
neueda
Python Database Access

Contents

The Basic Steps

- Install the mysql-connector-python module
- Connecting to a MySQL database
- Executing a query

Install the mysql-connector-python module

1. We need to install the mysql-connector-python package:

```
C:\Users\Administrator>pip install mysql-connector-python
```

2. This installs the library needed to connect to MySQL from Python
3. We use the connect method to connect to a MySQL database:

```
import mysql.connector
mydb = mysql.connector.connect(
  host="localhost",
  user="root",
  password="c0nygre",
  database="sakila"
)
print(mydb)
```

neueda

Connecting to a MySQL database

1. Once we have connected we can use the cursor method to return a cursor.
2. The cursor can be used to execute SQL statements.
3. This code executes a SELECT statement that returns a collection that holds the results. We can loop through the results to display them.

```
# create a cursor object to execute queries
mycursor = mydb.cursor()
# execute a query
mycursor.execute("SELECT category_id, name FROM category")
# fetch all the results from the query
result = mycursor.fetchall()
# print the results
for row in result:
    print(row)
```

neueda

Executing a query

- Running the code in the previous slides will give this output:

```
<mysql.connector.connection_cext.CMySQLConnection object at 0x000001E7A0270DC0>
(1, 'Action')
(2, 'Animation')
(3, 'Children')
(4, 'Classics')
(5, 'Comedy')
(6, 'Documentary')
(7, 'Drama')
(8, 'Family')
(9, 'Foreign')
(10, 'Games')
(11, 'Horror')
(12, 'Music')
(13, 'New')
(14, 'Sci-Fi')
(15, 'Sports')
(16, 'Travel')
```

Creating a database

- Any SQL statements can be executed using a cursor object including the one used to create a database.

```
import mysql.connector
# establish connection to the MySQL server
mydb = mysql.connector.connect(
  host="localhost",
  user="root",
  password="c0nygre"
)
# create a cursor object
mycursor = mydb.cursor()
# create the products database
mycursor.execute("CREATE DATABASE IF NOT EXISTS velo")
```

Creating tables

- To create a table we use the SQL create table statement. Here we will create a products table with an auto_increment id column.

```
mycursor.execute("USE velo")
# create the products table
mycursor.execute('''CREATE TABLE products(
    id int auto_increment PRIMARY KEY,
    name varchar(30),
    price decimal(10,2)
)''')

print("Table created")
```

Inserting data

- We can use a cursor object to execute SQL insert statements
- Here some records are added to a table using
  - execute()
  - commit()
- The commit method will ‘commit’ the changes to the database
  - Don’t forget it otherwise nothing will change!

```
import mysql.connector
mydb = mysql.connector.connect(
  host="localhost",
  user="root",
  password="c0nygre",
  database="velo"
)
mycursor = mydb.cursor()
mycursor.execute('''INSERT INTO
                 products(name, price)
                 VALUES('Velo Road Bike', 1500),
                 ('Velo Mountain Bike',2500),
                 ('Velo Touring Bike', 1200);
                 ''')
mydb.commit()
print(mycursor.rowcount, "records inserted.")
```

neueda

Inserting data

- When doing inserts you can also use parameters which are provided in a collection (tuple)

```
import mysql.connector
mydb = mysql.connector.connect(
  host="localhost",
  user="root",
  password="c0nygre",
  database="velo"
)
mycursor = mydb.cursor()
sql = "INSERT INTO products (name, price) VALUES (%s, %s)"
data = ("Velo BMX Bike", 900)
mycursor.execute(sql, data)
mydb.commit()
print(mycursor.rowcount, "record inserted.")
```

Retrieving data

- We use a select statement with a cursor to retrieve data
- The **fetchall** method returns the records in a collection which we can iterate through using a for loop

```
import mysql.connector
mydb = mysql.connector.connect(
  host="localhost",
  user="root",
  password="c0nygre",
  database="velo"
)
mycursor = mydb.cursor()
mycursor.execute("SELECT * FROM products")
products = mycursor.fetchall()
for product in products:
    print(product)
```

```
C:\Courses\PythonREST>python select-data.py
(1, 'Velo Road Bike', Decimal('1500.00'))
(2, 'Velo Mountain Bike', Decimal('2500.00'))
(3, 'Velo Touring Bike', Decimal('1200.00'))
(4, 'Velo BMX Bike', Decimal('900.00'))
```

neueda

Retrieving data

- Here we use a parameter with a WHERE clause to filter the more expensive products

```
print("**** Expensive Products ****")
sql = "SELECT * FROM products WHERE price >= %s"
data = (1500,)
mycursor.execute(sql,data)
products = mycursor.fetchall()
for product in products:
    print(product)
```

```
C:\Courses\PythonREST>python select-data.py
(1, 'Velo Road Bike', Decimal('1500.00'))
(2, 'Velo Mountain Bike', Decimal('2500.00'))
(3, 'Velo Touring Bike', Decimal('1200.00'))
(4, 'Velo BMX Bike', Decimal('900.00'))
**** Expensive Products ****
(1, 'Velo Road Bike', Decimal('1500.00'))
(2, 'Velo Mountain Bike', Decimal('2500.00'))
```

neueda

Updating data

- We can use a cursor object to execute SQL update statements.
- Here a record is updated using the cursor execute method and the connection commit method

```
import mysql.connector
mydb = mysql.connector.connect(
  host="localhost",
  user="root",
  password="c0nygre",
  database="velo"
)
mycursor = mydb.cursor()
mycursor.execute('''UPDATE products
                SET name = 'Velo Carbon Road Bike'
                WHERE id = 1
                ''')
mydb.commit()
print(mycursor.rowcount, "records changed.")
```

neueda

Updating data

- In this example parameters are used with the update statement

```
sql = "UPDATE products SET name = %s WHERE
id = %s"
data = ("Velo FS Mountain Bike",2)
mycursor.execute(sql,data)
mydb.commit()
print(mycursor.rowcount, "records
changed.")
```

Deleting data

- We can use a cursor object to execute SQL delete statements
- Here a record is deleted using the cursor execute method and the connection commit method
- As with other CRUD statements parameters can be used

```
import mysql.connector
mydb = mysql.connector.connect(
  host="localhost",
  user="root",
  password="c0nygre",
  database="products"
)
mycursor = mydb.cursor()
sql = "DELETE FROM products WHERE id = 4"
mycursor.execute(sql)
mydb.commit()
print(mycursor.rowcount, "record(s) deleted")
```

neueda

Summary

- Getting Started
- Creating a database
- Creating tables
- Inserting data
- Retrieving data
- Updating data
- Deleting data


neueda
Introduction to Flask

Objectives

- What is Flask?
- Installation
- Getting started
- URLs and Views
- Enterprise Applications using Databases
- Processing Forms
- Handling static content
- Restful Flask

neueda

What is Flask

- Flask is a very lightweight and easy to use framework to make it quick and easy to create Web based applications using Python
- It is an alternative to the more complex and heavyweight Django framework

- Pinterest and LinkedIn would be well known examples of those that use the Flask framework

neueda

Installation

- Install Python
  - 3.2.x or higher
- Install Flask using pip
  - pip install flask

neueda

Create a Minimal App

- To create a simple Flask ‘hello world’ example

```
from flask import Flask

app = Flask(__name__)

@app.route('/')
def hello_world():
    return 'Hello World!'


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

- Now visit http://localhost:5000
  - It’s all up and running already!

neueda

Routing

- Flask applications map methods to routes using **@app.route** statements

```
@app.route('/')
def hello_world():
    return 'Hello World!'
```

- Multiple methods can be provided for different URL patterns

```
@app.route('/')
def index():
  return 'Index Page'
@app.route('/hello')
def hello():
  return 'Hello, World'
```


neueda

Parameters

- Aspects of the URL can easily be used as parameters to the methods

```
@app.route('/user/<username>')
def show_user_profile(username):
   # show the user profile for that user
   return 'User %s' % username

@app.route('/post/<int:post_id>')
def show_post(post_id):
   # show the post with the given id, the id is an integer
   return f"Post: {post_id}"
```

- Type information can also be specified
  - Such as **int**:post_id

Parameter Types

- The following types are supported

string	accepts any text without a slash (the default)
int	accepts integers
float	like int but for floating point values
path	like the default but also accepts slashes
any	matches one of the items provided
uuid	accepts UUID strings

Handling HTTP Methods

- Methods can also be routed based on HTTP method

```
@app.route('/login', methods=['GET', 'POST'])
def login():
    if request.method == 'POST':
       do_the_login()
    else:
       show_the_login_form()
```

- All HTTP methods are supported
  - Get, Post, Put, Delete, Options, Head

neueda

Templating

- Flask incorporates the Jinja2 templating engine for HTML content generation
  - Templates have access to session, request, and other commonly used variables

```
@app.route('/hello/')
@app.route('/hello/<name>')
def hello(name=None):
    return render_template('hello.html', name=name)
```

templates/hello.html

```
<!doctype html>
<title>Hello from Flask</title>
{% if name %}
<h1>Hello {{ name }}!</h1>
{% else %}
<h1>Hello, World!</h1>
{% endif %}
```


neueda

Creating an Enterprise App

- We have seen the basics, so now lets see how to create a database aware enterprise application
  - We will use SQLite as the database
- The example is based on the official Flask tutorial
  - https://flask.palletsprojects.com/en/1.1.x/tutorial/

Enterprise App Folders

- The folder structure required will be
  - EnterpriseApp/templates - for our page templates
  - EnterpriseApp/static - for static assets such as images/css and so on
  - EnterpriseApp/schema - for our database schema

neueda

Creating the Skeleton Application

- Within the EnterpriseApp project folder you can create an EnterpriseApp.py file

```
import os
import sqlite3
from flask import Flask, request, session, g, redirect, url_for, abort, render_template, flash

app = Flask(__name__)
app.config.from_object(__name__)  # load config from below in this file

# Here is the config that will be loaded by the above
app.config.update(dict(
    DATABASE=os.path.join(app.root_path, 'enterprise.db'),
    SECRET_KEY='development key',
    USERNAME='admin',
    PASSWORD='default'
))
# can use a config file instead with this name
app.config.from_envvar('ENTERPRISE_SETTINGS', silent=True)
```

Managing Database Connections

- Database connections can be created using a simple function

```
def connect_db():
    """Connects to the specific database."""
    rv = sqlite3.connect(app.config['DATABASE'])
    rv.row_factory = sqlite3.Row
    return rv
```

- To optimise performance the connection can be cached

```
def get_db():
    if not hasattr(g, 'sqlite_db'):
        g.sqlite_db = connect_db()
    return g.sqlite_db
```

- Note that to be scalable, you would most likely want a database connection pool
  - Connection pooling is beyond the scope of this course
  - See this StackOverflow Blog post for more information
    - https://stackoverflow.blog/2020/10/14/improve-database-performance-with-connection-pooling/

neueda

What is g?

- What exactly is g?

```
def get_db():
    if not hasattr(g, 'sqlite_db'):
        g.sqlite_db = connect_db()
    return g.sqlite_db
```

- The g variable is for global variables where the value is valid for the application
  - https://flask.palletsprojects.com/en/2.0.x/appcontext/

neueda

Closing Connections

- Connections can be closed at the end using a decorated function

```
@app.teardown_appcontext
def close_db(error):
    if hasattr(g, 'sqlite_db'):
        g.sqlite_db.close()
```


neueda

Handling Views

- Below is a view function to return the list of items

```
@app.route('/')
def show_entries():
    db = get_db()
    cur = db.execute('select title, text from entries order by id desc')
    entries = cur.fetchall()
    return render_template('show_entries.html', entries=entries)
```

The Default Template

- Here is a template for the list of items that also contains a form to allow for the creation of new items

```
{% extends "layout.html" %}
{% block body %}
  {% if session.logged_in %}
    <form action="{{ url_for('add_entry') }}" method=post class=add-entry>
      <dl>
        <dt>Title:
        <dd><input type=text size=30 name=title>
        <dt>Text:
        <dd><textarea name=text rows=5 cols=40></textarea>
        <dd><input type=submit value=Share>
      </dl>
    </form>
  {% endif %}
  <ul class=entries>
  {% for entry in entries %}
    <li><h2>{{ entry.title }}</h2>{{ entry.text|safe }}
  {% else %}
    <li><em>No entries here so far</em>
  {% endfor %}
  </ul>
{% endblock %}
```


neueda

The ‘Base’ Layout

- Layouts can extend other layouts with the base layout containing common content

templates/layout.html

```
<!doctype html>
<title>Enterprise Example</title>
<link rel=stylesheet type=text/css href="{{ url_for('static', filename='style.css') }}">
<div class=page>
  <h1>Enterprise Example</h1>
  <div class=metanav>
  {% if not session.logged_in %}
    <a href="{{ url_for('login') }}">log in</a>
  {% else %}
    <a href="{{ url_for('logout') }}">log out</a>
  {% endif %}
  </div>
  {% for message in get_flashed_messages() %}
    <div class=flash>{{ message }}</div>
  {% endfor %}
  {% block body %}{% endblock %}
</div>
```


neueda

Handling New Entries

- When the form is submitted, the following function will be triggered

```
@app.route('/add', methods=['POST'])
def add_entry():
    if not session.get('logged_in'):
        abort(401)
    db = get_db()
    db.execute('insert into entries (title, text) values (?, ?)',
               [request.form['title'], request.form['text']])
    db.commit()
    flash('New entry was successfully posted')
    return redirect(url_for('show_entries'))
```

- Flash messages can be shown in the templates

```
{% for message in get_flashed_messages() %}
    <div class=flash>{{ message }}</div>
  {% endfor %}
```


neueda

Logging In and Out

- Simple logging in and out can be handled by routing functions

```
@app.route('/login', methods=['GET', 'POST'])
def login():
    error = None
    if request.method == 'POST':
        if request.form['username'] != app.config['USERNAME']:
            error = 'Invalid username'
        elif request.form['password'] != app.config['PASSWORD']:
            error = 'Invalid password'
        else:
            session['logged_in'] = True
            flash('You were logged in')
            return redirect(url_for('show_entries'))
    return render_template('login.html', error=error)
```

```
@app.route('/logout')
def logout():
    session.pop('logged_in', None)
    flash('You were logged out')
    return redirect(url_for('show_entries'))
```

```
<h2>Login</h2>
<form action="{{ url_for('login') }}" method=post>
    <dl>
        <dt>Username:
        <dd><input type=text name=username>
        <dt>Password:
        <dd><input type=password name=password>
        <dd><input type=submit value=Login>
    </dl>
</form>
```


neueda

Styling

- A CSS can be placed in the static folder and then referenced from the layout.html page

static/style.css

```
body            { font-family: sans-serif; background: #eee; }
a, h1, h2       { color: #377ba8; }
h1, h2          { font-family: 'Georgia', serif; margin: 0; }
h1              { border-bottom: 2px solid #eee; }
h2              { font-size: 1.2em; }
```

templates/layout.html

```
<!doctype html>
<title>Enterprise Example</title>
<link rel=stylesheet type=text/css href="{{ url_for('static', filename='style.css')
}}">
<div class=page>
..
```


neueda

REST API Creation

- Flask can also be used to create a REST API via an extension called **Flask-Restful**
- A minimal app is shown below

```
from flask import Flask
from flask_restful import Resource, Api

app = Flask(__name__)
api = Api(app)

class HelloWorld(Resource):
    def get(self):
        return {'hello': 'world'}

api.add_resource(HelloWorld, '/')

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


neueda

Installation

- The installation process is trivial by simply running
  - **pip install flask-restful**

neueda

Flask Restful Resources

- When using Flask Restful you can specify **resources**
- Resources can easily have HTTP methods applied to them

```
class TodoSimple(Resource):
    def get(self, todo_id):
        return {todo_id: todos[todo_id]}

    def put(self, todo_id):
        todos[todo_id] = request.form['data']
        return {todo_id: todos[todo_id]}

api.add_resource(TodoSimple, '/<string:todo_id>')
```


neueda

Testing with the requests Library

- The REST API can then be tested with the requests library from Python

```
>>>from requests import put, get
>>>put('http://localhost:5000/todo1', data={'data': 'Remember the milk'}).json()
{u'todo1': u'Remember the milk'}
>>>get('http://localhost:5000/todo1').json()
{u'todo1': u'Remember the milk'}
```


neueda

Setting Response Codes and Headers

- It is also possible to set response codes and headers

```
class Todo1(Resource):
    def get(self):
        # Default to 200 OK
        return {'task': 'Hello world'}

class Todo2(Resource):
    def get(self):
        # Set the response code to 201
        return {'task': 'Hello world'}, 201

class Todo3(Resource):
    def get(self):
        # Set the response code to 201 and return custom headers
        return {'task': 'Hello world'}, 201, {Content-type': ‘application/json'}
```

Complete Example

- To see a complete CRUD type example, view FlaskRestfulFullCrud.py in the demos which is taken from
  - http://flask-restful.readthedocs.io/en/latest/quickstart.html#full-example

neueda

Summary

- What is Flask?
- Installation
- Getting started
- URLs and Views
- Enterprise Applications using Databases
- Processing Forms
- Handling static content
- Restful Flask

neueda

Appendix: Creating the Database

- The database can also be created using flask
- A function can be added with a decorator than enables it to be run using the flask CLI

```
def init_db():
    db = get_db()
    with app.open_resource(‘schema/schema.sql', mode='r')
as f:
        db.cursor().executescript(f.read())
    db.commit()

@app.cli.command('initdb')
def initdb_command():
    init_db()
    print('Initialized the database.')
```


neueda

Running the flask CLI

- To use the CLI, an environment variable can be set referring to your entry point python file

```
set FLASK_APP=EnterpriseFlask.py
```

- Then the CLI can be invoked to call your new decorated function

```
flask initdb
```


neueda


neueda
RESTful Web Services

Objectives

- Introducing REST
- The principles of REST
- Implementing a REST Web service

Introducing REST

- REST stands for **Representational State Transfer**
- REST is a style of architecture
  - The Web is a large set of resources
  - As a client selects a resource, they are then in a particular state
  - The client can then select a different resource, and be in a different state as they now have a different resource
- For example, drilling into eBay to find a specific item

neueda

Identifying Resources

- One key concept in REST is the concept of a resource
- A resource is always addressable by a URL
  - http://www.conygre.com/courseList
- A retrieved resource contains URLs of other resources, which allows you to get more information (and therefore change state)

neueda

REST URLs

- One important principle in REST is that the URLs are logical names for resources, not physical locations
- The actual location should be resolved by technology on the server
  - Such as a Spring MVC Controller

The REST Principles

- There is a request and a response
- Resources are all named through URLs
- Resources provide links to other resources
- There is a standard interface into the application using one of the HTTP methods

HTTP Method	Role in REST
GET	Retrieve data
POST	Add data
PUT	Edit data
DELETE	Remove data

REST and Web Services

- Web services that use REST tend to have the following characteristics
  - Clients send HTTP requests using the four HTTP methods
  - Simple parameters can be passed in the URL
  - Complex parameters are passed in the body as JSON
    - Other formats are sometimes used such as XML

neueda

Designing a RESTful API

- APIs can be developed using REST
- For example
  - GET www.meals.co.uk/restaurants?location=bristol
    - Get a list of Bristol restaurants
  - GET www.meals.co.uk/restaurants/2554
    - Get details for the Burger Joint Restaurant (restaurant #2554)
  - GET www.meals.co.uk/orders/454
    - Get the details for order number 454
  - DELETE www.meals.co.uk/orders/454
    - Remove order number 454
  - POST www.meals.co.uk/orders
    - (provide a JSON food order in the request body)
    - Create a new order

Security with REST

- With the URLs for RESTful APIs very simple and even guessable, it can be important to secure the API
- Approaches can include
  - Providing a user token to use with each request
  - Enforcing client authentication

neueda

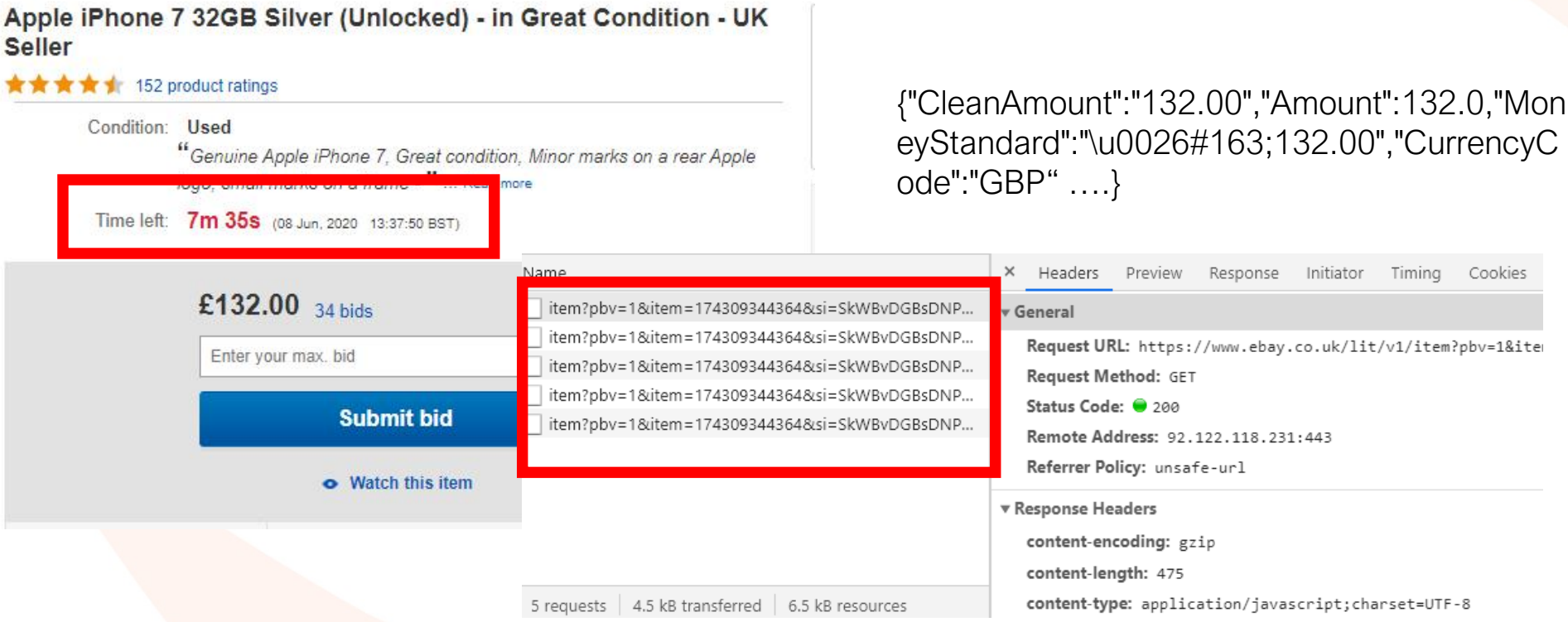
Benefits of REST

- REST based services are simple to implement
- Frameworks exist for multiple languages and platforms
  - Django / Flask for Python
  - ASP.NET API for .NET
  - Spring MVC for Java
  - Express for NodeJS
- REST based services are straightforward to invoke

neueda

Ebay Example

- The pricing is updated by frequent calls to a REST API to get the latest price



{"CleanAmount":"132.00","Amount":132.0,"MoneyStandard":"\u0026#163;132.00","CurrencyCode":"GBP" ….}

neueda

How Mature is Your REST?

- Implementations of REST APIs vary in terms of the their level of maturity
- The Richardson Maturity Model defines four levels of REST
  - Level 0 – Just use HTTP as a RPC protocol
  - Level 1 – Just use POST or GET but also differentiate resources
  - Level 2 – use the appropriate HTTP verbs
  - Level 3 – within responses, provide URLs to be used for subsequent requests
- For more information, visit
  - http://martinfowler.com/articles/richardsonMaturityModel.html

Level 3 and HATEOAS

- With Level 3 services, the response from a request includes the URLs that can be used for subsequent requests
  - HATEOAS – Hypertext As The Engine Of Application State
- If you retrieved a list of restaurants, each restaurant would include the URL to get the menu
  - This means that the service provider could change those restaurant URLs if it wanted to

neueda

Self Describing APIs

- When you get to Level 3 based services, the APIs become self describing
- Once you know the entry point, the rest of the API is apparent within the responses

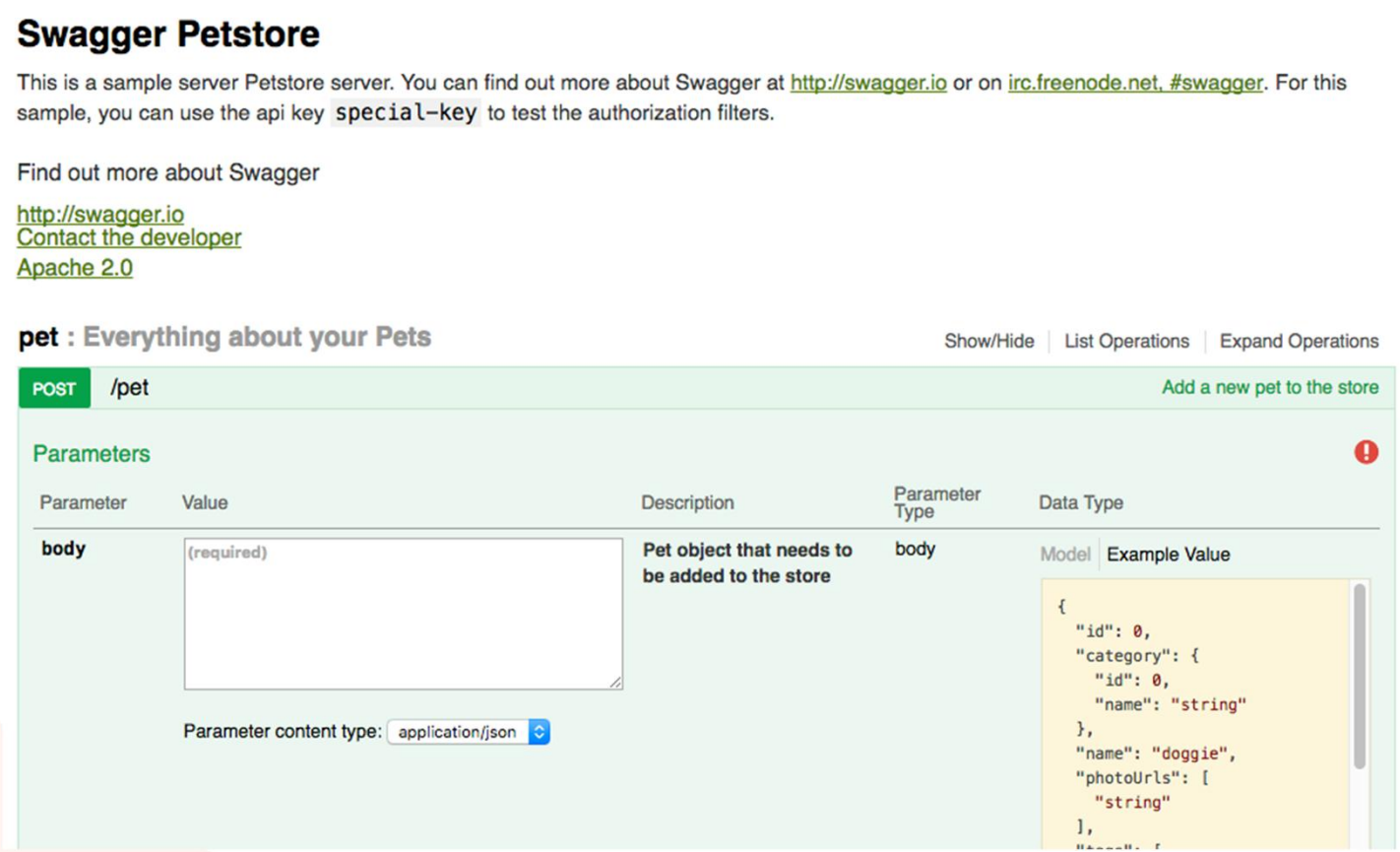
neueda

Documenting REST APIs

- REST APIs can be documented using **OpenAPI** (formerly known as **Swagger**)
- OpenAPI can be used to
  - Import an existing API into a platform to create a new version
  - Provide an automatically generated Web interface describing and allowing you to try out your API
  - Autogenerate client code in multiple languages, and the documentation allows people to understand the JSON structures and API calls
    - https://www.openapis.org/

neueda

OpenAPI Example Web Interface Example



neueda

Summary

- Introducing REST
- The principles of REST
- Implementing a REST Web service

neueda


neueda
Python Rest Services

Contents

- Implementing a Rest service in Python
- Flask and FlaskRestful
- Flask RESTful Approaches
- Using a Resource Class
- Example CRUD REST Service using MySQL

neueda

Implementing a Rest Service in Python

- Install Flask and Flask restful
- Import Flask Web and Rest objects
- Add sample data
- Add a class that inherits from Resource
- Handle GET Requests
- Handle POST Requests
- Handle PUT Requests
- Handle DELETE Requests
- URL Patterns and running the service

neueda

Install Flask and Flask_restful

- There are many Python packages available, to help you implement a Rest service…
  - We will be using Flask
  - Install the following Python packages:

```
pip install flask
pip install flask_restful
```


neueda

Flask RESTful Approaches

- When creating a REST API you can either
  - Create functions with app.route patterns as we saw before
  - Create a class extending Resource that has methods for get / post / put / delete

FlaskRestful/CompactDiscRestful.

```
@app.route('/compact_discs', methods=['GET'])
def get_compact_discs():
    cursor = db.cursor()
    cursor.execute("SELECT * FROM compact_discs")
    compact_discs = cursor.fetchall()
    cursor.close()
    return jsonify(compact_discs)
```

```
class CompactDiscsResource(Resource):
    def get(self):
        cursor = db.cursor()
        cursor.execute("SELECT * FROM compact_discs")
        compact_discs = cursor.fetchall()
        cursor.close()
        return jsonify(compact_discs)
```

FlaskRestful/CompactDiscRestfulUsingResourc
e.py

neueda

Using a Resource class

- The following example will use the Resource class approach
- These imports are needed to implement a Rest Api using flask

```
from flask import Flask
from flask_restful import Api, Resource,
reqparse
import mysql.connector
```

- We need a Flask and Api object and can use these statements

```
app = Flask(__name__)
api = Api(app)
```

The Database Connection

- The database connection is managed in the same way as we saw in our previous example

```
db = mysql.connector.connect(
    host="localhost",
    user="root",
    password="c0nygre1",
    database="conygre"
)
```


neueda

Add a Class that Inherits from Resource

- We need to add a class that inherits from Resource
- Note the plural use of CompactDiscsResource
  - The significance of this will become apparent

```
class CompactDiscsResource(Resource):
```


neueda

Handling a GET Request for All Items

- This function handles GET requests and returns all the CompactDisc data
- The function name informs the Flask engine that this is for Get requests

```
class CompactDiscsResource(Resource):
    def get(self):
        cursor = db.cursor()
        cursor.execute("SELECT * FROM compact_discs")
        compact_discs = cursor.fetchall()
        cursor.close()
        return jsonify(compact_discs)
```

- You will see later how the URL is inferred
- You will also see later how we handle a single API calls for a single item

Using POST for Create Requests

- This function handles POST requests
- The RequestParser library function is being used to parse the arguments from the incoming JSON data

```
 def post(self):
        parser = reqparse.RequestParser()
        parser.add_argument('title', required=True)
        parser.add_argument('artist', required=True)
        parser.add_argument('track_count', required=True, type=int)
        parser.add_argument('price', required=True, type=float)
        args = parser.parse_args()
        title = args['title']
        artist = args['artist']
        track_count = args['track_count']
        price = args['price']
        cursor = db.cursor()
        cursor.execute("INSERT INTO compact_discs (title, artist, track_count, price) VALUES (%s, %s,
%s, %s)", (title, artist, track_count, price))
        db.commit()
        cursor.close()
        return {'message': 'Compact disc created successfully'}, 201
```

Mapping to a URL Pattern

- The URL can then be mapped by mapping the Resource class to a URL pattern

```
api.add_resource(CompactDiscsResource, '/compact_discs')
```

- The URL path to get all compact discs is
  - GET /compact_discs
- The URL path to add a compact disc is
  - POST /compact_discs

What about Individual Resources?

- For individual resources, another Resource class is added to the application
  - Note the singular use of the word CompactDiscResource
  - This is for individual resources instead of multiple resources

```
class CompactDiscResource(Resource):
```


neueda

Implementing Get by ID

- To retrieve an item by ID, the resource method can be implemented as follows

```
def get(self, cd_id):
        cursor = db.cursor()
        cursor.execute("SELECT * FROM compact_discs WHERE id = %s", (cd_id,))
        compact_disc = cursor.fetchone()
        cursor.close()
        if compact_disc:
            return jsonify(compact_disc)
        else:
            return {'error': 'Compact disc not found'}, 404
```

- The cd_id parameter will automatically be set from the URL path which is mapped by another add_resource call

```
api.add_resource(CompactDiscResource, '/compact_discs/<int:cd_id>')
```

Handling Updates

- Updates are almost identical to inserts, except that this time there is an ID for the one to update

```
def put(self, cd_id):
        parser = reqparse.RequestParser()
        parser.add_argument('title', required=True)
        ..
        args = parser.parse_args()

        title = args['title']
        artist = args['artist']
        ..

        cursor = db.cursor()
        cursor.execute("UPDATE compact_discs SET title = %s, artist = %s, track_count = %s, price = %s
WHERE id = %s",  (title, artist, track_count, price, cd_id))
        db.commit()
        cursor.close()
        return {'message': 'Compact disc updated successfully'}, 200
```


neueda

Handling Deletes

- Deletes also use the cd_id parameter

```
def delete(self, cd_id):
        cursor = db.cursor()
        cursor.execute("DELETE FROM compact_discs WHERE id = %s", (cd_id,))
        db.commit()
        cursor.close()

        return {'message': 'Compact disc deleted successfully'}, 200
```

Summary

- Implementing a Rest service in Python
- Flask and FlaskRestful
- Flask RESTful Approaches
- Using a Resource Class
- Example CRUD REST Service using MySQL

neueda