        }

    return {}
```

To keep values inside of some input fields when the page is refreshed, set the value of the fields to be the previous submitted values that were sent:

```
Your name" class="form-control" value="{{ name }}">
' Your email address" class="form-control" value="{{ email }}
nolder=" Password" class="form-control" value="{{ password }}
>Register</button>
```

To redirect users, the return value of a view is set to flask.redirect('URL')

To create a user on a database, take the user inputs from a form and use them to insert a new record on the database

Passlib can be used for hashing passwords, as well as verifying the passwords. It also salts the password (adds random words in between the password) as well as folding the password many times (rehashing the password many times):

```python
def create_user(name: str, email: str, password: str) -> Optional[User]:
    if find_user_by_email(email):
        return None

    user = User()
    user.email = email
    user.name = name
    user.hashed_password = hash_text(password)

    session = db_session.create_session()
    session.add(user)
    session.commit()

    return user


def hash_text(text: str) -> str:
    hashed_text = crypto.encrypt(text, rounds=171204)
    return hashed_text


def verify_hash(hashed_text: str, plain_text: str) -> bool:
    return crypto.verify(plain_text, hashed_text)
```

Once a hash has been computed, it can never be recomputed again as it has been randomly salted:

```
def login_user(email: str, password: str) -> Optional[User]:
    session = db_session.create_session()

    user = session.query(User).filter(User.email == email).first()
    if not user:
        return None

    if not verify_hash(user.hashed_password, password):
        return None

    return user
```

Cookies are used for saving the account details of a user so that they stay logged in when they use the website. However, if the user wants, they could hack the cookie to change the details to appear to be another user, hence the userId should also be hashed so that the user cannot change it to another user's account:

```
auth_cookie_name = 'pypi_demo_user'


def set_auth(response: Response, user_id: int):
    hash_val = __hash_text(str(user_id))
    val = "{}:{}".format(user_id, hash_val)
    response.set_cookie(auth_cookie_name, val)


def __hash_text(text: str) -> str:
    text = 'salty__' + text + '__text'
    return hashlib.sha512(text.encode('utf-8')).hexdigest()
```

Set_auth sets the user ID as well as a hashed value that identifies that the user is logged in correctly

Creating the cookie can be done as follows:

```
def __add_cookie_callback(_, response: Response, name: str, value: str):
    response.set_cookie(name, value, max_age=timedelta(days=30))


def get_user_id_via_auth_cookie(request: Request) -> Optional[int]:
    if auth_cookie_name not in request.cookies:
        return None

    val = request.cookies[auth_cookie_name]
    parts = val.split(':')
    if len(parts) != 2:
        return None

    user_id = parts[0]
    hash_val = parts[1]
    hash_val_check = __hash_text(user_id)
    if hash_val != hash_val_check:
        print("Warning: Hash mismatch, invalid cookie value")
        return None

    return try_int(user_id)
```

Displaying an account page is as follows:

```
@blueprint.route('/account')
@response(template_file='account/index.html')
def index():
    user_id = cookie_auth.get_user_id_via_auth_cookie(flask.request)
    if user_id is None:
        return flask.redirect('/account/login')

    user = user_service.find_user_by_id(user_id)
    if not user:
        return flask.redirect('/account/login')

    return {
        'user': user
    }
```

```
def find_user_by_id(user_id: int) -> Optional[User]:
    session = db_session.create_session()
    user = session.query(User).filter(User.id == user_id).first()
```

A trick for returning request data from all sources of data input can be done as follows:

```
class RequestDictionary(dict):
    def __getattr__(self, key):
        return self.get(key)


def create(**route_args) -> RequestDictionary:
    request = flask.request
    data = {
        **request.args,  # The key/value pairs in the URL query string
        **request.headers,  # Header values
        **request.form,  # The key/value pairs in the body, from a HTML post form
        **route_args  # And additional arguments the method access, if they want them merge
    }

    return RequestDictionary(data)
```

This eliminates the need to worry about whether data was retrieved via URL, or header, or post ext…

The order of the input data matters as form will override args (as it is more secure/important)

This means that the login page can be altered to show the following:

```
@blueprint.route('/account/login', methods=['POST'])
@response(template_file='account/login.html')
def login_post():
    data = request_dict.create()

    email = data.email.lower().strip()
    password = data.password.strip()
```

Client and server-side validation:

View models can be used for simplifying views by creating a class that is dedicated with dealing with data being passed to and from that view and the template. A base view can be specified and contains attributes that will be passed to the template for all pages, for example the user ID which is saved to the cookie, or an error attribute which returns if there is an error in a field ext… This has the following format:

```
class ViewModelBase:
    def __init__(self):
        self.request: Request = flask.request
        self.request_dict = request_dict.create('')

        self.error: Optional[str] = None
        self.user_id: Optional[int] = cookie_auth.get_user_id_via_auth_cookie(self.request)

    def to_dict(self):
        return self.__dict__
```

To make this useful for each of the view methods, a subclass of the view model base should be made that includes attributes that are passed to the view template. This is what a view model will look like for an index page:

```
class IndexViewModel(ViewModelBase):
    def __init__(self):
        super().__init__()
        self.user = user_service.find_user_by_id(self.user_id)
```

This is the updated view for the index page:

```
def index():
    vm = IndexViewModel()
    if not vm.user:
        return flask.redirect('/account/login')

    return vm.to_dict()
```

In SQLAlchemy, the session is always closed whenever there is a commit, this is to avoid errors with the database. This can create the following error: ‘sqlalchemy.orm.exc.DetachedInstanceError’. This can be fixed by editing the factory code to as follows:

```
def create_session() -> Session:
    global __factory

    session: Session = __factory()

    session.expire_on_commit = False

    return session
```

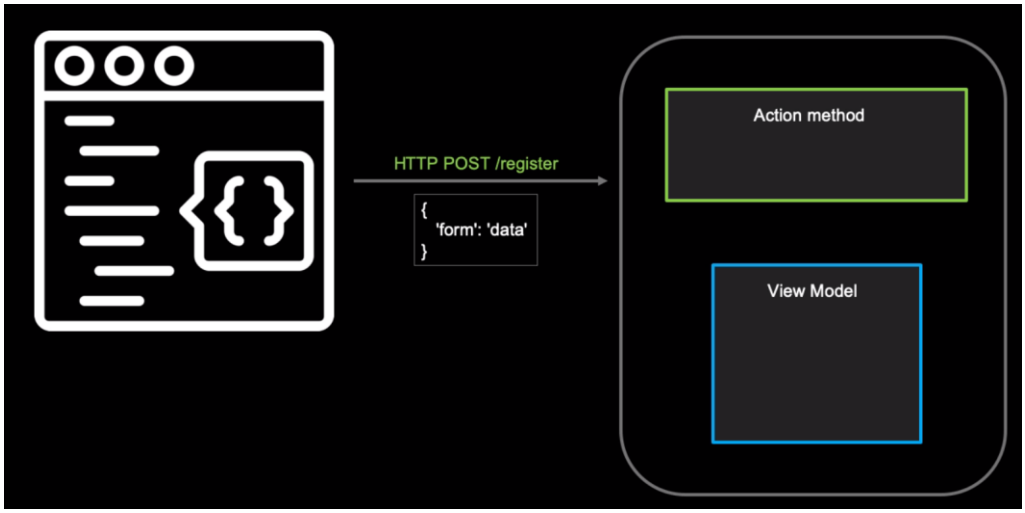
The register view model can be implemented as:

```
class RegisterViewModel(ViewModelBase):
    def __init__(self):
        super().__init__()
        self.name = self.request_dict.name
        self.email = self.request_dict.email.lower().strip()
        self.password = self.request_dict.password.strip()

    def validate(self):
        if not self.name or not self.name.strip():
            self.error = 'You must specify a name.'
        elif not self.email or not self.email.strip():
            self.error = 'You must specify a email.'
        elif not self.password:
            self.error = 'You must specify a password.'
        elif len(self.password.strip()) < 5:
            self.error = 'The password must be at least 5 characters.'
        elif user_service.find_user_by_email(self.email):
            self.error = 'A user with that email address already exists.'
```

The register view will be refactored as:

```
@blueprint.route('/account/register', methods=['POST'])
@response(template_file='account/register.html')
def register_post():
    vm = RegisterViewModel()
    vm.validate()

    if vm.error:
        return vm.to_dict()

    user = user_service.create_user(vm.name, vm.email, vm.password)
    if not user:
        vm.error = 'The account could not be created'
        return vm.to_dict()

    resp = flask.redirect('/account')
    cookie_auth.set_auth(resp, user.id)

    return resp
```

View models produce the following pattern:

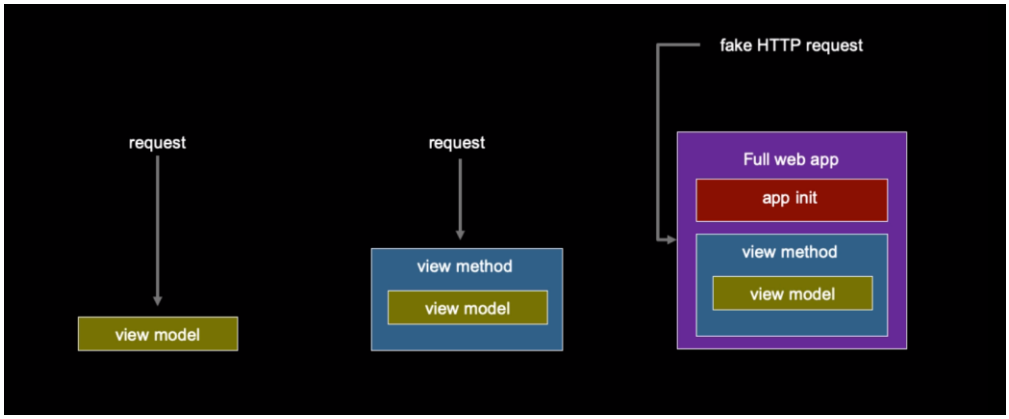


Testing:

Testing can be incredibly difficult as most of the all of the parts of the website rely and interact with other parts of the website

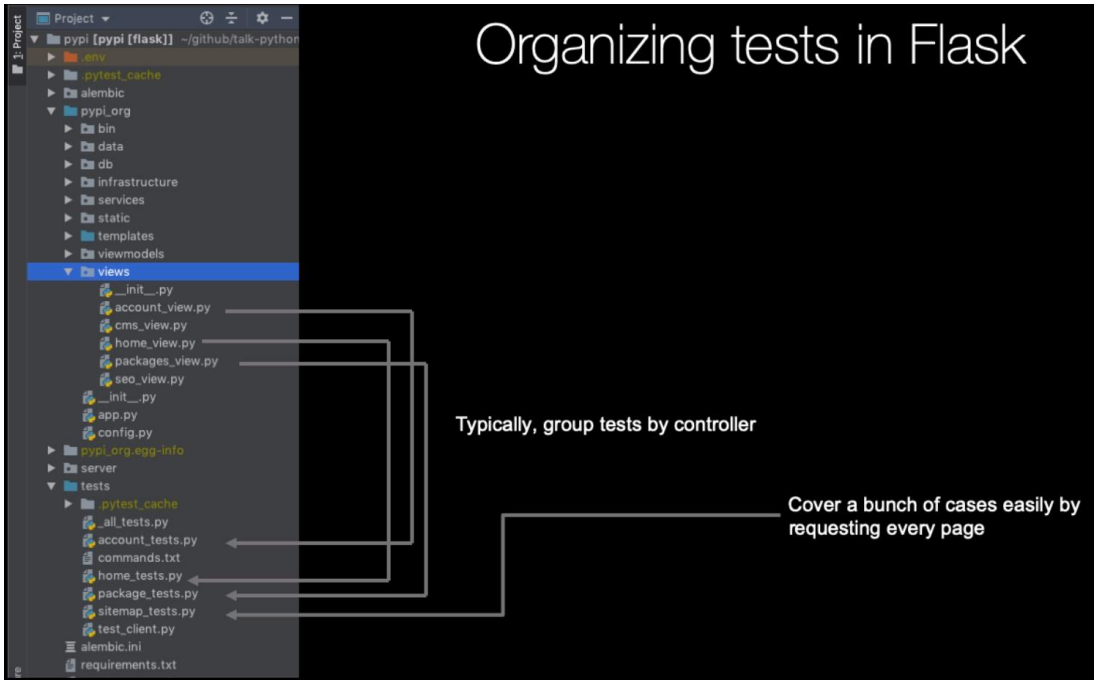
Three types of tests:

- View models
- Views
- Integration



Sitemaps can be used to test every single pages on the website

Tests are organised as such:



Tests are done using pytests- all functions done using pytest must start with test_[function_name]

Tests should follows the three A's of testing:

- Arrange- set the values that need to be tested
- Act- pass the values into the function to be tested
- Assert- compare returned values with the result

Fake data can be presented as a flask request using the following code:

```
from pypi_org.app import app as flask_app


@pytest.fixture
def client():
    flask_app.app.config['TESTING'] = True
    client = flask_app.app.test_client()

    try:
        flask_app.register_blueprints()
    except:
        pass

    flask_app.init_db()
    # client.post()

    yield client
```

This sets up a way of setting the fake data. Until the integration tests, all that is needed is the flask_app to be referenced which is the actual app script that runs flask, however in the integration tests the client will be used for rendering the website without running flask

```
def test_register_validation_when_valid():
    # 3 A's of test: Arrange, Act, then Assert

    # Arrange
    form_data = {
        'name': 'Michael',
        'email': 'michael@talkpython.fm',
        'password': 'a'
    }

    with flask_app.request_context(path='/account/register', data=form_data):
        vm = RegisterViewModel()

    # Act
    vm.validate()

    # Assert
    assert vm.error is None
```

To avoid assessing the database and causing errors, any function that is being tested that accesses the database can be 'mocked' (replaced so it is not called) using the 'unittest.mock' module; this replaces a function's behaviour

This is utilised by setting the target as the function to be replaced, and passing it into the 'unittest.mock.patch()' function and setting an argument 'return_value' to be the function that will be used, in most cases this will be 'None' as this will skip executing the function. Hence, a test can be executed in a with context which will mock a function's behaviour:

```
# Act
target = 'pypi_org.services.user_service.find_user_by_email'
with unittest.mock.patch(target, return_value=None):
    vm.validate()

# Assert
assert vm.error is None
```

This can be used for testing if a user of the same email already exists in the database:

```
# Act
target = 'pypi_org.services.user_service.find_user_by_email'
test_user = User(email=form_data.get('email'))
with unittest.mock.patch(target, return_value=test_user):
    vm.validate()

# Assert
assert vm.error is not None
assert 'already exists' in vm.error
```

Testing view models has the following syntax:

```
from pypi_org import app as flask_app

def test_register_validation_no_email():

    # Arrange
    from pypi_org.viewmodels.account.register_viewmodel import RegisterViewModel
    form_data = {
        'email': '',
        'password': 'a'
    }
    with flask_app.app.test_request_context(path='/account/register', data=form_data):
        vm = RegisterViewModel()

        # Act
        vm.validate()

    # Assert
    assert vm.error is not None
    assert 'email' in vm.error
```

Testing view methods can be done by making sure that the correct webpage is shown based on the request that is sent to the method, as well as testing that the right errors are returned. When testing view methods, it is assumed that the view models inside of the view methods are working correctly, as previous tests have been performed on them. This is how you can test that the register page view method is working correctly:

```
from pypi_org.views.account_views import register_post
form_data = {
    'name': 'Michael',
    'email': 'michael@talkpython.fm',
    'password': 'a' * 6
}

target = 'pypi_org.services.user_service.find_user_by_email'
find_user = unittest.mock.patch(target, return_value=None)
target = 'pypi_org.services.user_service.create_user'
create_user = unittest.mock.patch(target, return_value=User())
request = flask_app.test_request_context(path='/account/register', data=form_data)
with find_user, create_user, request:
    # Act
    resp: Response = register_post()

# Assert
assert resp.location == '/account'
```

A detailed view method test for testing the package details can be done as such:

```
def test_package_details_no_db(self):

    # Arrange
    from pypi_org.views.packages_views import project
    from pypi_org.data.packages import Package

    test_package = Package()
    test_package.id = 'sqlalchemy'
    test_package.releases = [
       Release(created_date=datetime.datetime.now(), major_ver=1, minor_ver=2, build_ver=200),
       Release(created_date=datetime.datetime.now() - datetime.timedelta(days=10)),
   ]
   with mock.patch('pypi_org.services.package_service.package_by_id', return_value=test_package):
       with mock.patch('pypi_org.services.package_service.releases_for_package', return_value=test_package.releases):
           with flask_app.app.test_request_context(path='/project/sqlalchemy'):

           # Act
           r = project(test_package.id) # calls package_service.package_by_id() ...

    # Assert
    web_package: Package = r.model['package']
    self.assertEqual(web_package.id, 'sqlalchemy')
    self.assertEqual(len(web_package.releases), 2)
```

Integration tests are used for making sure that certain webpages render correctly. To do this, a client should be passed as an argument to the test function, from there any webpage can be rendered and the status code can be tested:

```
def test_int_account_home_no_login(client):
    target = 'pypi_org.services.user_service.find_user_by_id'
    with unittest.mock.patch(target, return_value=None):
        resp: Response = client.get('/account')

    assert resp.status_code == 302
    assert resp.location == 'http://localhost/account/login'


def test_int_account_home_with_login(client):
    target = 'pypi_org.services.user_service.find_user_by_id'
    test_user = User(name='Michael', email='michael@talkpython.fm')
    with unittest.mock.patch(target, return_value=test_user):
        resp: Response = client.get('/account')

    assert resp.status_code == 200
    assert b'Michael' in resp.data
```

In the test_int_account_home_with_login function, the webpage response data is tested to see if it contains the binary data for ‘michael’, this is because the account page has the welcome message ‘welcome [user]’. All data is returned in binary and thus all string comparisons must be converted into binary

An ‘all_tests’ file can be created which when run, will run all of the tests. This is easily done by including all of the separate tests files:

```
import sys
import os

container_folder = os.path.abspath(os.path.join(
    os.path.dirname(__file__), '..'
))
sys.path.insert(0, container_folder)


# noinspection PyUnresolvedReferences
from account_tests import *
# noinspection PyUnresolvedReferences
from package_tests import *
# noinspection PyUnresolvedReferences
from sitemap_tests import *
# noinspection PyUnresolvedReferences
from home_tests import *
```

The code at the top of the function means that the script can be run from the command line

Sitemaps are XML files that tell search engines all of the webpages that they should browse and discover, these pages are either static or dynamic. These are large files which have this format:

```
-<url>
    <loc>http://127.0.0.1:5006/project/beautifulsoup4</loc>
    <lastmod>2014-01-21T05:35:05</lastmod>
    <changefreq>weekly</changefreq>
    <priority>1.0</priority>
  </url>
-<url>
    <loc>http://127.0.0.1:5006/project/boto</loc>
    <lastmod>2006-09-16T14:32:21</lastmod>
    <changefreq>weekly</changefreq>
    <priority>1.0</priority>
  </url>
-<url>
    <loc>http://127.0.0.1:5006/project/boto3</loc>
    <lastmod>2014-11-11T20:30:40</lastmod>
    <changefreq>weekly</changefreq>
    <priority>1.0</priority>
  </url>
```

These can be traversed and tested to see that all of the webpages load correctly:

```
def test_int_site_mapped_urls(client):
    text = get_sitemap_text(client)
    x = xml.etree.ElementTree.fromstring(text)
    urls = [
        href.text.strip().replace('http://127.0.0.1:5000', '').replace('http://localhost',
        for href in list(x.findall('url/loc'))
    ]
    urls = [
        u if u else '/'
        for u in urls
    ]
    print('Testing {} urls from sitemap...'.format(len(urls)), flush=True)

    has_tested_projects = False
    for url in urls:
        if '/project/' in url and has_tested_projects:
            continue

        if '/project/' in url:
            has_tested_projects = True
```

```
        print('Testing url at ' + url)
        resp: Response = client.get(url)
        assert resp.status_code == 200


def get_sitemap_text(client):
    # <urlset xmlns="http://www.sitemaps.org/schemas/sitemap/0.9">
    #     <url>
    #         <loc>http://talkpython.fm/episodes/show/37/python-cybersecurity-and-penetratio
    #         <lastmod>2015-12-08</lastmod>
    #         <changefreq>weekly</changefreq>
    #         <priority>1.0</priority>
    #     </url>
    #     <url>
    #         ...
    #     </url>
    res: Response = client.get("/sitemap.xml")
    text = res.data.decode("utf-8")
    text = text.replace('xmlns="http://www.sitemaps.org/schemas/sitemap/0.9"', '')
    return text
```

Robots.txt contains all of the webpages that search engines should not traverse

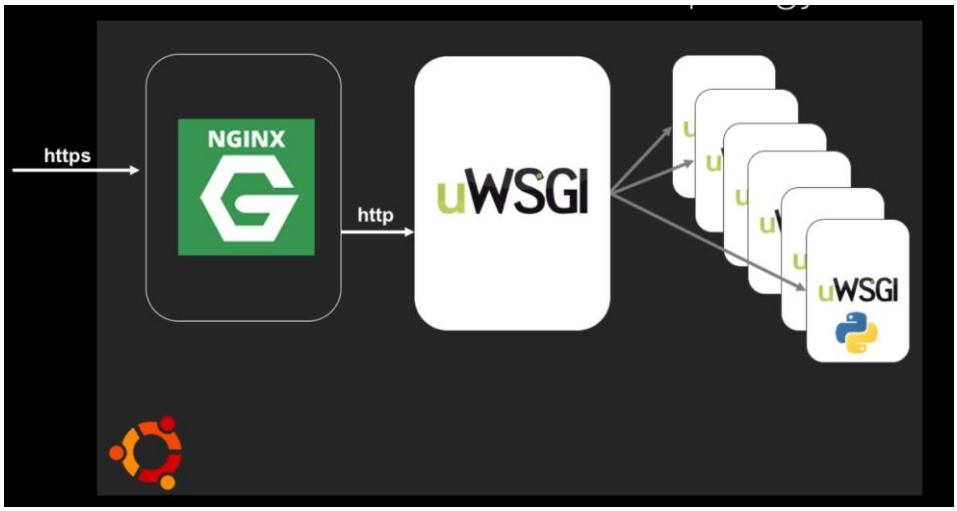
Deployment:

Ubunto server- hosts the website on the cloud

NGINX- the entity that users will actually interact with, this listens on port 80 and 443. This does not run the Python code, but rather sends the static files and delegates to the Python code

uWSGI- handles the Python requests (these can be duplicated multiple times to deal with different requests in parallel)

The following topology is what is used to run the flask server:



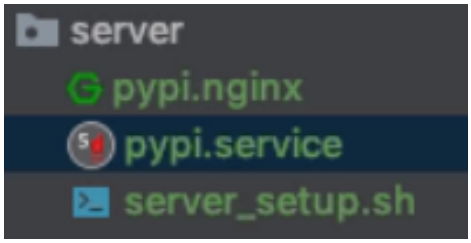
DigitalOcean- a hosting platform (can run a linux server)

Once the server is set up, using the static IP address, you can SSH connect to the server as the root user:

```
~ ssh root@104.248.77.15
The authenticity of host '104.248.77.15 (104.248.77.15)' can't be established.
ECDSA key fingerprint is SHA256:wbnDhII2EJ7wqJoOAxBx2Fv/IHpJ0U4+NB4JeMZb/q0.
Are you sure you want to continue connecting (yes/no)? yes
Warning: Permanently added '104.248.77.15' (ECDSA) to the list of known hosts.
```

As soon as the server is updated, all packages should be updated ‘apt update’ and ‘apt upgrade’. The server should be rebooted using the ‘reboot’ command

A new server directory should be established which has the following files:



All files should be pushed to the database using GIT

Server_setup.sh is a script that is run which sets the server up into a state where it can function correctly

When uWSGI runs the python app, it only reads the app = flask.flask(__name__) and run the app with its own settings, this means that none of the blueprints are set up and the database is not initialised. Hence the following code should be altered:

```
def main():
    configure()
    app.run(debug=True, port=5006)


def configure():
    print("Configuring Flask app:")

    register_blueprints()
    print("Registered blueprints")

    setup_db()
    print("DB setup completed.")
    print("", flush=True)


def setup_db():
    db_file = os.path.join(
        os.path.dirname(__file__),
        'db',
        'pypi.sqlite')
```

```python
def register_blueprints():
    from pypi_org.views import home_views
    from pypi_org.views import package_views
    from pypi_org.views import cms_views
    from pypi_org.views import account_views
    from pypi_org.views import seo_view

    app.register_blueprint(package_views.blueprint)
    app.register_blueprint(home_views.blueprint)
    app.register_blueprint(account_views.blueprint)
    app.register_blueprint(seo_view.blueprint)
    app.register_blueprint(cms_views.blueprint)


if __name__ == '__main__':
    main()
else:
    configure()
```

This means that if uWSGI is running the script, it will just configure the database and blueprints and run the app with its own settings

All of the code in the server setup file should be run on the SSH command line to make sure that it works correctly. This file has the following contents:

```
#!/usr/bin/env bash

# Consider running these two commands separately
# Do a reboot before continuing.
apt update
apt upgrade -y

apt install zsh
sh -c "$(curl -fsSL https://raw.github.com/robbyrussell/oh-my-zsh/master/tools/install.sh)"

# Install some OS dependencies:
sudo apt-get install -y -q build-essential git unzip zip nload tree
sudo apt-get install -y -q python3-pip python3-dev python3-venv
sudo apt-get install -y -q nginx
# for gzip support in uwsgi
sudo apt-get install --no-install-recommends -y -q libpcre3-dev libz-dev

# Stop the hackers
sudo apt install fail2ban -y

ufw allow 22
ufw allow 80
ufw allow 443
ufw enable
```

'Apt update'- makes sure that the server is correctly configured

'Apt install zsh'- a package that makes the UI better

'Sudo apt-get install build-essential' – git (for version control), zip/unzip (for compression), nload (for measuring server traffic), tree (for looking at directory structure)

'Sudo apt-get install python3'- for installing python and libraries

'sudo nginx'- for managing incoming requests

'sudo apt-get install libz'- used for compressing files in uWSGI

'sudo apt install fail2ban'- for watching if users try to login over SSH and fail, and if they fail multiple times they will get blacklisted and will not be able to access the website

'ufw allow'- exposes certain ports that are necessary (22 for SSH, 80 for HTTP, 443 for HTTPS). 22 must be included or the server will never be able to be accessed

```
# Basic git setup
git config --global credential.helper cache
git config --global credential.helper 'cache --timeout=720000'

# Be sure to put your info here:
git config --global user.email "you@email.com"
git config --global user.name "Your name"

# Web app file structure
mkdir /apps
chmod 777 /apps
mkdir /apps/logs
mkdir /apps/logs/pypi
mkdir /apps/logs/pypi/app_log
cd /apps

# Create a virtual env for the app.
cd /apps
python3 -m venv venv
source /apps/venv/bin/activate
pip install --upgrade pip setuptools
pip install --upgrade httpie glances
pip install --upgrade uwsgi
```

Git config 'credential.helper'- means that git login details are remembered for a month before having to be re-entered

Git config user – means that the git manager can interact (push/pull) from the server

#Web app file structure – makes the file structure for the app to run in (logs have not been implemented yet but it can be used if needed)

#Create virtual env- creates the virtual environment for the webserver. This is not strictly necessary for the server as this server is dedicated for running the website, however it means that if something goes wrong with the virtual environment, it can be removed and started again

-upgrade pip setuptools : updates pip

-upgrade httpie glances : shows all the processes being run on the server (this can be activated by running the command 'glances')

-upgrade uwsgi : installs uwsgi for running the server

Whenever the user logs out of SSH, the virtual environment is shut down. To automatically run the virtual environment, the command 'nano .zshrc' (or 'nano zbashrc' if connecting via bash) should be run and the command 'source /apps/venv/bin/activate' should be appended to the end of the file. This will activate the virtual environment whenever the user logs in

```
# clone the repo:
cd /apps
git clone https://github.com/talkpython/data-driven-web-apps-with-flask app_repo

# Setup the web app:
cd cd /apps/app_repo/app/ch15_deploy/final/
pip install -r requirements.txt

# Copy and enable the daemon
cp /apps/app_repo/app/ch15_deploy/final/server/pypi.service /etc/systemd/system/pypi.service

systemctl start pypi
systemctl status pypi
systemctl enable pypi

# Setup the public facing server (NGINX)
apt install nginx

# CAREFUL HERE. If you are using default, maybe skip this
rm /etc/nginx/sites-enabled/default

cp /apps/app_repo/app/ch15_deploy/final/server/pypi.nginx /etc/nginx/sites-enabled/pypi.nginx
update-rc.d nginx enable
```

This next part is used for setting up the files on the server

Git clone- used to clone all of the files onto the server

Pip install -r requirements.txt : installs all of the modules needed for running the website

At this point, the app should be able to be run 'python [app_path/app.py]. Next we need to make the server work under uWSGI. The next command changed the working directory to the service file which will look as follows:

```
[Unit]
Description=uWSGI PyPI server instance
After=syslog.target

[Service]
ExecStart=/apps/venv/bin/uwsgi -H /apps/venv --master --processes 4 --threads 2 --http :5000
RuntimeDirectory=/apps/app_repo/app/ch15_deploy/final/
Restart=always
KillSignal=SIGQUIT
Type=notify
StandardError=syslog
NotifyAccess=all

[Install]
WantedBy=multi-user.target
```

[Continued ExecStart]

```
--manage-script-name --python-path /apps/app_repo/app/ch15_deploy/final --mount /=wsgi:app
```

Next a new file should be created at the top app directory (which is specified by the RuntimeDirectory signifier). This is called ‘wsgi.py’ and has the following content:

```
from pypi_org.app import app, main

if __name__ == '__main__':
    main()
```

Next the ExecStart command should be run (as cd and the file path, not cp) to check that the script works, this command should be run while in the RuntimeDirectory

This should produce the following output:

```
Configuring Flask app:
Registered blueprints
Connecting to DB with sqlite:////apps/app_repo/app/ch15_deploy/final/pypi_org/db/pypi.sqlite
DB setup completed.

WSGI app 0 (mountpoint='/') ready in 1 seconds on interpreter 0x5652adb31bb0 pid: 10896 (default app)
uWSGI running as root, you can use --uid/--gid/--chroot options
*** WARNING: you are running uWSGI as root !!! (use the --uid flag) ***
*** uWSGI is running in multiple interpreter mode ***
spawned uWSGI master process (pid: 10896)
spawned uWSGI worker 1 (pid: 10898, cores: 2)
spawned uWSGI worker 2 (pid: 10899, cores: 2)
spawned uWSGI worker 3 (pid: 10900, cores: 2)
spawned uWSGI worker 4 (pid: 10901, cores: 2)
spawned uWSGI http 1 (pid: 10902)
```

By opening another SSH connection and running the command ‘http localhost:5000’, the website data should be returned

Next the #Copy and enable the daemon commands should be executed

Cp … … - is used to copy the script so that the service is run as a system app

Systemctl start [app] - will run the app

Systemctl status [app] – show the status of the app

Systemctl enable [app] – means that the app will still run after the SSH connection is lost

Nginx- used for sending static files and managing SSLs

Apt install nginx- used to setup the server

Rm … is used for removing the default webpage that is displayed

Cp … …- this copies the nginx configuration files over to the server (many nginx files can be used for running many different websites)

```
server {
    listen 80;
    server_name fake_pypi.com;
    server_tokens off;
    charset utf-8;
    client_max_body_size 150M;

    location /static {
        gzip            on;
        gzip_buffers    8 256k;
        uwsgi_buffers   8 256k;

        alias /apps/app_repo/app/ch15_deploy/final/pypi_org/static;
        expires 365d;
    }
    location / {
        try_files $uri @yourapplication;
    }
```

The server listens on port 80 under the domain ‘fake_pypi.com’ (this can be changed to the domain that is being used)

Server_tokens off- does not pass back the version of the server

Location /static- shows the location of all the static files on the server and sets the expiry date that they are cached to be 365 days (these are not handled by python any more but are handled by nginx

Location /- says if static files are not being referenced, go to the yourapplication

Location @yourapplication- sets up a local callback to 127.0.0.1:5000

Update-rc.d nginx enable- enables nginx on the system

Service nginx restart – restarts the service so it detects the configuration page

Adding SSL

Firstly the domain needs to be configured to the server for SSL to work

Setting up SSL can be done as follows (using Let's Encrypt):

```
# Optionally add SSL support via Let's Encrypt:
# https://www.digitalocean.com/community/tutorials/how-to-secure-nginx-with-let-s-encrypt-

add-apt-repository ppa:certbot/certbot
apt install python-certbot-nginx
certbot --nginx -d fakepypi.talkpython.com
```

Add-apt-repository ppa:certbot/certbot – registers the package authority

Apt install … - installs the certbot module

Certbot … - configures certbot to work with the nginx and the domain

This will produce the following statements (the correct inputs have also been entered as follows):

```
Plugins selected: Authenticator nginx, Installer nginx
Enter email address (used for urgent renewal and security notices) (Enter 'c' to
cancel): michael@talkpython.fm

- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
Please read the Terms of Service at
https://letsencrypt.org/documents/LE-SA-v1.2-November-15-2017.pdf. You must
agree in order to register with the ACME server at
https://acme-v02.api.letsencrypt.org/directory
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
(A)gree/(C)ancel: A

- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
Would you be willing to share your email address with the Electronic Frontier
Foundation, a founding partner of the Let's Encrypt project and the non-profit
organization that develops Certbot? We'd like to send you email about our work
encrypting the web, EFF news, campaigns, and ways to support digital freedom.
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
(Y)es/(N)o: N
Obtaining a new certificate
Performing the following challenges:
http-01 challenge for fakepypi.talkpython.com
Waiting for verification...
Cleaning up challenges
Deploying Certificate to VirtualHost /etc/nginx/sites-enabled/pypi.nginx

Please choose whether or not to redirect HTTP traffic to HTTPS, removing HTTP acces
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
1: No redirect - Make no further changes to the webserver configuration.
2: Redirect - Make all requests redirect to secure HTTPS access. Choose this for
new sites, or if you're confident your site works on HTTPS. You can undo this
change by editing your web server's configuration.
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
Select the appropriate number [1-2] then [enter] (press 'c' to cancel): 2
```

This will go through the configuration file and add all the SSL setup commands.

These certificates are valid for about 90 days before they need to be renewed

uWSGI Configuration summary:

```
# pypi.service
[Unit]
Description=uWSGI PyPI server instance
After=syslog.target

[Service]
ExecStart=/apps/venv/bin/uwsgi -H /apps/venv --master --processes 4 --threads 2 --http :5000 --
manage-script-name --python-path /apps/app_repo/app/ch15_deploy/final --mount /=wsgi:app
RuntimeDirectory=/apps/app_repo/app/ch15_deploy/final/

Restart=always
KillSignal=SIGQUIT
Type=notify
StandardError=syslog
NotifyAccess=all

[Install]
WantedBy=multi-user.target
```

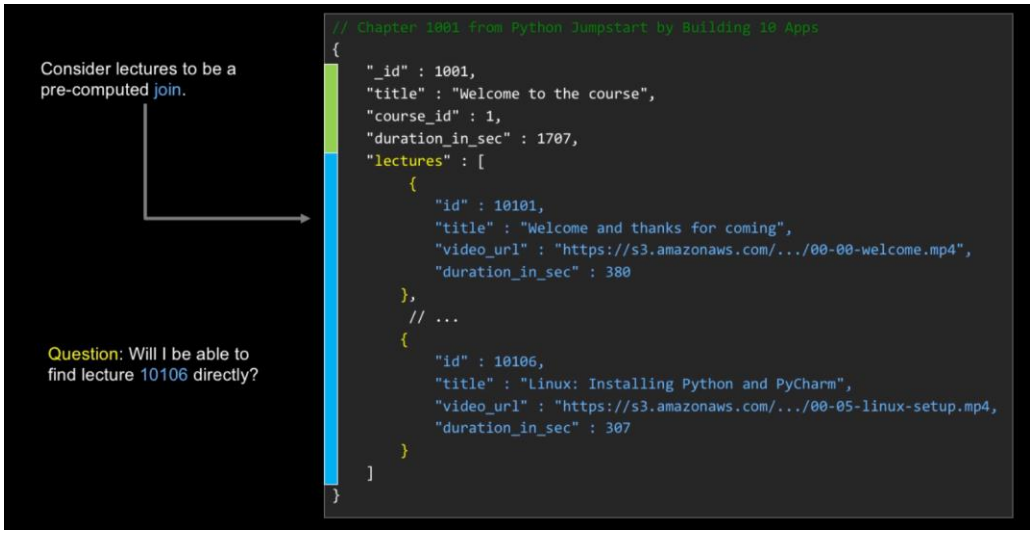
Nginx site configuration summary:

```
# pypi.nginx
server {
    listen 80;
    server_name fakepypi.talkpython.com;
    server_tokens off;

    charset utf-8;
    client_max_body_size 1M;

    location /static {
        gzip            on;
        gzip_buffers    8 256k;
        uwsgi_buffers   8 256k;

        alias /apps/app_repo/app/ch15_deploy/final/pypi_org/static;
        expires 365d;
    }

    # ...
}
```

```
server {
    # ...

    location / {
        try_files $uri @yourapplication;
    }

    location @yourapplication {
        gzip            on;
        gzip_buffers    8 256k;
        uwsgi_buffers   8 256k;

        server_tokens off;
        include uwsgi_params;
        proxy_set_header Host $host;
        proxy_set_header real_scheme $scheme;
        proxy_set_header X-Forwarded-Protocol $scheme;
        proxy_set_header X-Real-IP $remote_addr;
        proxy_pass http://127.0.0.1:5000;
    }
}
```

MongoDB:

This is an example of a document database, meaning it is not relational. MongoDB has the following format:



Mongoengine- a module used for working with MongoDB. The database can be setup as:

```python
import ssl

import mongoengine


def global_init(user=None, password=None, port=27017, server='localhost', use_ssl=True, db_name=
    if user or password:
        # noinspection PyUnresolvedReferences
        data = dict(
            username=user,
            password=password,
            host=server,
            port=port,
            authentication_source='admin',
            authentication_mechanism='SCRAM-SHA-1',
            ssl=use_ssl,
            ssl_cert_reqs=ssl.CERT_NONE)
        mongoengine.register_connection(alias='core', name=db_name, **data)
        data['password'] = '************'
        print(" --> Registering prod connection: {}".format(data))
    else:
        print(" --> Registering dev connection")
        mongoengine.register_connection(alias='core', name=db_name)
```

The SQL entity classes are written as:

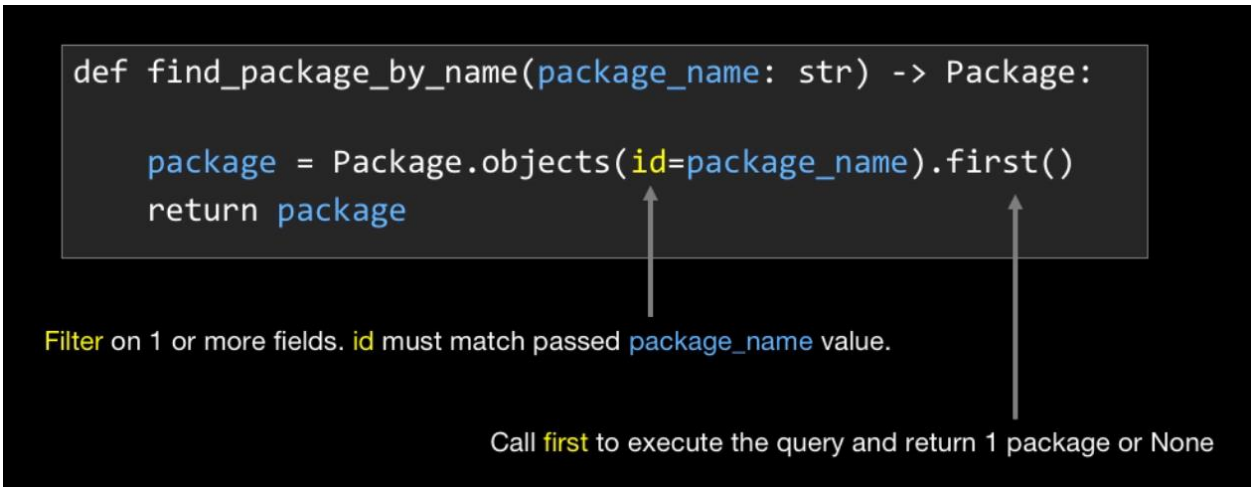
```python
import datetime
import mongoengine


class User(mongoengine.Document):
    name = mongoengine.StringField()
    email = mongoengine.StringField(unique=True)
    hashed_password = mongoengine.StringField()
    created_date = mongoengine.DateTimeField(default=datetime.datetime.now)

    meta = {
        'collection': 'users',
        'db_alias': 'core',
        'indexes': [
            'email',
            'hashed_password',
            'created_date',
        ]
    }
```

The database can therefore be created and a user can be created as such:

```python
def setup_db():
    mongo_setup.global_init()

    user = User()
    user.name = 'Michael Kennedy'
    user.email = 'michael@talkpython.fm'

    user.save()
```

Robo 3T- used for viewing MongoDB database and records

MongoDB does not need to worry about relationships, and it does not need to worry about sessions. Hence, querying the database is much simpler than SQLAlchemy:



Migrating users from SQL to MongoDB can be implemented as follows:

```
def migrate_users():
    if MongoUser.objects().count():
        return

    session = db_session.create_session()
    sql_users = session.query(SqlUser).all()
    for sut in sql_users:
        su: SqlUser = sut
        u = MongoUser()
        u.created_date = su.created_date
        u.hashed_password = su.hashed_password
        u.name = su.name
        u.email = su.email
        u.save()
```