# Web Database Programming

The purpose of this book is to form the basis of a university module, for students who have already been introduced to programming, relational databases and the Web.

The combination of web and database arises is real life in a great many scenarios. First, web applications and sites usually have a backing database of resources and/or information from clients. Second, business applications often have a web front end. In both of these situations the owner of the database is the same as the owner of the web application. In both of these cases, the database and the application are commonly on the same system and this simplifies issues of authentication and authorisation; also, the interaction between application and database usually is by means of the DBMS API (such as JDBC, JPA, ADO.NET, or ODBC).

This course deals with the more interesting situation where the web application and database have different owners and/or are on different platforms. Such multi-tier systems have been studied in the past, for example, with enterprise Java. Distributed systems generally have focused in recent years on Service Oriented Architecture, so that HTTP has been widely used for linking components of such applications. Strangely, however, despite the popularity of RESTful architectures, HTTP has not been widely used for communication *between* the application and the database, and very few DBMS vendors provide such an interface. This course takes this particular use case in order to develop students’ skills in a range of related technologies.

The course has the following objectives:

* To give you practical experience of using REST APIs
* To give you practical experience of using AngularJS
* To develop explicit authentication and authorisation mechanisms
* To explore transaction and caching issues in such composite systems
* To develop strict techniques for loosely coupled distributed systems

The course is essentially platform-neutral. Instructions will be given for Windows 10 and Linux (CentOS 7), both for single machines and virtual servers. However, if the database is on Linux mono will need to be installed. These options will make for some repetition in the practical instructions. In most cases implementation will start with the database and application on the same machine, but should ideally move on to separate systems as described above, to motivate the use of HTTP. A convenient isolated way to model this is to use two virtual machines on a common virtual network, and two CentOS VM images are supplied (using vhdx format) to facilitate this.

The only actual programming will be in JavaScript, as modifications to a client single-page application, so the only AngularJS package used will be $http.

The only database used will be MySQL, on the server. There will be some exploration of the use of a variety of data types with JSON.

The only web server used will be a REST interface utility written in C# that has been kept deliberately simple. It will be fully explained as part of the course.

## Chapter 0: Setting up

The simplest way of getting things working is on a single Windows or Linux machine, and you should create a folder for your application, e.g. c:\temp\wdbp or /home/wdbp . This will be referred to as your working folder in these instructions and obviously you should use / or \ as appropriate for your operating system when accessing subfolders. The machine with the database on it is called the server machine in these notes: obviously if you only have the one machine the server machine and application machines are the same.

You will use a text editor such as Notepad++, and a modern web browser of your choice. The course has been tested with Edge, IE11, Firefox and Chrome.

If the server is a separate or virtual machine, these notes will assume that you have no difficulty copying files from one machine to another, and so no detailed notes for such steps will be given. For example, (a) remote computer connections can share a file system on the host computer, (b) files can be extracted from virtual floppy disks using 7-zip, (c) if a virtual machine is shut down, its virtual hard disk can be removed and mounted in the host operating system, and (d) an ISO creation utility can be used to provide files for a VM.

The remaining content of this Chapter gives detailed instructions to

* install MySQL on your workstation or the server, and for creating a simple database containing a sample table
* install and start up the RestIf software on your workstation or the server: you can use Python 3.4 or .NET
* Optional: install and start up the RESTClient tool to check everything is working
* Optional: create a virtual machine using CentOS (for either the server or the application)

If you have more than one machine, it may be useful to have the RestIf sources available for consultation on the machine where you are writing your JavaScript. We assume that you have a desktop GUI (Windows or Linux GNOME or whatever) on the application machine, including a text editor and a browser. You do not need a desktop GUI on the server machine.

### Obtaining MySQL

You may already have MySQL: we need to have a mysql-server on the server machine. If not, browse to mysql.com and select Downloads and Community (GPL) Downloads (<https://dev.mysql.com/downloads/mysql/>) and choose the instructions for your operating system. On CentOS it is replaced by mariadb, and the instructions are given in the CentOS VM section below.

On Windows, MySQL is usually installed as a service, so you do not need to start it up manually. You do need to login on the system to create databases and tables, so we assume you know what MySQL’s root password is. (Confusingly for Linux users, MySQL’s root is not the same as Linux’s.) For Linux, see Appendix A.

## Installing Python 3

Currently, the RESTIF server is available for Python 3.5.3 or .NET . .NET is always available on Windows (we use version 4.6), and you can get Python for Windows from python.org. The limitation to Python 3.5.3 comes from the version of the MySQL connector available at the time of writing. For Linux, see Appendix A (you will need to use Python 3.4 and PyMySql).

## Installing RESTIF

Extract the software accompanying this document to somewhere convenient such as c:/restif. For .NET, the RestifD subfolder contains a .exe file you should run as administrator to start up the RESTIF server.

For Python, you will find a Restif.py file in the top restif folder. Run this as administrator. Full details of the software are given in Appendix B.

Administrator privileges are needed only in order to start up a web service on port 8088/

Whichever version you use, leave it running. It normally shows a blank DOS window.

## RESTIF’s REST API

Every REST API involves designing what the URL pattern is and what verbs are allowed.

Let us keep things very simple. The following lines show the path(i.e. the portion of the URL that immediately follows the hostname:port prefix). We assume the server only interacts with MySQL on the given host, and that the username and password will come from an Authentication header in the request. Apart from the first two lines in this table, db represents an existing database, and tb an existing table in that database.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Verb | Path | Request data | Returns | Notes |
| GET | / |  | An array of strings | The result of show databases; |
| GET | /db |  | An array of strings | The result of connect db; show tables; |
| GET | /db/tb |  | An array of rows | The result of select \* from tb; |
| GET | /db/tb/s |  | An array of rows | The result of select \* from tb where s;  e.g. s might be id=5 and rvv=17 |
| POST | /db | a list of SQL statements |  | The statements are executed in a transaction |
| POST | /db/tb | a single row |  | The row will be added to the table |
| PUT | /db/tb/s | a partial row |  | The row(s) matching s will be modified using the given Json object |
| DELETE | /db/tb |  |  | Delete all rows in tb |
| DELETE | /db/tb/s |  |  | Delete all rows in tb matching condition s |

Note that we always select rows in the table using the URL (using the selection condition s). If we sometimes used the Json data to select rows the API would still be easy to implement, but harder to explain and use.

## Chapter 1

**A simple JSON GET from a database**

Chapter 2: POSTing new Json data to create a new row of the table.  Authentication and authorisation.

Chapter 3: Use a WebRequest to PUT Json data into a row, and DELETE. Roles.

Chapter 4: Defining RESTViews with SQL: GET data from a RESTView.

Chapter 5: Updatable views and joins. Simple POST, PUT and DELETE for RESTViews.

Chapter 7: Using JSON inside SQL 2016: POSTing a CompoundStatement to the database. Atomic execution.

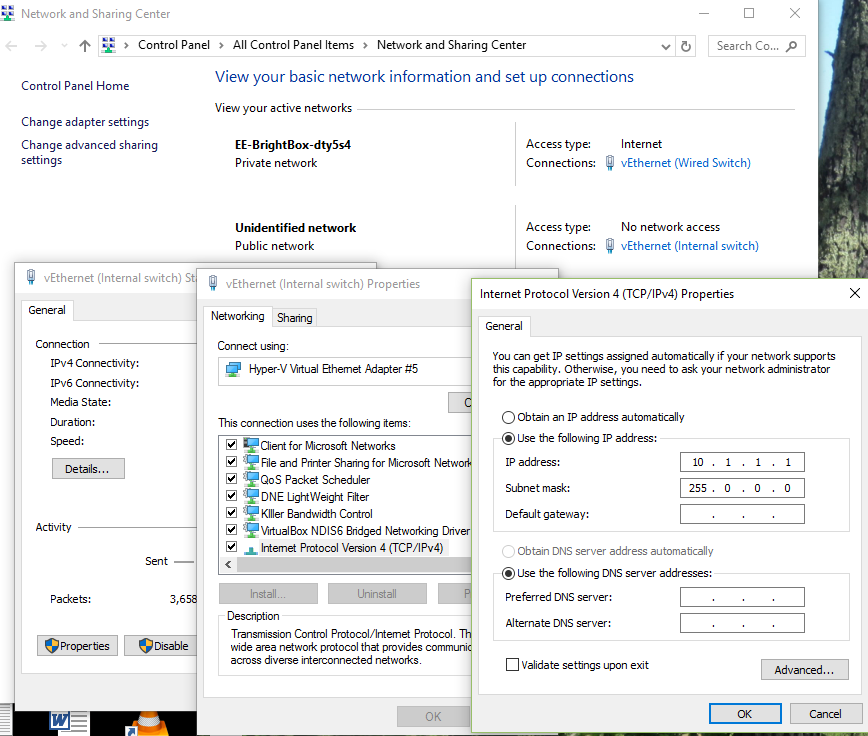
Chapter 8: Using ETags to separate retrieval from changes. Transactions as retrieval followed by a change.

Chapter 9: Distributed systems (multiple databases, multiple hosts).

# Appendix A

All the details for Linux

## Creating your own CentOS VM

For the course you will have two machines provided on a Hyper-V server, as the labs do not allow virtual machines to connect to the external network. These instructions will allow you to create your own VMs. You will need two machines: wdbps for the server (base), and wdbp app, or just one: wdbp for both server and app.

The server machine will be the one with MySQL and Mono: it does not need a desktop environment. Your web application will be on app: if you are using JavaScript then the desktop environment will give you the browser facilities you need for JavaScript programming.

I know many Linux aficionados do not like mono. I also would prefer to use Python: but it would need to be Python 3, and unfortunately there seems to be no good MySQL connector on Linux for Python 3.

### The virtual local network

If you are using Hyper-V to host your VMs you will need to set up a virtual network so you can communicate between your VMs and your desktop host. On the Hyper-V manager, create a new internal virtual switch. In the host OS’s Network and Sharing Centre, click the new virtual switch and set its IPv4 address to 10.1.1.1 . Click the Advanced.. button, clear the Metric check box and give the value 2. We will finish configuring this network once we have created the VM(s).

Also create an external virtual switch, using your usual external network adapter. In the Network and Sharing Centre, click the new virtual switch and in the IPv4 properties, click the Advanced.. button, clear the Metric check box and give the value 1.

From CentOS.org, download a minimal installation disk: CentOS-7-x86\_64-Minimal-1611.iso (680MB). This will mean your VM needs the external network during installation.

### First steps for both VMs:

With Hyper-V or your favourite virtualisation environment create a machine called wdbp (and/or wdbps), 1GB RAM and 8GB of virtual disk should be sufficient. Use the CentOS installation disk image to set up a minimal system with the following settings:

Language and region: English UK

Root password: wdbp (this is short so you click Done twice)

Full name: wdbp owner

Username: wowner (is administrator)

Password for wowner: owner (this is also short)

Boot into the new system, login and start an administrator shell with sudo bash. Next we configure the virtual network.

yum update

yum install nano vim

yum install net-tools

On wdbps you want

ifconfig eth0 10.1.1.4 netmask 255.0.0.0

On wdbps you want

ifconfig eth0 10.1.1.5 netmask 255.0.0.0

Then on both machines

nano /etc/hosts

add three lines as follows (\t is a tab):

10.1.1.1\thost

10.1.1.4\tbase

10.1.1.5\tapp

and save the file.

### Install MySQL/MariaDB

MySQL is needed on the server machine (base).

Now install mysql (mariadb) and server.

yum install mariadb-server mariadb

/sbin/service mariadb start

firewall-cmd –add-port=3306/tcp

firewall-cmd –permanent –add-port=3306/tcp

## Open network ports

firewall-cmd –add-port=8088/tcp

firewall-cmd –permanent –add-port=8088/tcp

firewall-cmd –add-port=8078/tcp

firewall-cmd –permanent –add-port=8078/tcp

### Install PyMySQL

If you need access to the external network at this stage (replace the IP addresses here to suit your host system), configure eth0 using the following commands:

# ifconfig eth0 192.168.1.109 netmask 255.255.0.0

# route add default gw 192.168.1.1

# nano /etc/resolv.conf

and add a line for your DNS

nameserver 192.168.1.1

save the file

# yum install wget

# wget http://dl.fedoraproject.org/pub/epel/7/x86\_64/e/epel-release-7.10.noarch.rpm

# rpm -ivh epel-release-7-10.noarch.rpm

# yum install python34-setuptools

# easy\_install-3.4 pip

***# pip3 install*** ***PyMySQL***

Now remember that you need to say python3 to run a python3 program: e.g.

$ **python3**

..

>>> **import sys**

>>> **sys.version\_info**

sys.version\_info(major=3, minor=4, micro=5, releaselevel='final', serial=0)

>>> **quit()**

### Optional: Install mono

Mono is needed on the server machine (base).

yum install yum-utils

rpm --import "http://keyserver.ubuntu.com/pks/lookup?op=get&search=0x3FA7E0328081BFF6A14DA29AA6A19B38D3D831EF"

yum-config-manager --add-repo <http://download.mono-project.com/repo/centos7/>

yum install mono-devel

yum install mono-complete

### Install the RESTIF server

The RESTIF server is placed on the server machine (base). Insert the RESTIF.iso disk in the virtual DVD drive.

mount /dev/cdrom /mnt -t cd9660

cd ~wowner

cp -r /mnt/restif .

mono restif/restifd.exe

The restifd.exe does not normally produce output. You can check if it is running from a browser on the host machine. Try http://10.1.1.1:8088/

### Install a desktop

You will need a desktop of your choice on the wdbp (app) machine. For example:

yum -y groups install "GNOME Desktop"

startx

# Appendix B

# A RESTIF Server in Python: step by step

The purpose of this article is to develop a generic Web server in Python 3.5 that can provide a simple REST interface to any MySQL[[1]](#footnote-1) database without configuration, supporting a variety of formats including Json. The only assumption is that RESTIF runs on the same machine as the database: any authentication credentials required for database access will be supplied in the HTTP requests.

Be aware that Python always run a single operating-system thread: so requests to the server are handled sequentially. The .NET solution in Appendix C offers better server parallelism.

We could use HTTP just as transport, for example, to POST a set of SQL commands to a database to be run as a well-configured transaction. Since there is no totally safe way of committing a transaction that involves more than one database (this is the famous two-army situation), we will limit our discussion to such transactions.

The real purpose of the RESTIF server is to support REST-style HTTP requests (GET, POST, PUT and DELETE) requests from a web application, not from a browser. If the web application is implemented using script Web pages, it is simplest to supply data in JSON format, but other formats could well be useful.

Since HTTP runs on top of TCP, there is a scalability issue since the TCP connection renegotiates packet length every time the message length changes and this renegotiation typically takes 1 second. This is not normally a problem in web applications, and for simplicity in this worksheet we will not try to make all messages have the same length. Also, we will suppose that payload of GET responses is of a reasonable size (this can always be assured by appropriate design of the database and application).

We will motivate and achieve this work through a sample Payroll database in order to demonstrate the concepts. However, the RESTIF server will be generic, i.e. will not have any of our Payroll details hardwired. This means the RESTIF software developed in this worksheet can be used for any MySQL database without change.

## A generic REST API

Every REST API involves designing what the URL pattern is and what verbs are allowed.

Let us keep things very simple. The following lines show the path(i.e. the portion of the URL that immediately follows the hostname:port prefix). We assume the server only interacts with MySQL on the given host, and that the username and password will come from an Authentication header in the request. Apart from the first two lines in this table, db represents an existing database, and tb an existing table in that database.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Verb | Path | Request data | Returns | Notes |
| GET | / |  | An array of strings | The result of show databases; |
| GET | /db |  | An array of strings | The result of connect db; show tables; |
| GET | /db/tb |  | An array of rows | The result of select \* from tb; |
| GET | /db/tb/s |  | An array of rows | The result of select \* from tb where s;  e.g. s might be id=5 and rvv=17 |
| POST | /db | a list of SQL statements |  | The statements are executed in a transaction |
| POST | /db/tb | a single row |  | The row will be added to the table |
| PUT | /db/tb/s | a partial row |  | The row(s) matching s will be modified using the given Json object |
| DELETE | /db/tb |  |  | Delete all rows in tb |
| DELETE | /db/tb/s |  |  | Delete all rows in tb matching condition s |

Note that we always select rows in the table using the URL (using the selection condition s). If we sometimes used the Json data to select rows the API would still be easy to implement, but harder to explain and use.

## An Employee management system

The tables we will use for the demonstration are[[2]](#footnote-2):

create table employee (id integer primary key, name varchar(20), notes varchar(1000));

create table posting (empid integer, start date, grade varchar(4), manager integer,foreign key (empid) references employee(id),foreign key (manager) references employee(id));

create table holiday (empid integer, start date, end date, agreed date,foreign key (empid) references employee(id));

And let’s give them some initial data:

insert into EMPLOYEE values(1562,'John Black','Sales');

insert into EMPLOYEE values(1567,'Mary White','Finance');

insert into EMPLOYEE values(1569,'Paul Green','HR');

insert into POSTING values(1562,'2012-02-01','A1',1569);

insert into POSTING values(1562,'2012-04-01','A2',null);

insert into POSTING values(1567,'2012-02-01','B1',1569);

insert into POSTING values(1569,'2012-02-01','A2',null);

insert into HOLIDAY values(1567,'2012-03-02','2012-03-07','2012-02-05');

insert into HOLIDAY values(1569,'2012-04-04','2012-04-18',null);

EMPLOYEE

|  |  |  |
| --- | --- | --- |
| 1562 | John Black | Sales |
| 1567 | Mary White | Finance |
| 1569 | Paul Green | HR |

POSTING

|  |  |  |  |
| --- | --- | --- | --- |
| 1562 | 1/2/2012 | A1 | 1569 |
| 1562 | 1/4/2012 | A2 |  |
| 1567 | 1/2/2012 | B1 | 1569 |
| 1569 | 1/2/2012 | A2 |  |

HOLIDAY

|  |  |  |  |
| --- | --- | --- | --- |
| 1567 | 2/3/2012 | 7/3/2012 | 5/2/2012 |
| 1569 | 4/4/2012 | 18/4/2012 | 3/4/2012 |

We will grant all privileges on this database to a local user called admin (admin does not have a login account on the server).

create user 'admin'@'localhost' identified by 'apwd';

grant all on Payroll.\* to 'admin'@'localhost' identified by 'apwd';

## Getting Python on Windows

If you are not using a Windows workstation as a server, you can skip to the section entitled “Installing RESTIF.py on a Linux server”. The application machine does not need MySQL or Python.

From mysql.com download and install the MySQL Community Server. I got version 5.7.

If our example had issues of scalability I would be configuring a compiler for Python to machine code, but for this example a single server thread will be fine, and there is no need to worry about multithreading or compilation.

If you haven’t already done so, download Python from Python.org, which is available for a great many platforms. Python 3.5.3 was the latest released version at the time of writing for which a MySQL Connector is available for Windows, and this has been used for this tutorial*.*

Once you have installed Python and MySQL, also install the correct MySQL Connector/Python from mysql.com. In these notes I have used version 8.0.4 of the connector.

## Create a web server

This Appendix develops RESTIF.py in stages. If you want to skip this, you can simply skip to the section called “Installing RESTIF.py on Windows”.

I use Visual Studio, whose Python tools I find helpful, but anything will do: many authors recommend Notepad++. In favour of Visual Studio, the debugger is great: if you can, use a system with a Python debugger that is as good. So the instructions in this tutorial will just use phrases like “create a new file”, “add a method to” etc, rather than being Visual Studio-specific. Be careful with indentation though: always use tabs rather than spaces at the start of lines. In Visual studio you want to create a new Python application called Restif.

The name of the server in these notes is Restif.py but you can use any names you want. As we go, note that the python code never contains any mention of Payroll or the above tables.

1. Create file Restif.py if necessary, and replace any contents with:

from http.server import \*

class myHandler(BaseHTTPRequestHandler):

def do\_GET(self):

self.send\_response(200)

self.send\_header('Content-type','text/html')

self.end\_headers()

self.wfile.write(bytes("Hello World !",'utf-8'))

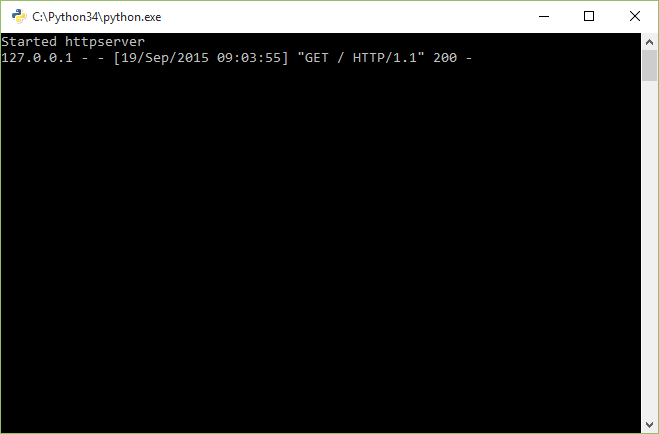
server = HTTPServer(('', 8088), myHandler)

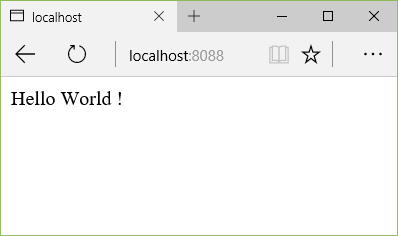
print ('Started httpserver')

server.serve\_forever()

This is completely standard Python3.4 code that you will find in many places on the Web.

Check it works: When you start it up it should say “Started httpserver”. Open a browser and give the URL as http://localhost:8088/



  
The browser should show “Hello World” and the command window show a log entry for GET. You can try other URLs starting with <http://localhost:8088>. You will see the URLs in the log entries but obviously the browser display will be the same.

If this doesn’t work, the most likely causes are (a) you have incorrect indentation in your Python code (in step 4). Ensure that in the code you use tabs and no spaces (or spaces and no tabs) at the beginning of lines: and that the code is correctly indented as shown; (b) on Windows you are not running the program as Administrator (start Visual Studio using Run As Administrator) (c) your Python installation is not 3.4 (d) there is some other process listening on port 8088. Fix these problems before continuing.

1. Let’s make this server tell us about databases we can use, and add a Send method:

from http.server import \*

import mysql.connector

import sys

class RestifHandler(BaseHTTPRequestHandler):

def Send(self,status,mess):

self.send\_response(status)

if mess!=None:

bs = bytes(mess,'utf-8')

self.send\_header('Content-Length',len(bs))

self.end\_headers()

self.wfile.write(bs)

return

def do\_GET(self):

try:

conn = mysql.connector.connect(user='admin',password='apwd')

cursor = conn.cursor(raw=True)

cursor.execute('show databases')

mess = str()

for db in cursor.fetchall():

mess += ' '+db[0].decode('utf-8')

self.Send(200,mess)

except Exception as e:

self.Send(403,e.msg)

return

def log\_request(code,size):

return

try:

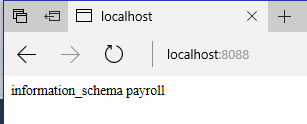
server = HTTPServer(('',8088),RestifHandler)

server.serve\_forever()

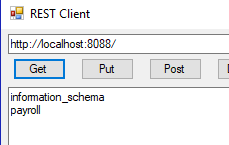
except KeyboardInterrupt:

print ('Exiting')

server.socket.close()

The log\_request method is an override: to suppress the details of http requests in the application window. You can omit it if you prefer to see them. We return to this aspect later.

1. Check this works. Run the program as administrator. In your browser try http://localhost:8088

The browser should display a list of databases on the server, including payroll.

In later steps we will use RESTClients a lot, as browsers won’t help us with PUT and DELETE. Locate RESTClient in the folder containing this document, and in the bin/Debug folder double-click the executable. Give the above URL in the address bar at the top, and click Get. You will see that this client respects the newline characters in the list returned.

Stop the server with ^C or by closing the python.exe window.

## The list of tables in the database

1. The RestifHandler knows the url supplied: it is called path. Let’s modify do\_GET to take different actions depending on the number of components in the path.

def do\_GET(self):

try:

segments = self.path.split('/')

db = None

if segments[1]!='':

db = segments[1]

conn = mysql.connector.connect(user='admin',password='apwd',database=db)

 cursor = conn.cursor(raw=True)

if db is None:

cursor.execute('show databases')

else:

cursor.execute('show tables')

mess = str()

for rw in cursor.fetchall():

mess += rw[0].decode('utf-8') + '\n'

self.Send(200,mess)

keeping the lines except… onwards. Now if we use the url <http://localhost:8088/Payroll> we get a list of table names:

## Querying the database

1. We look to see if we have a tablename in the path, and the rest of the URL can be a where condition.

def do\_GET(self):

try:

segments = self.path.split('/')

db = None

tb = None

wh = None

if segments[1]!='':

db = segments[1]

if len(segments)>2:

tb = segments[2]

if len(segments)>3:

wh = segments[3]

conn = mysql.connector.connect(user='root',password='pwd',database=db)

cursor = conn.cursor(raw=True)

if db==None:

cursor.execute('show databases')

elif tb==None:

cursor.execute('show tables')

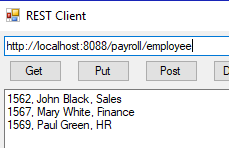
elif wh==None:

cursor.execute('Select \* from '+tb)

else:

cursor.execute('Select \* from '+tb+' where '+wh)

mess = str()

 for rw in cursor.fetchall():

n = 0

if tb!=None:

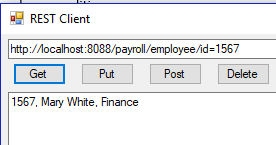
n = len(rw)-1

for i in range(n):

mess += rw[i].decode('utf-8') + ', '

mess += rw[n].decode('utf-8') + '\n'

self.Send(200,mess)

Now we can also have <http://localhost:8088/Payroll/employee> and <http://localhost:8088/Payroll/employee/id=1567>

1. Note the lack of headings here. So that we can improve things, let’s restructure this code a bit:

def do\_GET(self):

try:

segments = self.path.split('/')

db = None

tb = None

wh = None

if segments[1]!='':

db = segments[1]

if len(segments)>2:

tb = segments[2]

if len(segments)>3:

wh = segments[3]

conn = mysql.connector.connect(user='admin',password='apwd',database=db)

if conn is None:

self.Send(400,'Cannot connect')

return

if tb is None:

mess = self.Get1(conn,db)

else:

mess = self.Get2(conn,tb,wh)

self.Send(200,mess)

except Exception as e:

self.Send(403,e.msg)

if conn!=None:

conn.close()

return

def Get1(self,conn,db):

cursor = conn.cursor(raw=True)

if db==None:

cursor.execute('show databases')

else:

cursor.execute('show tables')

mess = str()

for rw in cursor.fetchall():

mess += rw[0].decode('utf-8') + '\n'

cursor.close()

return mess

def Get2(self,conn,tb,wh):

cursor = conn.cursor()

if wh==None:

cursor.execute('Select \* from '+tb)

else:

cursor.execute('Select \* from '+tb+' where '+wh)

rws = cursor.fetchall()

mess = ''

for rw in rws:

n = len(rw)

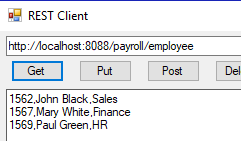
m= ''

for i in range(n):

mess += m+str(rw[i])

m = ','

mess += '\n'

 cursor.close()

return mess

Note we have dropped raw and decode in Get2. The separator m (middle) is initially empty and then is a comma.

Check this works the same way as before.

1. We really should put single quotes around fields that are strings, using the type information in cursor.\_description:

def Get2(self,conn,tb,wh):

cursor = conn.cursor()

if wh==None:

cursor.execute('Select \* from '+tb)

else:

cursor.execute('Select \* from '+tb+' where '+wh)

rws = cursor.fetchall()

desc = cursor.\_description

mess = ''

for rw in rws:

n = len(rw)

m= ''

for i in range(n):

mess += m+self.Value(desc[i][1],str(rw[i]))

m = ','

mess += '\n'

cursor.close()

return mess

def Value(self,typ,val):

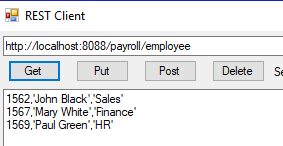
q = ""

if typ in {253,254,252,10,7,11}: # char, varchar, text, date, timestamp, time

q = "'" # single quote because Sql option uses text/plain too

return q+val+q

The list of numbers in this method was the result of a small experiment on different database columns.



1. Now let’s modify Get2 to get hold of the column names and use Json.

def Get2(self,conn,tb,wh):

cursor = conn.cursor()

if wh==None:

cursor.execute('Select \* from '+tb)

else:

cursor.execute('Select \* from '+tb+' where '+wh)

rws = cursor.fetchall()

desc = cursor.\_description

mess = '['

rsep = '' # row separator initially empty

for rw in rws:

mess += rsep

rsep = ',\n' # for next time

mess += '{'

fsep = '' # field separator initially empty

for i in range(len(desc)):

mess += fsep

fsep = ', '

mess += '"' + desc[i][0] +'": ' + self.Value(desc[i][1],str(rw[i]))

mess += '}'

cursor.close()

return mess + ']'

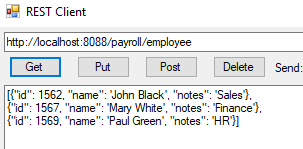
def Value(self,typ,val):

q = ""

if typ in {253,254,252,10,7,11}: # char, varchar, text, date, timestamp, time

q = "'" # single quote because Sql option uses text/plain too

return q+val+q



1. Using the Accept header to choose the format characters would be better:

def Get2(self,conn,tb,wh):

cursor = conn.cursor()

if (wh==None):

cursor.execute('Select \* from '+tb)

else:

cursor.execute('Select \* from '+tb+' where '+wh)

rows = cursor.fetchall()

desc = cursor.\_description

acc = self.headers['Accept']

fmts = { 'text/plain':['','\n','','',',',''], # array start mid end; obj start mid end

'application/json':['[',',\n',']','{',', ','}']}

fmt = fmts[acc]

mess = fmt[0] # array start

rsep = '' # row separator initially empty

for rw in rows:

mess += rsep

rsep = fmt[1] # row sep is array mid

mess += fmt[3] # obj start

fsep = ''

for i in range(len(desc)):

mess += fsep

fsep = fmt[4] # obj mid

mess += self.Field(acc,desc,rw,i)

mess += fmt[5] # obj end

cursor.close()

return mess + fmt[2] # array end

def Field(self,acc,desc,data,i):

v = self.Value(desc[i][1],str(data[i]))

if acc=='text/plain':

return v

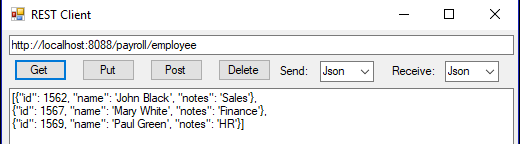
return '"' + desc[i][0] +'": ' + v

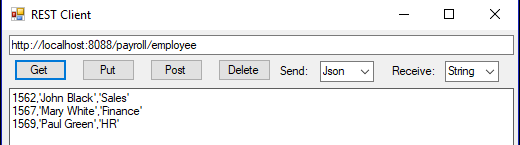
def Value(self,typ,val):

q = ""

if typ in {253,254,252,10,7,11}: # char, varchar, text, date, timestamp, time

q = "'" # single quote because Sql option uses text/plain too

return q+val+q 



Of course, some of the coding ideas above can be extended to support other formats if desired.

fmt is an array of delimiters for array start, object start, field separator, object end, array end.

## Getting started with POST

1. Let’s add a method to get hold of posted data:

def GetData(self):

if not self.headers.\_\_contains\_\_('Content-Length'):

return None

n = int(self.headers['Content-Length'])

return str(self.rfile.read(n),'utf-8')

This will return a string. In the next step we will use the Content-Type header to deal with Json.

The first POST method in our REST API receives a list of SQL statements to be executed in the database. You can skip to the next step if this possibility is not needed.

def do\_POST(self):

stmts = self.GetData()

if stmts is None:

self.Send(400,'Nothing to do')

return

conn = mysql.connector.connect(user='admin',password='apwd',database=db)

if conn is None:

self.Send(400,'Cannot connect')

return

try:

conn.start\_transaction(consistent\_snapshot=True,

isolation\_level='SERIALIZABLE')

conn.cmd\_query\_iter(stmts)

conn.commit()

self.Send(200,'OK')

except Exception as e:

self.Send(403,e.msg)

conn.close()

return

1. As a first step to implementing the other sorts of POST, let us analyse the path:

def do\_POST(self):

segments = self.path.split('/')

db = None

tb = None

if segments[1]!='':

db = segments[1]

if len(segments)>2:

tb = segments[2]

data = self.GetData()

if data is None:

self.Send(400,'Nothing to do')

return

conn = mysql.connector.connect(user='admin',password='apwd',database=db)

if conn is None:

self.Send(400,'Cannot connect')

return

if tb is None:

self.Post1(conn,data)

else:

self.Post2(conn,tb,data)

conn.close()

self.Send(200,'OK')

return

def Post1(self,conn,data):

try:

conn.start\_transaction(consistent\_snapshot=True,

isolation\_level='SERIALIZABLE')

k = 1

for result in conn.cmd\_query\_iter(data):

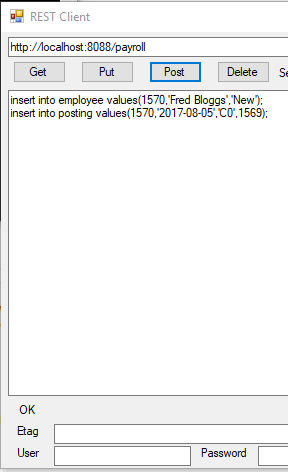
k = k+1

conn.commit()

self.Send(200,'OK')

except Exception as e:

self.Send(403,'line '+str(k)+': '+e.msg)

 return

def Post2(self,tb,data):

return

Try the POST button with url <http://localhost:8188/payroll> and posted data

insert into employee values(1570,'Fred Bloggs','New');

insert into posting values(1570,'2017-08-05','C0',1569);

1. For the Post2 method we need to examine the data. Python is able to parse Json using ast.literal\_eval().(There is an opportunity here to support other Content-Types.)

def Post2(self,conn,tb,data):

try:

data = ast.literal\_eval(data)

cmd = 'insert into '+tb+' ('

vals = ') values ('

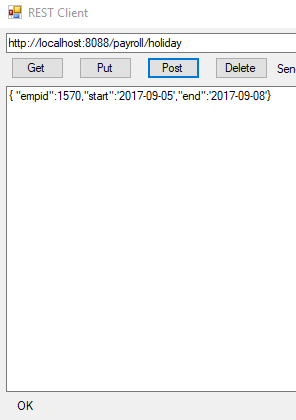
cm = ''

for k in data.keys():

cmd += cm+k

v = data[k]

q = ''

 if type(v) is str:

q = "'"

vals += cm+q+str(v)+q

cm = ','

conn.cmd\_query(cmd+vals+')')

conn.commit()

self.Send(200,'OK')

except Exception as e:

self.Send(403,e.msg)

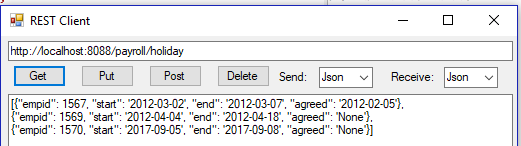
return

This code uses the keys and values in the posted data to construct an SQL insert statement, and commits it to the database. Note that even though we have only one command, we still need to call commit(0 explicitly.

1. Try the POST button with url <http://localhost:8088/payroll/holiday> and posted data

{ "empid":1570,"start":'2017-09-05',"end":'2017-09-08'}

Check everything has worked using GET:



1. The version of Restif.py provided online has some changes to the above code to avoid using += for strings. It is much more efficient to construct an array of strings and use join. You can see en example in the next step.

## PUT and DELETE

1. These are now very straightforward:

def do\_PUT(self):

segments = self.path.split('/')

db = None

tb = None

wh = None

if segments[1]!='':

db = segments[1]

if len(segments)>2:

tb = segments[2]

if len(segments)>>3:

wh = segments[3]

data = self.GetData()

if data is None:

self.Send(400,'Nothing to do')

return

conn = mysql.connector.connect(user='admin',password='apwd',database=db)

if conn is None:

self.Send(400,'Cannot connect')

return

try:

data = ast.literal\_eval(data)

cmd = ['update '+tb+' set ']

cm = ''

for k in data.keys():

v = data[k]

q = ''

if type(v) is str:

q = "'"

cmd.append(cm+k+'='+q+str(v)+q)

cm = ','

conn.cmd\_query(''.join(cmd)+' where '+wh)

conn.commit()

self.Send(200,'OK')

except Exception as e:

self.Send(403,e.msg)

conn.close()

return

def do\_DELETE(self):

segments = self.path.split('/')

db = None

tb = None

wh = None

if segments[1]!='':

db = segments[1]

if len(segments)>2:

tb = segments[2]

if len(segments)>>3:

wh = segments[3]

conn = mysql.connector.connect(user='admin',password='apwd',database=db)

if conn is None:

self.Send(400,'Cannot connect')

return

try:

data = ast.literal\_eval(data)

cmd = 'delete from '+tb

if wh is not None:

cmd += ' where '+wh

conn.cmd\_query(cmd)

conn.commit()

self.Send(200,'OK')

except Exception as e:

self.Send(403,e.msg)

conn.close()

return

## Authentication

1. We now implement basic Http authentication using the standard Authorization. We note that basic HTTP authentication is secure if used with https. In Restif.py, add at the top

import base64

add a Connect method to our RestIfHandler class

def Connect(self,db):

usr = 'admin'

pwd = 'apwd'

h = self.headers['Authorization']

if len(h)>6:

d = str(base64.b64decode(h[6:len(h)]),'utf-8') # 6 is len('Basic ')

s = d.split(':')

usr = s[0]

pwd = s[1]

return mysql.connector.connect(user=usr,password=pwd,database=db)

and use self.Connect(db) instead of calling mysql.connector.connect each time.

If you find any issues with this tutorial, please write to me: [Malcolm.Crowe@tawqt.com](mailto:Malcolm.Crowe@tawqt.com) .

# Appendix C

# A RESTIF Server in C#: step by step

The purpose of this article is to develop a generic Web server in Python 3.6 that can provide a simple REST interface to any MySQL[[3]](#footnote-3) database without configuration, supporting a variety of formats including Json. The only assumption is that RESTIF runs on the same machine as the database: any authentication credentials required for database access will be supplied in the HTTP requests.

We could use HTTP just as transport, for example, to POST a set of SQL commands to a database to be run as a well-configured transaction. Since there is no totally safe way of committing a transaction that involves more than one database (this is the famous two-army situation), we will limit our discussion to such transactions.

The real purpose of the RESTIF server is to support REST-style HTTP requests (GET, POST, PUT and DELETE) requests from a web application, not from a browser. If the web application is implemented using script Web pages, it is simplest to supply data in JSON format, but other formats could well be useful.

Since HTTP runs on top of TCP, there is a scalability issue since the TCP connection renegotiates packet length every time the message length changes and this renegotiation typically takes 1 second. This is not normally a problem in web applications, and for simplicity in this worksheet we will not try to make all messages have the same length. Also, we will suppose that payload of GET responses is of a reasonable size (this can always be assured by appropriate design of the database and application).

We will motivate and achieve this work through a sample Payroll database in order to demonstrate the concepts. However, the RESTIF server will be generic, i.e. will not have any of our Payroll details hardwired. This means the RESTIF software developed in this worksheet can be used for any MySQL database without change.

## A generic REST API

Every REST API involves designing what the URL pattern is and what verbs are allowed.

Let us keep things very simple. The following lines show the path(i.e. the portion of the URL that immediately follows the hostname:port prefix). We assume the server only interacts with MySQL on the given host, and that the username and password will come from an Authentication header in the request. Apart from the first two lines in this table, db represents an existing database, and tb an existing table in that database.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Verb | Path | Request data | Returns | Notes |
| GET | / |  | An array of strings | The result of show databases; |
| GET | /db |  | An array of strings | The result of connect db; show tables; |
| GET | /db/tb |  | An array of rows | The result of select \* from tb; |
| GET | /db/tb/s |  | An array of rows | The result of select \* from tb where s;  e.g. s might be id=5 and rvv=17 |
| POST | /db | a list of SQL statements |  | The statements are executed in a transaction |
| POST | /db/tb | a single row |  | The row will be added to the table |
| PUT | /db/tb/s | a partial row |  | The row(s) matching s will be modified using the given Json object |
| DELETE | /db/tb |  |  | Delete all rows in tb |
| DELETE | /db/tb/s |  |  | Delete all rows in tb matching condition s |

Note that we always select rows in the table using the URL (using the selection condition s). If we sometimes used the Json data to select rows the API would still be easy to implement, but harder to explain and use.

## An Employee management system

The tables we will use for the demonstration are[[4]](#footnote-4):

create table employee (id integer primary key, name varchar(20), notes varchar(1000));

create table posting (empid integer, start date, grade varchar(4), manager integer,foreign key (empid) references employee(id),foreign key (manager) references employee(id));

create table holiday (empid integer, start date, end date, agreed date,foreign key (empid) references employee(id));

And let’s give them some initial data:

insert into EMPLOYEE values(1562,'John Black','Sales');

insert into EMPLOYEE values(1567,'Mary White','Finance');

insert into EMPLOYEE values(1569,'Paul Green','HR');

insert into POSTING values(1562,'2012-02-01','A1',1569);

insert into POSTING values(1562,'2012-04-01','A2',null);

insert into POSTING values(1567,'2012-02-01','B1',1569);

insert into POSTING values(1569,'2012-02-01','A2',null);

insert into HOLIDAY values(1567,'2012-03-02','2012-03-07','2012-02-05');

insert into HOLIDAY values(1569,'2012-04-04','2012-04-18',null);

EMPLOYEE

|  |  |  |
| --- | --- | --- |
| 1562 | John Black | Sales |
| 1567 | Mary White | Finance |
| 1569 | Paul Green | HR |

POSTING

|  |  |  |  |
| --- | --- | --- | --- |
| 1562 | 1/2/2012 | A1 | 1569 |
| 1562 | 1/4/2012 | A2 |  |
| 1567 | 1/2/2012 | B1 | 1569 |
| 1569 | 1/2/2012 | A2 |  |

HOLIDAY

|  |  |  |  |
| --- | --- | --- | --- |
| 1567 | 2/3/2012 | 7/3/2012 | 5/2/2012 |
| 1569 | 4/4/2012 | 18/4/2012 | 3/4/2012 |

We will grant all privileges on this database to a local user called admin (admin does not have a login account on the server).

create user 'admin' identified by 'apwd';

grant all on Payroll.\* to 'admin' identified by 'apwd';

## A Simple Web Server

This Appendix develops RestifD.sln in stages. If you want to skip this, you can simply return to the section in Chapter 0 entitled “Installing RESTIF on Windows.

I use Visual Studio. These notes are written for Visual Studio 2017, to take advantage of some C# shortcut coding.

1. Create a New Windows Classic Desktop Console Application Project called RestifD in the folder c:\restif . In Solution Explorer, rename Program.cs as Restif.cs and agree that the renaming should apply to the class “Program”. Replace the entire contents on Restif.cs with:

using System;

using System.Text;

using System.Net;

namespace RestifD

{

class Restif

{

static HttpListenerContext context;

static string host = "localhost";

static void Main(string[] args)

{

var listener = new HttpListener();

listener.Prefixes.Add("http://" + host + ":8078/");

listener.Start();

for (;;)

try

{

context = listener.GetContext();

Run();

}

catch(Exception e)

{

Console.WriteLine(e.Message);

break;

}

}

static void Run()

{

var meth = context.Request.HttpMethod;

string db = null;

string mess = null;

switch (meth)

{

case "GET":

mess = Get1(db);

break;

}

Send(200,mess);

}

static string Get1(string db) { return "TBD G1"; }

static void Send(int status,string mess)

{

var b = Encoding.UTF8.GetBytes(mess);

var c = context.Response;

c.StatusCode = status;

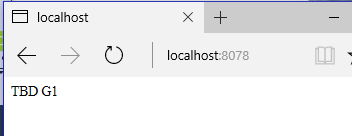
c.StatusDescription = (status == 200) ? "OK" : "ERROR";

c.ContentLength64 = b.Length;

var st = c.OutputStream;

st.Write(b, 0, b.Length);

st.Close();

 }

}

}

Try running this program. The program itself shows a blank window. But if you try <Http://localhost:8078> in your browser you can see it is working.

This program is a simple server (rather like the first stage of the Python version in Appendix B). But we have ambitions to make it multithreaded, and we will do that now. (Note that every time we want to change the code we have to stop the server.)

1. The first step is to make a fresh instance of the Restif class for each request, and make most of the definitions above into properties and methods of the instance. This will ensure they don’t interfere with each other.

using System;

using System.Text;

using System.Net;

namespace RestifD

{

class Restif

{

HttpListenerContext context;

static string host = "localhost";

static void Main(string[] args)

{

var listener = new HttpListener();

listener.Prefixes.Add("http://" + host + ":8078/");

listener.Start();

for (;;)

try

{

new Restif { context = listener.GetContext() }.Run();

}

catch(Exception e)

{

Console.WriteLine(e.Message);

break;

}

}

void Run()

{

var meth = context.Request.HttpMethod;

string db = null;

string mess = null;

switch (meth)

{

case "GET":

mess = Get1(db);

break;

}

Send(200,mess);

}

string Get1(string db) { return "TBD G1"; }

void Send(int status,string mess)

{

var b = Encoding.UTF8.GetBytes(mess);

var c = context.Response;

c.StatusCode = status;

c.StatusDescription = (status == 200) ? "OK" : "ERROR";

c.ContentLength64 = b.Length;

var st = c.OutputStream;

st.Write(b, 0, b.Length);

st.Close();

}

}

}

Check this still works.

1. Now let’s make each Run into a separate Task, so they can be executed in parallel: we want the server to deal with the next request while the previous ones are executing. Very little change is required for this:

using System;

using System.Text;

using System.Net;

using System.Threading.Tasks;

namespace RestifD

{

class Restif

{

HttpListenerContext context;

static string host = "localhost";

static void Main(string[] args)

{

var listener = new HttpListener();

listener.Prefixes.Add("http://" + host + ":8078/");

listener.Start();

for (;;)

try

{

new Restif { context = listener.GetContext() }.Run();

}

catch(Exception e)

{

Console.WriteLine(e.Message);

break;

}

}

void Run()

{

Task.Run(() =>

{

var meth = context.Request.HttpMethod;

string db = null;

string mess = null;

switch (meth)

{

case "GET":

mess = Get1(db);

break;

}

Send(200, mess);

});

}

string Get1(string db) { return "TBD G1"; }

void Send(int status,string mess)

{

var b = Encoding.UTF8.GetBytes(mess);

var c = context.Response;

c.StatusCode = status;

c.StatusDescription = (status == 200) ? "OK" : "ERROR";

c.ContentLength64 = b.Length;

var st = c.OutputStream;

st.Write(b, 0, b.Length);

st.Close();

}

}

}

This web server is multithreaded and production ready. (Check it still works.) Let’s make it a bit more useful.

1. As a next step, let’s start working with MySQL. In Solution Explorer, right-click the project and select Add>References..>Assemblies>Extensions>MySql.Data. Modify the code as follows:

using System;

using System.Text;

using System.Net;

using System.Threading.Tasks;

using MySql.Data.MySqlClient;

namespace RestifD

{

class Restif

{

HttpListenerContext context;

static string host = "localhost";

MySqlConnection conn = null;

static void Main(string[] args)

{

var listener = new HttpListener();

listener.Prefixes.Add("http://" + host + ":8078/");

listener.Start();

for (;;)

try

{

new Restif { context = listener.GetContext() }.Run();

}

catch(Exception e)

{

Console.WriteLine(e.Message);

break;

}

}

void Run()

{

Task.Run(() =>

{

var meth = context.Request.HttpMethod;

string db = null;

var connstring = "server=" + host + ";uid=admin;password=apwd";

string mess = null;

try

{

conn = new MySqlConnection(connstring);

conn.Open();

switch (meth)

{

case "GET":

mess = Get1(db);

break;

}

Send(200, mess);

} catch (Exception e)

{

Send(403, e.Message);

}

conn?.Close();

});

}

string Get1(string db)

{

var cmd = conn.CreateCommand();

cmd.CommandText = "show databases";

var rdr = cmd.ExecuteReader();

var r = "";

while (rdr.Read())

r += " " + rdr[0].ToString();

return r;

}

void Send(int status,string mess)

{

var b = Encoding.UTF8.GetBytes(mess);

var c = context.Response;

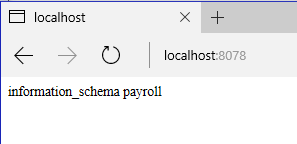
c.StatusCode = status;

c.StatusDescription = (status == 200) ? "OK" : "ERROR";

c.ContentLength64 = b.Length;

var st = c.OutputStream;

st.Write(b, 0, b.Length);

 st.Close();

}

}

}

This makes each request run in a separate connection to the database. We have added exception handling to ensure these connections all get closed whatever happens. And we have made Get1 return the list of databases accessible by the given user.

1. Now let’s collect a database name if the URL contains one, and list the tables it contains.

string get(string[] p, int i)

{

return (i < p.Length) ? (p[i] != "") ? p[i].Trim('/') : null : null;

}

void Run()

{

Task.Run(() =>

{

var meth = context.Request.HttpMethod;

var path = context.Request.Url.Segments;

string db = get(path,1);

var connstring = "server=" + host + ";uid=admin;password=apwd";

if (db != null)

connstring += ";database=" + db;

string mess = null;

try

{

conn = new MySqlConnection(connstring);

conn.Open();

switch (meth)

{

case "GET":

mess = Get1(db);

break;

}

Send(200, mess);

} catch (Exception e)

{

Send(403, e.Message);

}

conn?.Close();

});

}

string Get1(string db)

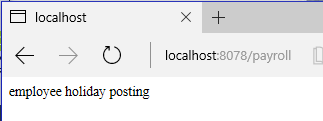
{

var cmd = conn.CreateCommand();

cmd.CommandText = (db==null)?"show databases":"show tables";

var rdr = cmd.ExecuteReader();

var r = "";

 while (rdr.Read())

r += " " + rdr[0].ToString();

return r;

}

Check this works as before and with the url <http://localhost:8078/payroll> .

1. Now let’s look to see if we have a tablename and maybe a where condition in the path.

void Run()

{

Task.Run(() =>

{

var meth = context.Request.HttpMethod;

var path = context.Request.Url.Segments;

string db = get(path,1);

string tb = get(path, 2);

string wh = get(path, 3);

var connstring = "server=" + host + ";uid=admin;password=apwd";

if (db != null)

connstring += ";database=" + db;

string mess = null;

try

{

conn = new MySqlConnection(connstring);

conn.Open();

switch (meth)

{

case "GET":

mess = (tb==null)?Get1(db):Get2(tb,wh);

break;

}

Send(200, mess);

} catch (Exception e)

{

Send(403, e.Message);

}

conn?.Close();

});

}

string Get1(string db)

{

var cmd = conn.CreateCommand();

cmd.CommandText = (db==null)?"show databases":"show tables";

var rdr = cmd.ExecuteReader();

var r = "";

while (rdr.Read())

r += " " + rdr[0].ToString();

return r;

}

string Get2(string tb, string wh)

{

var cmd = conn.CreateCommand();

cmd.CommandText = "select \* from " + tb;

if (wh != null)

cmd.CommandText += " where " + wh;

var rdr = cmd.ExecuteReader();

var r = "";

while (rdr.Read())

{

var c = "";

for (var i = 0; i < rdr.VisibleFieldCount; i++)

{

r += c+ rdr.GetName(i)+"="+ rdr[i].ToString();

c = ",\n";

}

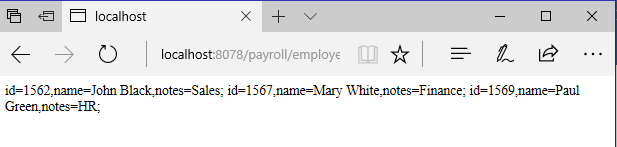
r += ";";

}

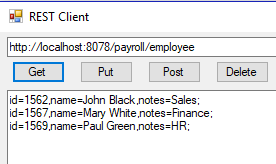
return r;

}

Try this out with <http://localhost:8078/payroll/employee>



Hmm. The browser expects to ignore newlines. So let’s change to using RESTClient instead:



This might be okay if the client is accepting text/plain.

1. Let’s fix quotation marks around fields that need them:

string Get2(string tb, string wh)

{

var cmd = conn.CreateCommand();

cmd.CommandText = "select \* from " + tb;

if (wh != null)

cmd.CommandText += " where " + wh;

var rdr = cmd.ExecuteReader();

var r = "";

while (rdr.Read())

{

var c = "";

for (var i = 0; i < rdr.VisibleFieldCount; i++)

{

var sv = rdr[i].ToString();

r += c+ rdr.GetName(i)+"="+ Format(rdr[i]);

c = ",";

}

r += ";\n";

}

return r;

}

string Format(object ob)

{

if (ob is DBNull)

return "null";

if (ob is DateTime)

{

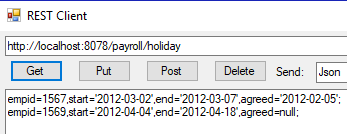
var dt = (DateTime)ob;

return "'"+dt.ToString("yyyy-MM-dd")+"'";

}

if (ob is string)

return "'" + ob.ToString() + "'";

 return ob.ToString();

}

This now also works for the holiday table, as shown. But we want to support JavaScript clients.

1. Let’s check the Accept header and use Json formatting if this is what the user wants.

string Get2(string tb, string wh)

{

var cmd = conn.CreateCommand();

cmd.CommandText = "select \* from " + tb;

if (wh != null)

cmd.CommandText += " where " + wh;

var rdr = cmd.ExecuteReader();

// 0: fieldstart 1: fieldend 2: liststart 3: listmid 4: listend

// 5: rowstart 6: rowmid 7: rowend

var fmt = new string[] { "","=","", "\n", "", "", ",", ";" };

if (context.Request.Headers["Accept"] == "application/json")

fmt = new string[] { "\"", "\": ", "[", ",\n", "]", "{", ", ", "}" };

var r = fmt[2];

var m = "";

while (rdr.Read())

{

var c = "";

r += m;

m = fmt[3];

r += fmt[5];

for (var i = 0; i < rdr.VisibleFieldCount; i++)

{

var sv = rdr[i].ToString();

r += c+ fmt[0]+ rdr.GetName(i)+fmt[1]+ Format(rdr[i]);

c = fmt[6];

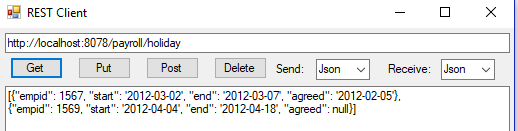
}

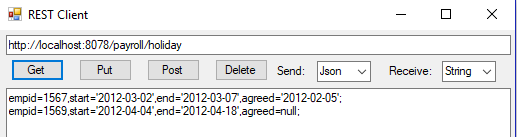
r += fmt[7];

}

return r + fmt[4];

}

Check this works: 



1. Let’s avoid using += on strings – it is very slow.

string Get2(string tb, string wh)

{

var cmd = conn.CreateCommand();

cmd.CommandText = "select \* from " + tb;

if (wh != null)

cmd.CommandText += " where " + wh;

var rdr = cmd.ExecuteReader();

// 0: fieldstart 1: fieldend 2: liststart 3: listmid 4: listend

// 5: rowstart 6: rowmid 7: rowend

var fmt = new string[] { "","=","", "\n", "", "", ",", ";" };

if (context.Request.Headers["Accept"] == "application/json")

fmt = new string[] { "\"", "\": ", "[", ",\n", "]", "{", ", ", "}" };

var r = new StringBuilder(fmt[2]);

var m = "";

while (rdr.Read())

{

var c = "";

r.Append(m);

m = fmt[3];

r.Append(fmt[5]);

for (var i = 0; i < rdr.VisibleFieldCount; i++)

{

var sv = rdr[i].ToString();

r.Append(c+ fmt[0]+ rdr.GetName(i)+fmt[1]+ Format(rdr[i]));

c = fmt[6];

}

r.Append(fmt[7]);

}

return r + fmt[4];

}

Check this gives the same outputs.

## Getting started with POST

1. Let’s add a method to get hold of posted data. We will need to add using System.IO at the top of the file:

string Receive()

{

var r = context.Request;

if (!r.HasEntityBody)

return "No data??";

var rdr = new StreamReader(r.InputStream, r.ContentEncoding);

var rs = rdr.ReadToEnd();

r.InputStream.Close();

rdr.Close();

return rs;

}

1. Now let’s implement the first POST method in our API:

void Run()

{

Task.Run(() =>

{

var meth = context.Request.HttpMethod;

var path = context.Request.Url.Segments;

string db = get(path,1);

string tb = get(path, 2);

string wh = get(path, 3);

var connstring = "server=" + host + ";uid=admin;password=apwd";

if (db != null)

connstring += ";database=" + db;

string mess = null;

try

{

conn = new MySqlConnection(connstring);

conn.Open();

switch (meth)

{

case "GET":

mess = (tb==null)?Get1(db):Get2(tb,wh);

break;

case "POST":

if (db!=null)

mess = Post1(Receive());

break;

}

Send(200, mess);

} catch (Exception e)

{

Send(403, e.Message);

}

conn?.Close();

});

}

string Post1(string data)

{

var tr = conn.BeginTransaction(System.Data.IsolationLevel.Serializable);

foreach (var s in data.Split(';'))

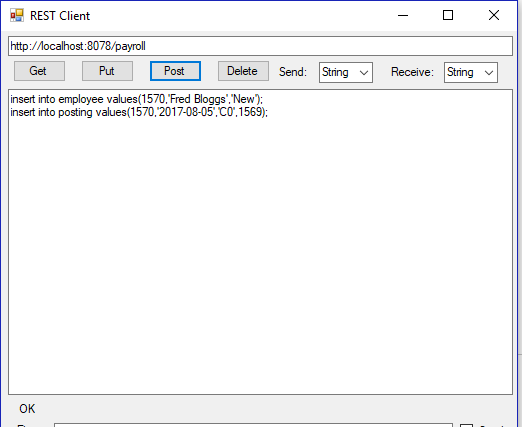
{

var cmd = conn.CreateCommand();

cmd.Transaction = tr;

cmd.CommandText = s.Trim();

if (cmd.CommandText == "")

 continue;

cmd.ExecuteNonQuery();

}

tr.Commit();

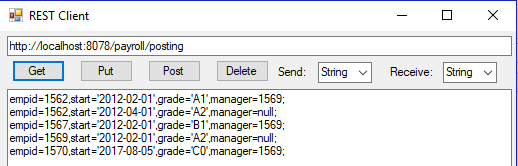
return "OK";

}

Try this out: In RESTClient place the following SQL statements in the big textbox.

insert into employee values(1570,'Fred Bloggs','New');

insert into posting values(1570,'2017-08-05','C0',1569);

give the URL as <http://localhost:8078/payroll> and and click Post (both dropboxes should say String but we aren’t looking at them).

Do a GET on posting to check we now have Fred. (Note that repeating the POST unchanged will fail as you will have a duplicate key in the employee table.)

1. For the second POST method in our API we need to examine the Json data. I know there is an excellent Json library for C# from Newtonsoft, but I don’t like adding a 10MB library when 6KB will do. So let’s add the Json parser in Document.cs to our project. From Solution Explorer use Add>Existing Item..
2. So make the following changes to implement the second POST method:

void Run()

{

Task.Run(() =>

{

var meth = context.Request.HttpMethod;

var path = context.Request.Url.Segments;

string db = get(path,1);

string tb = get(path, 2);

string wh = get(path, 3);

var connstring = "server=" + host + ";uid=admin;password=apwd";

if (db != null)

connstring += ";database=" + db;

string mess = null;

try

{

conn = new MySqlConnection(connstring);

conn.Open();

switch (meth)

{

case "GET":

mess = (tb==null)?Get1(db):Get2(tb,wh);

break;

case "POST":

var pd = Receive();

if (tb != null)

mess = Post2(tb, new Document(pd));

else if (db != null)

mess = Post1(pd);

else

throw new Exception("not supported");

break;

}

Send(200, mess);

} catch (Exception e)

{

Send(403, e.Message);

}

conn?.Close();

});

}

string Post2(string tb,Document d)

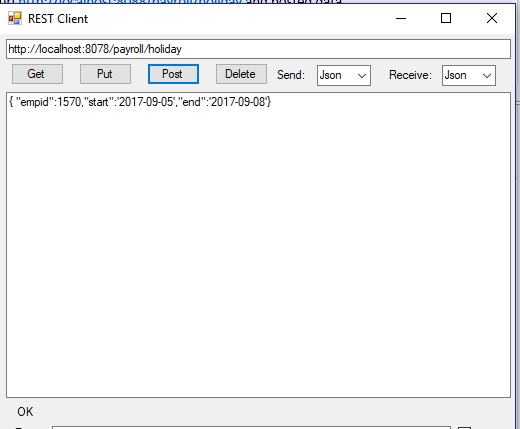
{

var cls = "insert into " + tb + "(";

var vls = ") values (";

var cm = "";

foreach (var e in d.fields)

 {

cls += cm + e.Key;

vls += cm + Format(e.Value);

cm = ",";

}

var cmd = conn.CreateCommand();

cmd.CommandText = cls + vls + ")";

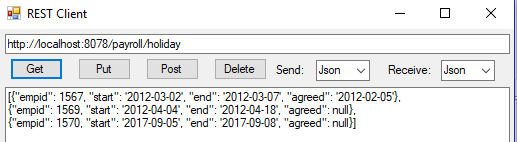
cmd.ExecuteNonQuery();

return "OK";

}

As you can see, we use the Document class to parse the Json, and construct an SQL insert statement to send to the database. Let us verify this works by posting a Holiday for Fred. The address bar should contain <http://localhost:8078/payroll/holiday> , the dropdowns should be Json (though we don’t check them), and we need a Json object in the big textbox.

{ "empid":1570,"start":'2017-09-05',"end":'2017-09-08'}

Click the Post button:

Verify that worked using GET.

## PUT and DELETE

1. These are now very straightforward:

void Run()

{

Task.Run(() =>

{

var meth = context.Request.HttpMethod;

var path = context.Request.Url.Segments;

string db = get(path,1);

string tb = get(path, 2);

string wh = get(path, 3);

var connstring = "server=" + host + ";uid=admin;password=apwd";

if (db != null)

connstring += ";database=" + db;

string mess = null;

try

{

conn = new MySqlConnection(connstring);

conn.Open();

switch (meth)

{

case "GET":

mess = (tb==null)?Get1(db):Get2(tb,wh);

break;

case "POST":

var pd = Receive();

if (tb != null)

mess = Post2(tb, new Document(pd));

else if (db != null)

mess = Post1(pd);

else

throw new Exception("not supported");

break;

case "PUT":

if (db == null || tb == null || wh == null)

throw new Exception("not supported");

mess = Put(tb,wh,new Document(Receive()));

break;

case "DELETE":

if (db == null || tb == null)

throw new Exception("not supported");

mess = Delete(tb,wh);

break;

}

Send(200, mess);

} catch (Exception e)

{

Send(403, e.Message);

}

conn?.Close();

});

}

string Put(string tb,string wh,Document d)

{

var cmt = "update " + tb + " set ";

var cm = "";

foreach (var e in d.fields)

{

cmt += cm + e.Key + "=" + Format(e.Value);

cm = ",";

}

var cmd = conn.CreateCommand();

cmd.CommandText = cmt+" where "+wh;

cmd.ExecuteNonQuery();

return "OK";

}

string Delete(string tb,string wh)

{

var cmd = conn.CreateCommand();

cmd.CommandText = "delete from " + tb;

if (wh != null)

cmd.CommandText += " where " + wh;

return "" + cmd.ExecuteNonQuery() + " rows deleted";

}

## Authentication

1. Let’s require authentication and check out the Authorization header:

static void Main(string[] args)

{

var listener = new HttpListener();

listener.Prefixes.Add("http://" + host + ":8078/");

listener.AuthenticationSchemes = AuthenticationSchemes.Basic;

listener.Start();

for (;;)

try

{

new Restif { context = listener.GetContext() }.Run();

}

catch(Exception e)

{

Console.WriteLine(e.Message);

break;

}

}

void Run()

{

Task.Run(() =>

{

var meth = context.Request.HttpMethod;

var path = context.Request.Url.Segments;

var h = context.Request.Headers["Authorization"];

var s = Encoding.UTF8.GetString(

Convert.FromBase64String(h.Substring(6))).Split(':');

string db = get(path, 1);

string tb = get(path, 2);

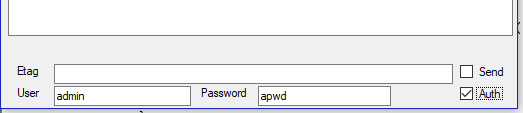
string wh = get(path, 3);

var connstring = "server=" + host + ";uid="+s[0]+";password="+s[1];

if (db != null)

connstring += ";database=" + db;

With this change you will always need to fill in the bottom line of RESTClient with correct credentials (in this case root/admin annd admin/apwd will work) and check the Auth box.



## Final touches

1. We need to be able to run the server for different hosts:

static void Main(string[] args)

{

if (args.Length > 0)

host = args[0];

var listener = new HttpListener();

listener.Prefixes.Add("http://" + host + ":8078/");

1. There are some standard headers we should add in our responses:

void Send(int status,string mess)

{

var b = Encoding.UTF8.GetBytes(mess);

var c = context.Response;

c.StatusCode = status;

c.StatusDescription = (status == 200) ? "OK" : "ERROR";

c.AddHeader("Cache-control", "no-store");

c.AddHeader("Expires", "-1");

c.AddHeader("Pragma", "no-cache");

c.ContentLength64 = b.Length;

var st = c.OutputStream;

st.Write(b, 0, b.Length);

st.Close();

}

This completes the construction of the Restif web server.

1. We are using CentOS for the server (if separate), which refers to MySQL as mariadb. [↑](#footnote-ref-1)
2. From [MySQL documentation](https://dev.mysql.com/doc/refman/5.5/en/create-table.html#create-table-reference-definition): *MySQL parses but ignores “inline REFERENCES specifications” (as defined in the SQL standard) where the references are defined as part of the column specification. MySQL accepts REFERENCES clauses only when specified as part of a separate FOREIGN KEY specification.* The explicit column reference ID is also required here. [↑](#footnote-ref-2)
3. We are using CentOS for the server (if separate), which refers to MySQL as mariadb. [↑](#footnote-ref-3)
4. From [MySQL documentation](https://dev.mysql.com/doc/refman/5.5/en/create-table.html#create-table-reference-definition): *MySQL parses but ignores “inline REFERENCES specifications” (as defined in the SQL standard) where the references are defined as part of the column specification. MySQL accepts REFERENCES clauses only when specified as part of a separate FOREIGN KEY specification.* The explicit column reference ID is also required here. [↑](#footnote-ref-4)