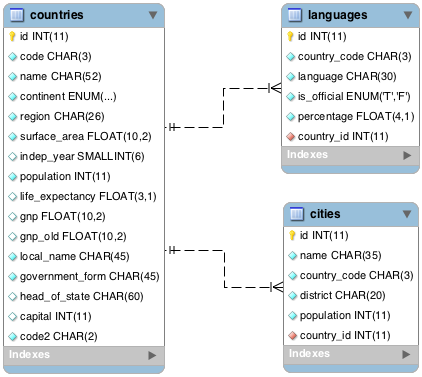
MYSQL: WORLD

**Assignment: MySQL Countries**

Using the world database and the pictured ERD, complete all 8 of the below queries.



First grab the ***world*** database [(download the file here](http://s3.amazonaws.com/General_V88/boomyeah/company_209/chapter_3569/handouts/chapter3569_5432_world.sql))

If possible, work in groups and use MySQL Workbench to do the queries. We want you to get familiar with using MySQL workbench. The questions are of varying difficulty level. Do the easier ones first.

**Queries**

1. What query would you run to get all the countries that speak Slovene? Your query should return the name of the country, language and language percentage. Your query should arrange the result by language percentage in descending order. (1)

2. What query would you run to display the total number of cities for each country? Your query should return the name of the country and the total number of cities. Your query should arrange the result by the number of cities in descending order. (3)

3. What query would you run to get all the cities in Mexico with a population of greater than 500,000? Your query should arrange the result by population in descending order. (1)

4. What query would you run to get all languages in each country with a percentage greater than 89%? Your query should arrange the result by percentage in descending order. (1)

5. What query would you run to get all the countries with Surface Area below 501 and Population greater than 100,000? (2)

6. What query would you run to get countries with only Constitutional Monarchy with a capital greater than 200 and a life expectancy greater than 75 years? (1)

7. What query would you run to get all the cities of Argentina inside the Buenos Aires district and have the population greater than 500, 000? The query should return the Country Name, City Name, District and Population. (2)

8. What query would you run to summarize the number of countries in each region? The query should display the name of the region and the number of countries. Also, the query should arrange the result by the number of countries in descending order. (2)

Note: You may download this PDF file displaying the expected results from the queries - [**Expected Results (World)**](http://s3.amazonaws.com/General_V88/boomyeah/company_209/chapter_3569/handouts/chapter3569_5438_MySQL-Basic-World-Expected-Result.pdf)

MYSQL: SAKILA

## Assignment: Sakila

Using the Sakila database, complete the below queries.

You can get the **Sakila** database and ERD here ([sakila-data.sql](http://s3.amazonaws.com/General_V88/boomyeah/company_209/chapter_3569/handouts/chapter3569_5430_sakila-data.sql) and [sakila-db-model.png](http://s3.amazonaws.com/General_V88/boomyeah/company_209/chapter_3569/handouts/chapter3569_5431_sakila-db-model.png)), please use these for reference.

### Note

Earlier in this section, we recommended for students name their tables all lower case and have a primary key called 'id' in each table. These are the rules we follow, however not all developers follow these rules. The SQL file you'll be working with does NOT follow the rules we discussed, including naming the fields lower case. We still want you to follow the rules we taught, but use this as an opportunity to get comfortable with other SQL files that do not completely follow the rules of normalization.

### Queries

1. What query would you run to get all the customers inside city\_id = 312? Your query should return customer first name, last name, email, and address.

2. What query would you run to get all comedy films? Your query should return film title, description, release year, rating, special features, and genre (category).

3. What query would you run to get all the films joined by actor\_id=5? Your query should return the actor id, actor name, film title, description, and release year.

4. What query would you run to get all the customers in store\_id = 1 and inside these cities (1, 42, 312 and 459)? Your query should return customer first name, last name, email, and address.

5. What query would you run to get all the films with a "rating = G" and "special feature = behind the scenes", joined by actor\_id = 15? Your query should return the film title, description, release year, rating, and special feature. Hint: You may use LIKE function in getting the 'behind the scenes' part.

6. What query would you run to get all the actors that joined in the film\_id = 369? Your query should return the film\_id, title, actor\_id, and actor\_name.

7. What query would you run to get all drama films with a rental rate of 2.99? Your query should return film title, description, release year, rating, special features, and genre (category).

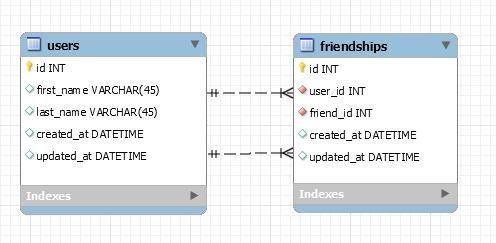
8. What query would you run to get all the action films which are joined by SANDRA KILMER? Your query should return film title, description, release year, rating, special features, genre (category), and actor's first name and last name.

Note: You may download this PDF file displaying the expected results from the questions above - [**Expected Result (Sakila)**](http://s3.amazonaws.com/General_V88/boomyeah/company_209/chapter_3569/handouts/chapter3569_5439_MySQL-Intermediate-Sakila-Expected-Result.pdf)

MYSQL: FRIENDSHIPS

**Assignment: Friendships**

Using the below ERD, write the SQL query that returns a list of users along with their friends' names.



Create the tables and populate them with some fake data.  Your results should look like below:

|  |  |  |  |
| --- | --- | --- | --- |
| first\_name | last\_name | friend\_first\_name | friend\_last\_name |
| Chris | Baker | Jessica | Davidson |
| Chris | Baker | James | Johnson |
| Chris | Baker | Diana | Smith |
| Diana | Smith | Chris | Baker |
| James | Johnson | Chris | Baker |
| Jessica | Davidson | Chris | Baker |

Your actual query will look something similar to this:

SELECT \* FROM users

LEFT JOIN friendships ON \_\_\_\_=\_\_\_\_

LEFT JOIN users as user2 ON \_\_\_\_ = \_\_\_\_

Take note that we're joining the ***users*** table again but we're specifying the second ***users*** table **as** user2.  You can then reference the second ***users***by calling user2 (e.g. user2.id, user2.first\_name, etc).

You can also rename the fields that are displayed on the result by using the **as** keyword, like the below example:

SELECT user2.first\_name as friend\_first\_name, user2.last\_name as friend\_last\_name, ... FROM ...

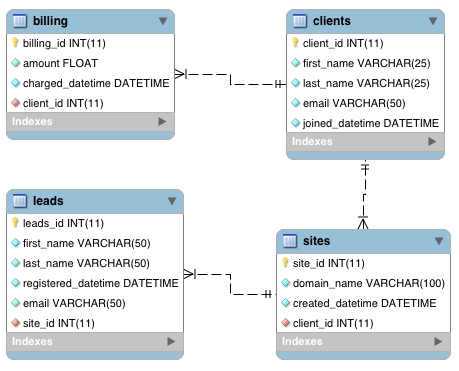
Knowing how to do joins can be tricky but is used quite often *and will most likely appear again in your belt exam.*

Note: The order which we return the results is alphabetical by friend\_last\_name.

MYSQL: LEAD GEN BUSINESS

## Assignment: Lead Gen Business

Complete the below SQL queries using the lead-gen-business-new database and the below wireframe.



### Note

If you're ahead, we strongly encourage that you work on this assignment as this would really help you understand how GROUP BY work and how powerful MySQL could be. If you have already spent more than 2 full days studying ERD and MySQL, just skip this assignment and come back later when you're done with the bootcamp.

If you already have a database called **lead-gen-business**, go ahead and drop that database and recreate by importing the [lead-gen-business-new.sql](http://s3.amazonaws.com/General_V88/boomyeah/company_209/chapter_3569/handouts/chapter3569_5435_lead-gen-business-new.sql) file. The database that is created will be different than the one in the **morepractice.sql** file that we used during the videos.

### Queries

1. What query would you run to get the total revenue for March of 2012?

2. What query would you run to get total revenue collected from the client with an id of 2?

3. What query would you run to get all the sites that client=10 owns?

4. What query would you run to get total # of sites created per month per year for the client with an id of 1? What about for client=20?

5. What query would you run to get the total # of leads generated for each of the sites between January 1, 2011 to February 15, 2011?

6. What query would you run to get a list of client names and the total # of leads we've generated for each of our clients between January 1, 2011 to December 31, 2011?

7. What query would you run to get a list of client names and the total # of leads we've generated for each client each month between months 1 - 6 of Year 2011?

8. What query would you run to get a list of client names and the total # of leads we've generated for each of our clients' sites between January 1, 2011 to December 31, 2011? Order this query by client id.  Come up with a **second** query that shows all the clients, the site name(s), and the total number of leads generated from each site for all time.

9. Write a single query that retrieves total revenue collected from each client for each month of the year. Order it by client id.

10. Write a single query that retrieves all the sites that each client owns. Group the results so that each row shows a new client. It will become clearer when you add a new field called 'sites' that has all the sites that the client owns. (HINT: use GROUP\_CONCAT)

Note: You may download this PDF file displaying the expected results from the questions above - [Expected Result (Leads)](http://s3.amazonaws.com/General_V88/boomyeah/company_209/chapter_3569/handouts/chapter3569_5440_MySQL-Optional-Leads-Expected-Result.pdf)

Flask + mysql

# Overview

Adding a database to your application is very simple, but extremely powerful. We've seen how we can get information posted from a form, validate the information to make sure it contains the proper information, display flash messages, and set some session variables. Now we are going to show you how you can put that information into your database and use session variables to log a user in! With the database in the mix, we can now start to make our content dynamically change based on a particular user or condition. This is where the fun begins! In this chapter we will cover the following:

1. **Connecting to MySQL**
2. **Learn how to Import and Export MySQL Database**
3. **Including files**
4. **Database Communication**
5. **Login and Registration**
6. **MySQL Injection**

# Import/Export DB

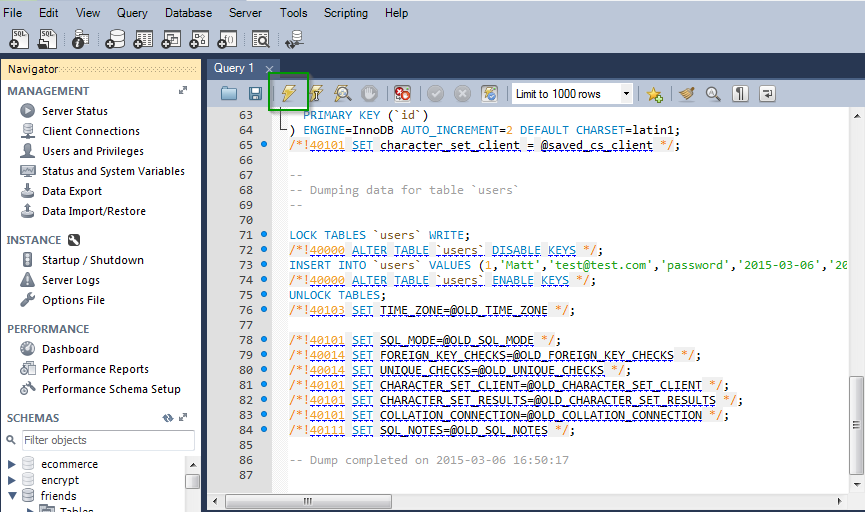
**Note:**We will be mostly creating our own database/erd and forward engineering into a database.

You may use either MySQL Workbench or phpMyAdmin in exporting and importing database (SQL) files.

## Using MySQL Workbench

#### Import:

Copy the SQL commands from the .sql file that you wish to import. Paste the copied contents to MySQL Workbench query window and click the bolt icon.

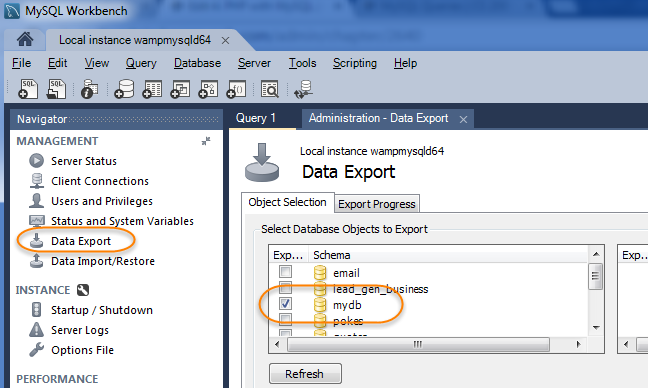


Upon successful run of your query, refresh the **Schemas**section (on the left side of Workbench) and your newly imported database will appear.

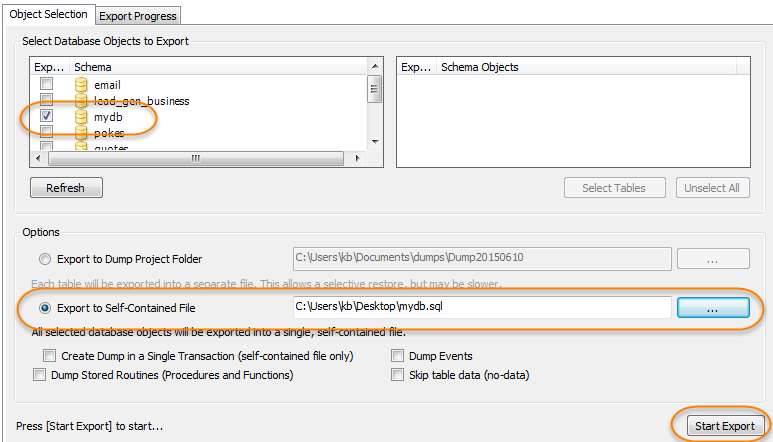
You may also use the Data Import/Restore option on Workbench to import database.

#### Export:

Click the Data Export option on your Workbench and choose the database/schema you wish to export.



Choose the **Export to Self-contained file**option. Be sure to indicate file name (don't forget to add .sql as the file extension) and specify the path to where you are saving the exported SQL file.



Then click Start Export. This will generate a .sql file which contains MySQL commands to duplicate the database you created.

# MySQL Connection

In this chapter, we are going to teach you how to connect to MySQL using a Python module named MySQL-python (for Mac) or mysqlclient (for PC). Keep in mind while you are going through this chapter that we have provided you with a connection file so that you can learn how to interact with the database and run certain commands through an interface that we are providing. **You do not need to understand how to create the connection file (you will by the end of the bootcamp we promise) but you do need to understand how it works.**

The first step in connecting to the database for Mac users is using Homebrew to install mysql, which is separate from how you installed it previously in the MySQL chapter. Before running the following commands we should activate a virtual environment that we used during the Flask Basic chapter or create a new virtual environment that we will dedicate to Flask and MySQL **and** run the following commands:

FOR Mac users:

(venv) brew install mysql

(venv) pip install MySQL-python

(venv) pip install Flask-SQLAlchemy

FOR PC users:

(venv) pip install Flask-SQLAlchemy

(venv) pip install mysqlclient

**Mac users, If you are getting errors with installing MySQL-python please do not struggle with troubleshooting it.  The 20-minute rule does not apply to this!  Reach out to your instructor.**

Once you have your Flask-SQLAlchemy and MySQL packages installed successfully, create a database called "mydb" that has a users table with a name field. Add 2-3 users using MySQLWorkbench so that you can test out the connection.

**PC users, if you are getting errors following the above instructions, try installing a different version of the mysqlclient package:   
pip install mysqlclient==1.3.4**

Now create a new project called "flask\_mysql" and create a server.py file and a mysqlconnection.py file.

mysqlconnection.py will be the file that connects to MySQL using the MySQL-python module we installed earlier.  Copy the following code snippets into the appropriate files.

### /flask\_mysql/mysqlconnection.py

""" import the necessary modules """

from flask\_sqlalchemy import SQLAlchemy

from sqlalchemy.sql import text

# Create a class that will give us an object that we can use to connect to a database

*class* MySQLConnection(*object*):

*def* \_\_init\_\_(self, app, db):

config = {

'host': 'localhost',

'database': db, # we got db as an argument

'user': 'root',

'password': 'root',

'port': '3306' # change the port to match the port your SQL server is running on

}

# this will use the above values to generate the path to connect to your sql database

DATABASE\_URI = "mysql://{}:{}@127.0.0.1:{}/{}".format(config['user'], config['password'], config['port'], config['database'])

app.config['SQLALCHEMY\_DATABASE\_URI'] = DATABASE\_URI

app.config['SQLALCHEMY\_TRACK\_MODIFICATIONS'] = True

# establish the connection to database

self.db = SQLAlchemy(app)

# this is the method we will use to query the database

*def* query\_db(self, query, data=None):

result = self.db.session.execute(text(query), data)

if query[0:6].lower() == 'select':

# if the query was a select

# convert the result to a list of dictionaries

list\_result = [dict(r) for r in result]

# return the results as a list of dictionaries

return list\_result

elif query[0:6].lower() == 'insert':

# if the query was an insert, return the id of the

# commit changes

self.db.session.commit()

# row that was inserted

return result.lastrowid

else:

# if the query was an update or delete, return nothing and commit changes

self.db.session.commit()

# This is the module method to be called by the user in server.py. Make sure to provide the db name!

*def* MySQLConnector(app, db):

return MySQLConnection(app, db)

### /flask\_mysql/server.py

from flask import Flask

# import the Connector function

from mysqlconnection import MySQLConnector

app = Flask(\_\_name\_\_)

# connect and store the connection in "mysql" note that you pass the database name to the function

mysql = MySQLConnector(app, 'mydb')

# an example of running a query

print mysql.query\_db("SELECT \* FROM users")

app.run(debug=True)

Note that we are not handling any routes in our server.py file. Instead, when we run "python server.py" we should see our users printed to the terminal.

**First, run the application to make sure that you are getting your users from the database.**

### Things to know about the connection:

* Read all of the comments in the connection file to fully understand what is going on. Note that you don't need to know how to create one of these files -- instead, you should know how to use the file and by the end of the bootcamp you will be experienced enough to create your own connection files.

HW #1

## MySQL Connection Errors

Play around with the connection credentials and produce as many "mysql connection" errors as possible. Make sure you copy and paste those errors in a **.txt**file. Determine their error numbers as well and tell us how to fix it. Example:

**mysql.connector.errors.ProgrammingError: 1045 (28000): Access denied for user 'root'@'localhost' (using password: NO)**

**fix:**Make sure you supplied the correct password value to self.config.password

config = {

'host': 'localhost',

'database': db, # we got db as an argument

'user': 'root',

'password': 'root',

'port': '3306' # change the port to match the port your SQL server is running on

}

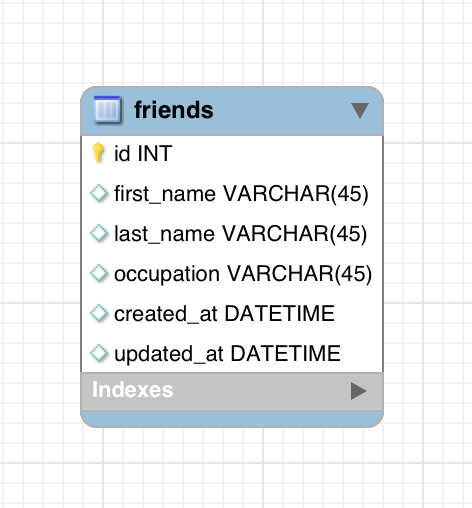
# MySQL Communication

You are all familiar with writing queries. You should be proud! Writing queries is the hard part. Displaying the information to the page is fairly easy. You will still be writing queries the exact same way that you already have been, but we will be adding a few additional steps so that we can communicate with the database through PHP or Python.

Let's create a very simple webpage where we can track all of our friends! We will implement the functionality to add friends and view friends through the website. All friends will be stored in the database. For this application, we won't have login/registration.

First, create your database using the following SQL file:  [friends.sql](http://s3.amazonaws.com/General_V88/boomyeah/company_209/chapter_3011/handouts/chapter3011_3907_friends.sql)

And consider the following ERD:



Now let's start creating our sample application.

As always we will start with our server.py file. Let's first make the app work with hard coded data and then we will add the back end.

For your file structure make sure that, in addition to your server.py file, you have the mysqlconnection.py file and a templates folder with an index.html file.

Call your project "friends".

### /friends/server.py

from flask import Flask, request, redirect, render\_template, session, flash

from mysqlconnection import MySQLConnector

app = Flask(\_\_name\_\_)

mysql = MySQLConnector(app,'friendsdb')

@app.route('/')

*def* index():

return render\_template('index.html')

@app.route('/friends', methods=['POST'])

*def* create():

# add a friend to the database!

return redirect('/')

app.run(debug=True)

And now for our Index.html

### /friends/templates/index.html

<html>

<head>

<title>Friends</title>

</head>

<body>

<h1>These are all my friends!</h1>

<p>First Name: Jay</p>

<p>Last Name: Patel</p>

<p>Occupation: Instructor</p>

<hr>

<p>First Name: Jimmy</p>

<p>Last Name: Jun</p>

<p>Occupation: Instructor</p>

<hr>

<h2>Add a Friend</h2>

<form action='/friends' method='POST'>

<label for="first\_name">First Name:<input type="text" name="first\_name" id="first\_name"></label>

<label for="last\_name">Last Name:<input type="text" name="last\_name" id="last\_name"></label>

<label for="occupation">Occupation:<input type="text" name="occupation" id="occupation"></label>

<input type="submit" value="Add">

</form>

</body>

</html>

Note that for now, we have just hard coded our friends and submitting our form will just redirect us to the page we are already on.

### Fetching Friends

First, let's add a few friends into our database manually through MySQL Workbench. This will allow us to have some seed data that we can use to create and test functionality to display all friends from the database.

INSERT INTO friends (first\_name, last\_name, occupation, created\_at, updated\_at)

VALUES ("Jay", "Patel", "Instructor", NOW(), NOW());

INSERT INTO friends (first\_name, last\_name, occupation, created\_at, updated\_at)

VALUES ("Jimmy", "Jun", "Instructor", NOW(), NOW());

Test that you successfully added the two friends into the database using a select query in MySQL Workbench:

SELECT \* FROM friends; # Should display two friends

Now that you have added the friends let's try to query the database for them from our base route. Update your route like this:

@app.route('/')

*def* index():

friends = mysql.query\_db("SELECT \* FROM friends")

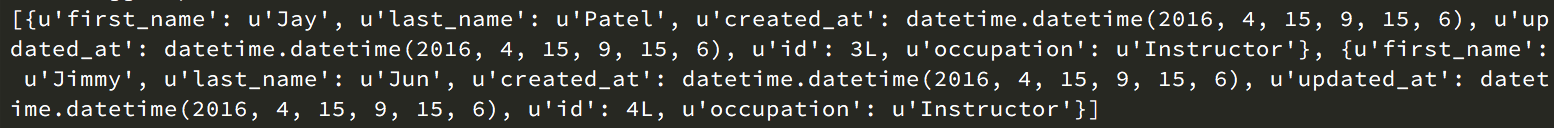
print friends

return render\_template('index.html')

Notice how we use the MySQL object's query\_db function and pass in the query that we would like to run.

#### Observing Results

Make sure to refresh your browser and check that you see your **friends printed out in the terminal.**Take a moment to examine how our query returned data. It should look something like this:



The fetch method returns us an **array of dictionaries where each dictionary represents a row in the table**. For each dictionary, the **keys correspond to the column names and the values correspond to that entries value**.

There are a couple things that may look new:

1. the 'u' that precede each key or value that is a string. The 'u' indicates a Unicode string that is inherently different from the simple string type. Unicode is used to represent a wider variety of languages and symbols. (Ex. **á** or **ü**). For our purposes, they work exactly as strings
2. the 'L' append to each value for the 'id' field in each dictionary. This represents a 'long' which is a type to store numbers in memory. The 'long' type is more accurate but will behave exactly as an int.

#### Displaying Results

Now let's display the data on the index.html page. As you remember we can pass data to the view using named parameters in the render\_template function. Let's go ahead and pass the whole friends dictionary to the template:

@app.route('/')

*def* index():

query = "SELECT \* FROM friends" # define your query

friends = mysql.query\_db(query) # run query with query\_db()

return render\_template('index.html', all\_friends=friends) # pass data to our template

Again, this is just passing the query into the query\_db function. Now on our index.html page, we will have to display all of the friends using embedded python.

**Replace the hardcoded friends on index.html** with the following to display friends using embedded python in the template:

<!-- we can use the line below to see all of our data in our template -->

{{ all\_friends }}

<h1>These are all my friends!</h1>

<!-- with all the data we can then construct a more structured output -->

{% for friend in all\_friends %}

<p>ID: {{ friend['id'] }}</p>

<p>First Name: {{ friend['first\_name'] }}</p>

<p>Last Name: {{ friend['last\_name'] }}</p>

<p>Occupation: {{ friend['occupation'] }}</p>

<hr>

{% endfor %}

Now if you test out your application again, you will see all of your friends being loaded from the database! Try adding a friend manually through MySQL Workbench and test that the page loads it.

#### Inserting Data into Queries

Now if wanted a route to fetch a specific user we would have to modify the query of fetching all the users slightly:

@app.route('/friends/<friend\_id>')

*def* show(friend\_id):

# Write query to select specific user by id. At every point where

# we want to insert data, we write ":" and variable name.

query = "SELECT \* FROM friends WHERE id = :specific\_id"

# Then define a dictionary with key that matches :variable\_name in query.

data = {'specific\_id': friend\_id}

# Run query with inserted data.

friends = mysql.query\_db(query, data)

# Friends should be a list with a single object,

# so we pass the value at [0] to our template under alias one\_friend.

return render\_template('index.html', one\_friend=friends[0])

Notice how we are grabbing the friend\_id from the URL and passing that into the function as the same name as the URL parameter, friend\_id.

### Inserting Records

Now that we can retrieve friends we have to add the functionality that allows us to create a friend! Let's focus in on our '/friends' route which will handle adding a friend and then redirect to the base route which already retrieves all friends and displays them.

First, let's make sure that we are properly getting all of the data from the form:

@app.route('/friends', methods=['POST'])

*def* create():

print request.form['first\_name']

print request.form['last\_name']

print request.form['occupation']

# add a friend to the database!

return redirect('/')

Now try adding a friend and see if the information gets printed to the terminal. Inputting "Andrew Lee Instructor" should print the following to the terminal:

Andrew

Lee

Instructor

Now that we know that the information successfully gets to the server, let's insert it into the database using our MySQL object:

@app.route('/friends', methods=['POST'])

*def* create():

# Write query as a string. Notice how we have multiple values

# we want to insert into our query.

query = "INSERT INTO friends (first\_name, last\_name, occupation, created\_at, updated\_at) VALUES (:first\_name, :last\_name, :occupation, NOW(), NOW())"

# We'll then create a dictionary of data from the POST data received.

data = {

'first\_name': request.form['first\_name'],

'last\_name': request.form['last\_name'],

'occupation': request.form['occupation']

}

# Run query, with dictionary values injected into the query.

mysql.query\_db(query, data)

return redirect('/')

Notice again, we just pass the query into the MySQL object's query\_db function.

### Updating Records

Say we wanted to update a specific record, we could create a form that would submit to the following route:

@app.route('/update\_friend/<friend\_id>', methods=['POST'])

*def* update(friend\_id):

query = "UPDATE friends

SET first\_name = :first\_name, last\_name = :last\_name, occupation = :occupation

WHERE id = :id"

data = {

'first\_name': request.form['first\_name'],

'last\_name': request.form['last\_name'],

'occupation': request.form['occupation'],

'id': friend\_id

}

mysql.query\_db(query, data)

return redirect('/')

Nothing fancy, just posting to a route with a URL parameter - friend\_id and using that friend\_id in our update query.

### Deleting Records

@app.route('/remove\_friend/<friend\_id>', methods=['POST'])

*def* delete(friend\_id):

query = "DELETE FROM friends WHERE id = :id"

data = {'id': friend\_id}

mysql.query\_db(query, data)

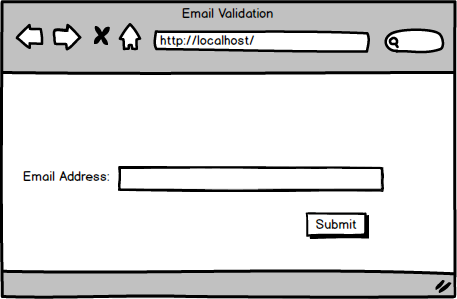
return redirect('/')

You can figure this one out and now you know how to interact with the database using the mysqlconnection.py file!

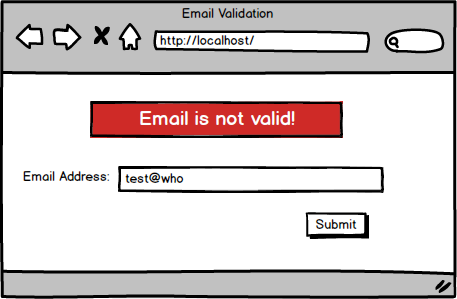
HW #2

## Assignment: Email Validation with DB

Create an application that asks a user to enter an email address.



1. If the email address is not valid, have a notification "Email is not valid!" to display on the homepage.



1. Once a valid email address is entered, save to the database the email address the user entered.
2. On the success page, display all the email addresses entered along with the date and the time (e.g. June 24th, 2013, 6:00 PM) when the email addresses were entered
3. (Bonus) add a new feature that allows the user to delete an email record on the success page.

Your success page should look like this once done.



With the bonus feature added, your app should look a little different than above. :)

HW#3

## Assignment: Full Friends

Create an application that will perform all the CRUD operations on the friends resource:

* In Index.html, each friend should have an "edit" button that will take the user to the '/friends/<id>/edit' URL which should display the edit page for that particular user
  + The edit page form should send a POST request to '/friends/<id>' which will actually update the user in the database with the new inputs
* In Index.html, each friend should have a "delete" button (part of a form) that should POST to '/friends/<id>/delete'
  + This route should delete the user from the database

At this point, you should have 2 pages and 5 routes which should be handled by the routes and methods below

|  |  |  |  |
| --- | --- | --- | --- |
| Method | URL | Route Handler Function | Purpose |
| GET | '/' | index() | Display all of the friends on the index.html page |
| POST | '/friends' | create() | Handle the add friend form submit and create the friend in the DB |
| GET | '/friends/<id>/edit' | edit(id) | Display the edit friend page for the particular friend |
| POST | '/friends/<id>' | update(id) | Handle the edit friend form submit and update the friend in the DB |
| POST | '/friends/<id>/delete' | destroy(id) | Delete the friend from the DB |

Make sure that your application uses the structure above. A **Friend**should have First Name, Last Name, Email, and a time stamp when the friend was created.

The following wireframe will help get you started:

