QPL Diamond Database

Michael Walsh ([mpwalsh@mit.edu](mailto:mpwalsh@mit.edu))

June 14, 2014

[diamondbase.mit.edu](http://diamondbase.mit.edu)

ssh –p 51522 diamondbase@diamondbase.mit.edu (password: diamond)

My goal here is to give an overview of how it *works*. It turns out these things can get fairly complicated quickly. As such, this does not fully explain how everything works. In fact, there are a few parts where it is oversimplified to the point of *almost* being inaccurate – but it saves us from starting a whole new discussion about something that is not useful to a basic understanding.

This dives in more than is necessary to simply be a user, however hopefully you will know how to use it after reading. If you are strapped for time, just read the first section.

# Website Structure

Below is an attempt to capture the organizational structure of the website. This is somewhat difficult to do, because in an effort to make navigation easier, there are more connections than what is listed below. Nonetheless, it provides a very nice and intuitive way to think about the workflow. All blue text is clickable, so you can sort the tables by the various headers.

Home (diamondbase.mit.edu/DB/)

├── Samples

│   └── Pieces

│ └── Actions

│ └── General Data

│ └── Local Data

│ └── Attachments

├── New Action

│   ├── Action Type 1

│   ├── Action Type 2

│   └── ...

├── New Sample

├── About

└── Admin

Auth App

Sample\_Database App

   ├── Groups

   ├── Users

   ├── Action Types

   ├── Data Types

   └── Designs

Read further to understand what is meant by these different apps and to understand how "Samples", "Pieces", "Actions" etc. relate to one another. Currently, these are the available actions:

* Implantation
* Anneal
* Experiment
* Bake
* Etch
* Acid Clean
* Create\*

Create is a unique Action, because you are not allowed to assign it to anything. In fact, it won't even appear in the New Action menu. It is automatically generated when a new piece is created. The purpose is to mark the date it was created and allow files to be attached describing the piece. For example, you should always attach a map file (image and maybe even a MATLAB vector file) to patterned pieces. This should also be the home of white light images. Put anything that describes the piece here.

General Data, Local Data and Attachments all have a certain Data Type associated with them. The only difference between them is who their parents can be (i.e. General can only have a parent that is a piece, Local can only have a parent that is General etc.) and their location setup. General is expected to be an image of some kind, so it has an xmin, xmax, ymin, ymax. Local will be attached to that coordinate plane somewhere, so it has x and y. And Attachment is expected to be some kind of analysis on the Local data. It is possible that you will want the Local to be some analysis. In that case, put something arbitrary in for x, y but make a note of it in the notes. Currently, the available data types are:

* NV
* Resonance
* Analysis File
* Map File
* Spectrum
* White Light
* Confocal
* Resonant Excitation

Both data types and action types can be created in the admin. You should give a considerable amount of thought to additions to either of these, because changing them will cause errors. You also don't have the permissions required to do this – let me know if you need to edit/delete a type for some reason so damage control can pursue. Just in case it isn't clear, the fields that you enumerate (separated by lines) are what will appear on the site. See Django–Database interface to understand how it works.

You can also add/edit Designs. This is a property of every piece that is created, as such deleting will also cause some issues. So, you don't have the permissions to delete items. Again, talk to me if something comes up.

That pretty much covers it. Read further to see how things talk to each other and how the database works.

# Server File Structure

/Users/diamondbase/Code/QPG/Diamond\_Database/

├── DB

│   └── sample\_db.sqlite3

├── QPL\_Diamonds

│   ├── \_\_init\_\_.py

│   ├── settings.py

│   ├── urls.py

│   ├── views.py

├── SSH\_Interface

│   ├── proxy.py

├── apache

│   └── wsgi.py

├── custom\_middleware

│   ├── \_\_init\_\_.py

│   └── middleware.py

├── foundation

│   ├── \_\_init\_\_.py

│   ├── static

│   │   └── foundation

│   │   ├── css

│   │   ├── img

│   │   └── js

│   ├── templatetags

│   │   ├── \_\_init\_\_.py

│   │   └── foundation\_tags.py

│   └── urls.py

├── manage.py

├── sample\_database

│   ├── \_\_init\_\_.py

│   ├── admin.py

│   ├── forms.py

│   ├── models.py

│   ├── templates

│   │   ├── registration

│   │   │   ├── base.html

│   │   │   ├── base\_site.html

│   │   │   └── login.html

│   │   └── sample\_database

│   │   ├── Modal\_form.html

│   │   ├── base.html

│   │   ├── details.html

│   │   ├── edit.html

│   │   ├── init.html

│   │   ├── new\_action.html

│   │   ├── new\_piece.html

│   │   ├── new\_sample.html

│   │   ├── piece.html

│   │   └── sample.html

│   ├── templatetags

│   │   ├── \_\_init\_\_.py

│   │   └── extras.py

│   ├── tests.py

│   ├── urls.py

│   ├── views.py

│   └── widgets.py

├── scripts

└── static

├── admin

│   ├── css

│   ├── img

│   └── js

└── foundation

├── css

└── js

Clearly, there are many files that talk to each other. But, no fear...it is not as complicated as it looks. Django is very modular. It divides code up into the root "project," in this case everything within scope of Diamond\_Database, and various "apps." One of these apps is considered the root of the site. It is easy to spot, because it will have a settings.py file which helps configure the server (shown in red, above). There are two more apps, foundation and sample\_database shown in green and blue. Foundation was an app downloaded from [here](http://foundation.zurb.com/). The purpose of this is to interface with the frontend of the site (what you see when you are in your browser). Sample\_database is the primary app in this project. We will discuss this more later.

The remaining folders and files that work directly with the web server are DB and static. These are pretty straightforward – DB is simply a wrapper for the SQLite3 database, the only database in this project. The folder, static, keeps track of the *project's* static content to be served. At this point, you may be wondering where the media folder is (most web servers have a static and media folder). We will get to that soon!

Finally, that leaves us with SSH\_Interface, apache, custom\_middleware, scripts and manage.py.

* SSH\_Interface is simply an organizational folder to contain python scripts that access and can modify the database in a safer way than having a bare python shell running.
* apache is a security folder that contains a strange file wsgi.py. This will be discussed briefly when we talk about the Apache server.
* custom\_middleware is a useful folder that organizes some middleware objects. We will discuss how those middleware objects work in Web Server http Trajectory. In this instance, it has to do with security.
* scripts is a folder that organizes python scripts that you write to talk to the DB
* Finally, manage.py is a Django file that does everything you could want. It allows you to open a shell configured to talk to the database, it was the original file that built the database (talked about later), it is also capable of running a tiny python webserver for debugging purposes.

In the end there are two ways to communicate (technically three because it isn't finished yet).

* Web application – http
* SSH – once finished SSH into a user that has only one privilege - execute the proxy.py file
* SSH as root – dangerous because you could end up deleting the entire database.

Behind the scenes, Houston is mounted and only accessible by the apache user in /Volumes/Houston.

# Django Organization

I struggled to determine if this section should precede the section and the full trajectory of an http request. However, we will take that as a black box for now. Being a database-based application, we will start there. However, before diving in, one quick (but important) note. You may have noticed an \_\_init\_\_.py file in a few folders above. These are completely empty python files that simply tell Django's engine that it needs to keep track of this folder because there are more python files inside. It is pretty fascinating how Django actually does this, but certainly outside the scope of this report.

The first step when making a Django app is to define your apps. Each app will have a few different files:

* models.py – tells Django how to construct the database and interface with it.
* forms.py – tells Django how to build forms in html, and how to analyze the filled out form to correspond to your model object.
* views.py – this is the home of the code that is executed when you visit a certain URL. There is a function for each URL that you offer. The output of the function is the html that is sent to the client.
* urls.py – the file that tells Django which function in views.py to call based on the incoming URL.
* tests.py – We didn't use it, but it would allow us to test different parts of the site systematically to find errors.
* templates – Contains Django html files that have some script built in that Django can interpret. These usually help out the view functions in preparing "nicer" html.

Not every app must have all of these, and some will have a few extras. It depends how organized you want to be. For example we could have defined my forms.py in models.py – but that is less intuitive.

## The Root App Settings (settings.py)

The root app, QPL\_Diamonds (sorry to anyone using this with something other than diamonds...), has an additional file: settings.py. Yes, it is also missing quite a few files that you see in other apps – this is because it has no need for a database, so there is no models.py, and it has no need for html templates or forms. It is nothing more than the first stop for an http request, but we will get there later. Nonetheless, the settings file sets many variables that Django's engine interprets (a few listed below):

* DEBUG – will return python errors to the client. When off, you will either see a typical 404 page or a server error page.
* INSTALLED\_APPS – a list of installed apps (in this case, sample\_database and foundation)
* MIDDLEWARE\_CLASSES – a list of installed middleware (we will get there in the next section)
* ROOT\_URLCONF – tells Django which application will serve as the root. That will be the first urls.py file that is called. In this case, it is QPL\_Diamonds.
* MEDIA\_URL – the URL that corresponds to media.
* MEDIA\_ROOT – the physical location of media files.
* STATIC\_URL – same as media; this one must match what Apache things the static URL is! This is because when loading a page, your browser will send many requests for static material without you knowing. If Django tells your browser the URL is one thing, but Apache is listening for something else, you will have a communication error and no content will be served because it doesn't know what to do with the URL.
* STATIC\_ROOT – same as media, with the same requirement as STATIC\_URL and apache.

It is also responsible for setting a few important security variables that we don't want the outside world to see (as such apache should never have permission to touch this file). As well as less important things like time-zone, languages, etc.

## The sample\_database Database (models.py)

This is the app that constructs the database. The goal is to represent all of the information as compactly as you can. There is no need to store something twice. The models.py file contains many classes (each corresponding to a database table), for example:

**class** Piece(models.Model):

slug = models.SlugField(help\_text="Appears in URLs") #Not used yet (but populated automatically upon object creation)

sample = models.ForeignKey(Sample)

name = models.CharField(max\_length=50)

design = models.ForeignKey(Design)

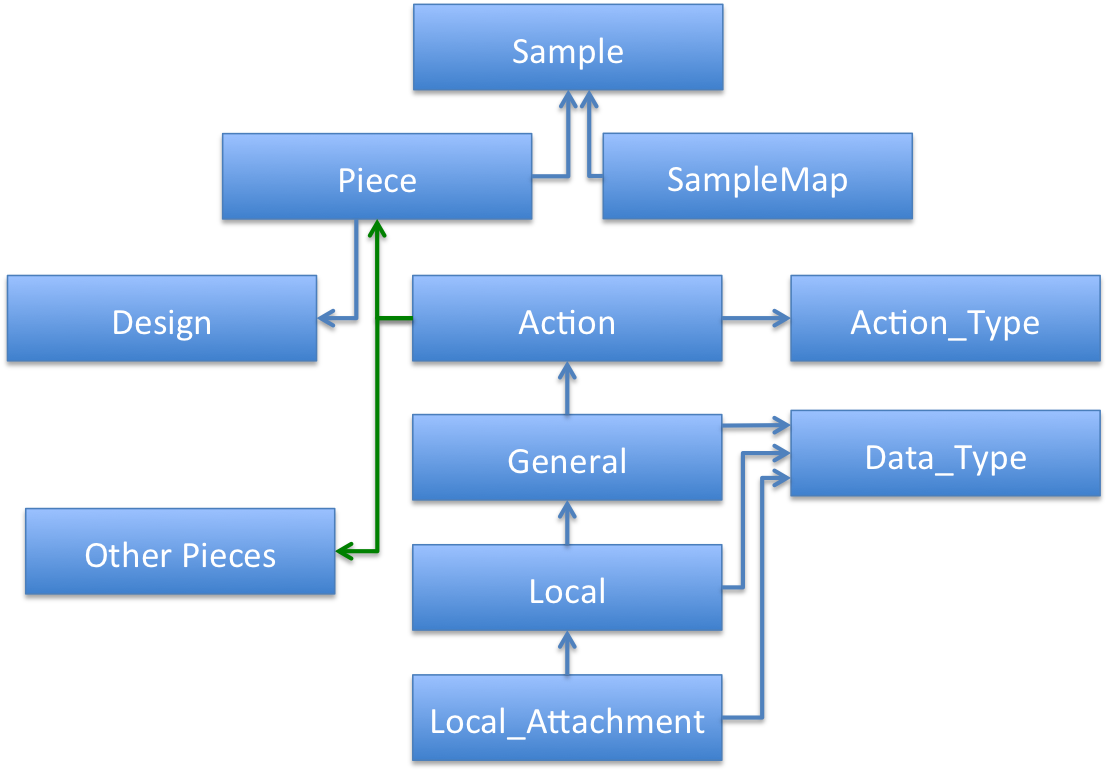
gone = models.BooleanField(default=False)

parent = models.ForeignKey('Piece',blank=True,null=True) #If a piece breaks

date\_created = models.DateTimeField(auto\_now\_add=True)

Each property of the class corresponds to a table column. Upon saving, one additional property is added: id. The models.\*Field tell Django what kind of data is going to be stored. Upon submitting a form, it will first verify that it is in the proper format, and if not, return an error. The models.ForeignKey maps to another object in the database (another class). In this usage, one of them is being mapped to the same type of object (perfectly fine); the other is mapped to Sample (another class in the file).

Below gives you an idea of the structure (only showing ForeignKey/ManyToManyField). The only difference is a Foreign Key usually represents a single parent. ManyToMany allows for as many relationships as necessary. A piece can have many actions, and an action can occur on many pieces. However, a piece corresponds to one sample. All of these arrows can be traversed in both directions in the database.



Blue arrows represent a ForeignKey and green arrows represent ManyToMany.

## The sample\_database HTTP response (views.py)

For every page you want to represent in your application, a corresponding function must govern it. The most basic function would be the function to your home page most likely – no additional inputs (other than the request). A simple construction could look as follows:

**def** index(request):

**if** request.method=='POST':

form = Form\_Class(request) #From forms.py

**if** form.is\_valid():

form.save()

**else**:

**return** some\_error

#Init by getting all the action\_types from the database, and setting the title

action\_types=Action\_Type.objects.all()

title='Welcome to QPL Diamond Database'

#Get data

samples = Sample.objects.all()

#Prepare Table

table\_head=...

table\_data=...

table = Table(table\_head,table\_data)

#Prepare Form

form = Form\_Class() #From forms.py

#Prepare html

context=RequestContext(request,{'action\_types':action\_types,

'title':title,

'table':table,

'form':form})

**return** render(request, 'template to use',context)

This is a severely truncated version of what is happening! First we determine if it is a POST (the result of a form submission). If it is, simply check the form's validity and save. Once saved, we still need to construct the page (since we aren't returning an error; we could have also constructed a page saying successful submission...but we chose not to here). We prepare a few variables by querying the database and doing some formatting (wrapped up in "..." and Table()). Next, a context object is constructed that defines a dictionary mapping string names to variables. This dictionary is what allows our template to be generic. Somewhere in the template, there is some html that surrounds title – which will replace itself with the string that was defined in the function. That html is then returned to the user.

## The Admin and Auth App

Yep, I have left this out of the discussion so far. The file structure for the admin app is located in the Django source directory. There is one additional file in all the other apps that hasn't been discussed yet – the admin.py file. This simply tells Django which objects in models.py that you want the admin app to use (called registering). This file also allows you to specify the desired behavior, but that is out of the scope of this explanation.

In the root urls.py file (QPL\_Diamonds), there is a regex that sends Django to the admin app ([diamondbase.mit.edu/admin/](http://diamondbase.mit.edu/admin/)). As you would expect, it takes the user to an administration page, where you can edit items that you have registered. Turns out you will see yet another app that is running called Auth. The admin app will list all apps that have more than one class in models.py registered.

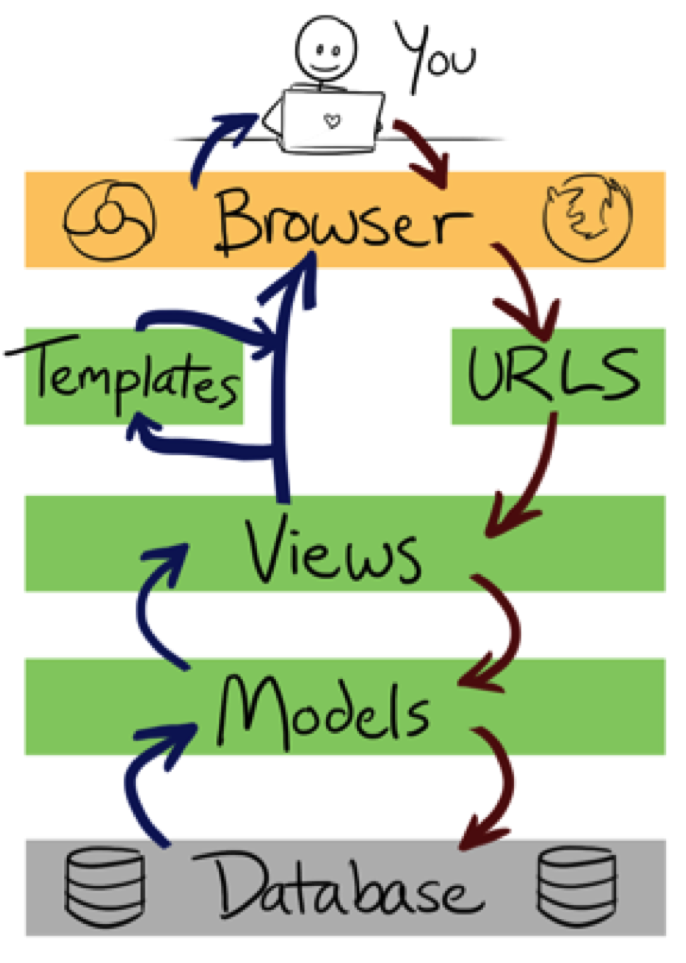
* Auth
  + Groups
  + Users
* Sample\_Database
  + Action Types
  + Data Types
  + Designs
  + Pieces
  + Samples

It is very intuitive to use – but not quite as organized as the Sample\_Database app ([diamondbase.mit.edu/DB/](http://diamondbase.mit.edu/DB/)) – we could modify admin.py more to make it easier; but that can be quite the time sink.

The Auth app keeps track of the users and their permissions. For our site, everyone has permissions to do everything with sample\_database objects and most permissions for auth. In the end – it doesn't *really* matter because we never check permissions before changing something in the sample\_database database (probably poor choice in name for the app). Instead, we just make sure that you actually do have an active account to access any part of the website – talked about more later. There are a few key things here – everyone should have "Active" and "Staff Status" checked. This means you will be able to log into the site and use the admin app. The superuser status designates all permissions without explicitly assigning them – not necessary.

A group is an object that keeps a list of allowed permissions. This makes it easy to just assign a user to a group instead of explicitly granting them.

## Django http Summary



An oversimplified picture of what is happening. You can imagine forms being grouped with models. Courtesy of <http://littlegreenriver.com/weblog/tag/python-2/>

From a high level, we can track your browser's http request entering Django. Django's engine sends the URL to the root app's urls.py. This file maps a URL using an extended [Regular Expression](http://en.wikipedia.org/wiki/Regular_expression) (regex) scheme. The extension allows the definition of variables within the URL. Once the correct regex is found it will either send it to a function in views.py (with the variables it extracted) or send it to another app's urls.py for further searching. In this case let's say the URL in question is diamondbase.mit.edu/DB/new\_sample/.

1. The black box of Apache sends /DB/edit/Action/ID27/ to Django
2. QPL\_Diamonds (root app) urls.py matches the line: url(r'^DB/', include('sample\_database.urls')) where it found the regex '^DB/' meaning begins with DB/ and can have anything after that.
3. QPL\_Diamond urls.py sends the part of the URL after the match: /edit/Action/ID27/ to the specified location. In this case, sample\_database's urls.py
4. sample\_database's urls.py matches the line: url(r'^edit/(?P<type>([^\/]+))/ID(?P<id>([^\/]+))/$',views.edit). P<blah> describes a variable. So, the regex identifies the variable "type" to be "Action" and the variable "id" to be "27". The next argument tells it to call its own views.py file and execute the function edit with the arguments it pulled out.
5. sample\_database's views.py's edit(request,type,id) is called.
6. This function accesses the classes in models.py, which in turn access the database (discussed in Django-Database Interface).
7. Because this particular URL has a form associated with it, it also accesses some classes in forms.py to get the html code to pass on.
8. A certain template is given a set of variables that it can use to paste in appropriate places within html code, and returned to views.py.
9. Finally, views.py sends the returned html back to the client and it is rendered in your browser. Tah-Dah!

# Django-Database Interface

There is a good chance the following sub-section belongs somewhere else...but it is important enough that I didn't want it bogged down in Django's organization. Also, hopefully very soon you will never have to worry about this because the website will take care of it for you!

## Action and Data Fields

Before jumping the gun, there is one "unusual" use of Django's fields. The goal was to make this as scalable and general as possible. Because most databases don't support dynamic fields, it is necessary to keep track of field names and field values that must be set on the fly. This is necessary for Actions and Data, which is why Action\_Type and Data\_Type exist. In these objects, field\_names are stored (delimited by new lines – '\r\n'). Before rendering any website page or return a value with SSH proxy, python will first separate items by this delimiter. If some strange errors crop up and are related to action or data fields, there is a good chance something isn't properly delimited.

The actually Action and Data objects have a property called fields. The values are stored in the same order as specified in the associated Action\_Type/Data\_Type. The website takes care of this automatically, but when communicating with the SSH proxy it is up to the user to make sure the data is in the right order! Django is not smart enough to understand the difference between xmin and laser power – they are both just numbers to it.

This is why changing a Data\_Type or Action\_Type after it has been in use is problematic. The Data or Action fields will no longer be consistent with the associated Type field\_names.

## Terminal Communication

Django provides a function called manage.py. If run with the first argument as "shell" it will enter an interactive python shell that is just about ready to talk to the database:

Diamond\_Database diamondbase$ python manage.py shell

First, you must import the correct models. This makes it so you never have to interface directly with the database; rather, you get to you use the convenient model layer defined in models.py. This can be done in one easy line:

>>> from sample\_database.models import \*

>>> ...Code till you are satisfied...

>>> exit()

This tells python (rather Django) that you want to import every object in your models file that is in your app called sample\_database. Now we can query, edit, and delete items in the database. We will only discuss a small fraction of the functionality of the python interface. For more, [Django's documentation](https://docs.djangoproject.com/en/1.7/) is absolutely outstanding. Google isn't so bad either.

Of course, anything that is possible in the command line is certainly possible in a script. If you are interested in doing so, just append the following to the top to initialize the Django engine. The datetime is optional – but that is how date objects are stored in the database, so if you want to have fun with them too, you should include it. Please save your scripts in the scripts folder

**import** sys, os, datetime

sys.path.append('/Users/diamondbase/Code/QPG/Diamond\_Database')

os.environ.setdefault("DJANGO\_SETTINGS\_MODULE", "QPL\_Diamonds.settings")

**from** django.db **import** models

**from** sample\_database.models **import** \*

#### Finding Items

Each item in the database belongs to a particular model. That model, stored as a class object in python, has quite a few methods to do all kinds of things. When trying to find items, you are going to want to access the "objects" of that model's class. The objects options returns another class – the object manager. This class also has tons of methods, but we will focus on three: all(), filter() and get().

The all() method will retrieve every object that is stored in the database, for example we will try and get all the Action\_Types:

>>> Action\_Type.objects.all()

[<Action\_Type: Implantation (3 fields)>, <Action\_Type: Anneal (2 fields)>, <Action\_Type: Experiment (0 fields)>, <Action\_Type: Bake (4 fields)>, <Action\_Type: Etch (7 fields)>, <Action\_Type: Acid Clean (3 fields)>, <Action\_Type: Created (0 fields)>]

You see a list returned with all of the action types that have been created! It is as easy as that. However, this can be shocking if you end up returning every data item for example. There are already nearly 1000 of them. You have the option to filter your search query by using the filter() method. You can filter on any property of your model class. Note that for data and action objects field names are NOT a property of the model. Remember? The field values are simply stored in one property: field and delimited by newlines. Their name is stored in the associated Type object. We will examine an example in the next sub-sub section. Let's try and retrieve only the Samples that are located at MIT.

>>> Sample.objects.filter(location="MIT")

[<Sample: Silicon 1 (MIT)>, <Sample: Silicon 2 (MIT)>, <Sample: Silicon 3 (MIT)>, <Sample: NbNetV1alu (MIT)>]

Voila. We have 4 samples at MIT (that are also in the database).

More frequently, you will want to get all the attachments that are of type "Resonance". First, we must consider what that means. It means that we want Local\_Attachment objects to be returned. But we want the ones that have a Data\_Type of Resonance; this is stored under the property "data\_type". It is the name field that is assigned "Resonance". This sounds pretty complicated all of the sudden, but it is not. We just need to understand that to follow a relationship (more on this below) we use double underscore. As an aside, no property can be labeled with a double underscore to avoid confusion.

>>> Local\_Attachment.objects.filter(data\_type\_\_name="Resonance")

The above will return a lot of objects. We can search by more than one parameter, just separated by commas. So, let us filter one more step to include only the ones with the General\_Data's notes of "C3R5". Again, you have to think how to trace back to General Data. Local\_Attachment has a parent property pointing to a Local\_Data which has a parent property pointing to a General\_Data which has the notes property. We will follow it with a query that only looks for resonances on piece EGM5a (notes that the Action object has a ManyToMany field called pieces...behaves just like a ForeignKey such as parent does in).

>>> Local\_Attachment.objects.filter(data\_type\_\_name="Resonance",parent\_\_parent\_\_notes="C3R5")

>>> Local\_Attachment.objects.filter(data\_type\_\_name="Resonance",parent\_\_parent\_\_parent\_\_pieces\_\_name="EGM5a")

One thing to note now that we can return these list structures. They aren't actually lists! They are a Django object called QuerySet. Because of this, they won't behave exactly as you would imagine. If you leave it in the QuerySet form and try to edit and save changes...it will not work. You must "evaluate" the QuerySet item by assigning it to a variable. For example, if I have a query set returned called items:

>>> len(items)

5

>>> items[3].foo

103

>>> items[3].foo = 203

>>> items[3].save()

>>> items[3].foo

103 <-- did not save!

>>> temp = items[3]

>>> temp.id

103

>>> temp.id = 203

>>> temp.save()

>>> temp.id

203

Finally, we can avoid a query set by using the get() method. Everything is exactly the same, only you need to make sure that you filter material only returns one object. This is why it is nice to use "id" or "name" (although if you name more than one thing the same you will have a problem).

>>> EGM5a = Piece.objects.get(name="EGM5a")

>>> EGM5b = Piece.objects.get(id=3) <-- assuming EGM5b has an id of 3

#### Organizing/Editing Items

The hard part is done. We now have our object either by using the get() method, or by assigning a query set item to a variable. As demonstrated above, any property in the model can be accessed. The only one you may not have seen coming was the id...but that is always set automatically when saving a new item. The two other methods we need to introduce here are save and delete (we won't cover how to create objects).

Any time you make a change to a model's property, you must save it before you exit, as seen a couple examples up.

To delete something, instead of calling save(), call delete() on the object. It is that easy – yes, it is that easy!! The minute you hit enter, it is game over. There is **no going back** to get that item. It is gone, and yes, that can be scary and certainly is dangerous.

#### Following Item Relationships

Most of this was already covered as well, but we will go over a few subtleties. Django keeps track of forward and backward relations ships. Let's say we are dealing with a foreign key in General Data (there are 2; one for the parent, one for the data\_type). It is super easy to follow forward, but what if I have a Data\_Type object, and I want to know the children.

>>> res = Data\_Type.objects.get(name="Resonance")

>>> ALL\_data\_res = res.data\_set.all() <-- General,Local,Attachment

Hopefully you are scratching your head a bit on that example. It is unusual that three different models would be returned. But, there is a subtlety that I haven't mentioned yet. These three models are actually a superclass of a model called Data. Pretty nifty.

Bottom line is that to go backwards and see all the children of an object. You type Object.child\_object\_lower\_case\_set.all(). Of course, you don't have to use all...you can use filter or get or any method of an object manager.

There is no difference between ForeignKey and ManyToMany relationships. The only difference that I can see is that a ManyToMany property returns a QuerySet, and a ForeignKey returns the actual object.

# Apache

When an http request comes in, Apache needs to know what to do with it. Because the Django application takes care of authentication, apache is configured to forward all requests. It does this by taking the root URL – diamondbase.mit.edu and looking at the next part of the path – diamondbase.mit.edu/url\_path. An alias is set up so Apache will rout the url\_path to the actual path on the hard drive. There are three such aliases:

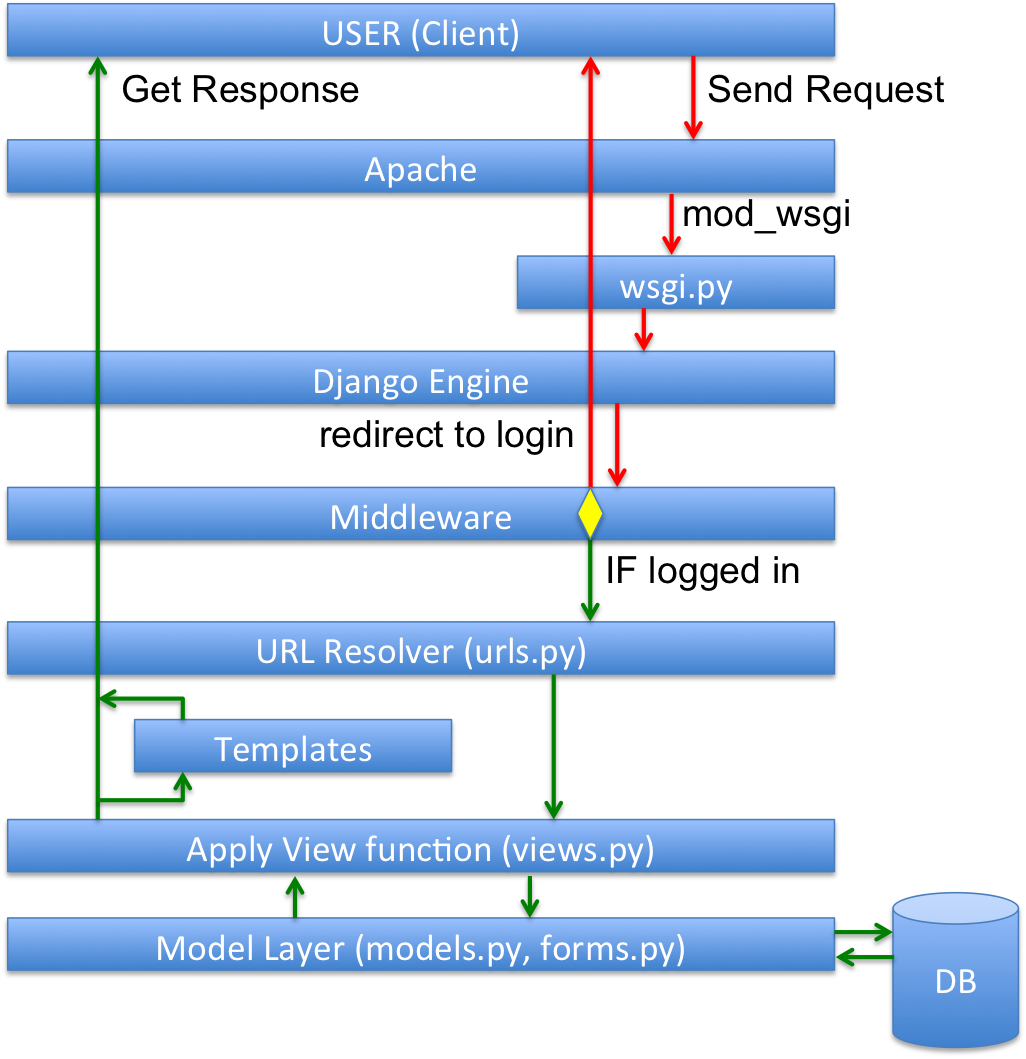
* Alias /
  + Any root requests (anything other than static and media will follow this alias). This is where the WSGI module comes into play. The alias for anything other than static and media is sent to that wsgi.py file that was mentioned earlier in the apache folder. This provides Apache the appropriate handle to the Django engine so it can run the necessary files.
* Alias /static
  + Maps /static -> /Users/diamondbase/Code/QPG/Diamond\_Database/static and serves which ever file is specified. No authentication is necessary. Example:
  + <http://diamondbase.mit.edu/static/admin/img/icon-yes.gif> is translated to /Users/diamondbase/Code/QPG/Diamond\_Database/static/admin/img/icon-yes.gif and indeed you will see the little green circle with a check mark. This should not be a security problem, because there are no important/confidential files living there. If you try and get fancy by using the '..' to go back a folder to access important files, you will see a forbidden error, since apache does not have permission to leave the static folder.
* Alias /media
  + A module called XSendFile tells apache that before serving media files, it must make sure the user is authenticated with the Django application. To do this, it will actually rout this request through the wsgi.py file as well.

The Apache config file and all other necessary files won't be discussed here. I will mention that they are kept in /etc/apache2/. The most important one is httpd.conf, which tells apache what is has permission to do, what user it should run as and how to interface with Django (all files must be edited as root). The log files for apache are kept /var/log/apache2/. You can view apache errors (not Django nor python) and view an access log.

Apache runs under User "apache" and group "\_www". This user has very limited permissions as a security precaution. The biggest security window is that it has access to Houston. This is necessary since this is where the media files are served. Don't worry - in normal conditions, an un-authenticated user will not be able to access or do anything. Even the recent vulnerability known as [Heartbleed](http://en.wikipedia.org/wiki/Heartbleed) wouldn't have been a problem (security isn't handled by Apache, but rather Django). Furthermore, we sit behind MIT's firewall; so, many attacks will be stopped there.

# Web Server http Trajectory (Security)

There are only a few more details to add to the Django summary. First, we left out one rather large component of what Django is doing. It is called middleware. All requests go through middleware before ever seeing the URL resolvers (urls.py). This is where security enters the picture. Most is done automatically through built in middleware, but you will notice in the file tree that there is a custom\_middleware folder. This simply provides a wrapper for the entire site. It makes sure that the user issuing the request is a valid user in the Auth app. If not, it will drop your request, and it will never even see the URL resolver. The middleware then replies with a redirect to a login page.



Red indicates any traffic. Green represents authenticated traffic only. The yellow diamond represents a decision.

# SSH Communication

Finally, a brief description of the SSH communication. The reason for adding this method was to allow MATLAB code in the lab to communicate easily (in theory it could send POST requests – but that would be a bit more difficult). Instead, it just calls a python function called proxy.py. This function parses the inputs and sends them to the right helping function. Hopefully this makes the database interface as easy as it can while maintaining a degree of security (to avoid accidental deletions).

For best/easiest use, create a private/public key pair (either choose no passphrase, or have your keychain remember it) and register your public key with diamondbase's authenticated\_keys (located in ~/.ssh). Then you can run ssh without entering a password:

ssh –p 51522 diamondbase@diamondbase.mit.edu python proxy.py [commands – see below]

Syntax:

get\_samples [n most recent] -> ID|sampleName (one per line)

qet\_diamonds sampleID -> ID|diamondName (one per line)

get\_experimentIDs diamondID -> ID|date (one per line)

get\_data\_types -> ID|dataType (one per line)

new\_experiment diamondID1,ID2,ID3 "notes" -> experimentID

add\_general experimentID DataTypeID "file\_path|raw\_path|xmin|xmax|ymin|ymax|field1|field2|...|notes" -> generalID

add\_local generalID DataTypeID "file\_path|raw\_path|x|y|field1|field2|...|notes" -> localID

add\_attachment localID DataTypeID "file\_path|raw\_path|field1|field2|...|notes" -> attachID

query "django query code" -> field\_value

Notes:

Example django query code: Data\_Type.objects.get(name='NV').id [MUST USE SINGLE QUOTES]

The last field in an add\_data needs to be url encoded (utf-8)

The global DELIM is used as a delimiter on a given line.

FILE PATH MUST BE WITHIN SCOPE OF SERVER if you want to be able to download it.