A *Simple* Guide to The Discovery Senate Website

By Matthew Byrd

I. Introduction

I am going to be honest with you: I've never made a website before, I didn't know how to make a website until now, and I've never released any of my code to the public before. Bearing that in mind, please know that the website is built to be intuitive and functional, to someone who has never made a database, looked at a database, or otherwise has had no affiliation with a database, it may not make sense the way I've implemented certain ideas. That's okay, it's why I'm writing this now so that people aren't going to destroy the website and all the data the senate collects later on. You can learn anything. So, while reading this, if you don't understand something, re-read until you do.

II. Django Basics

Hopefully, if you've taken on the burden of this website, you already know python. If not, stop reading here and go learn it, it's simple, intuitive, and will make the rest of this much easier. Good, now that we are all on the same page let's start looking at Django, the framework this website is based on.

Dhssenate.net is built on Django 1.11, and is split into multiple different 'apps' (I've made the website so each sect of Senate is self-contained for the most part, each one of these sects are apps). These apps each contain their own urls (what you use to access them with), their own models (what the database is made out of) and their own views (what is delivered to the user).

Models can be thought of as the blueprint to the website, the python code that makes them lays out what the model will contain. The models are defined by classes in each models.py file. For example:

```
from django.db import models # imports the models parent class

class example_model(models.Model):

    name = models.CharField(max_length=200)

    description = models.TextField()
```

The above code builds a model with two fields; one named ‘name’, and one named ‘description.’ The name field can only have a string of at most 200 characters, while the text field doesn’t require a max_length, allowing it to have as many as it wants. Once you run:

```
python manage.py makemigrations
```

Then

```
python manage.py migrate
```

The new models will be added to the database that is created when you start a django project. You can imagine the Database as a sort of excel document, each model represents a sheet, each model instance (whenever you build something with the model’s blueprint) represents the rows, and the columns are the fields (name and description). However, to visually see the data we have to register the model in the file admin.py with the following code:

```
from django.contrib import admin

from.models import example_model
```

```
@admin.register(example_model)
class example_model_admin(admin.ModelAdmin):
    model = example_model
```

The above code allows for the example_model we created to be viewed and modified on the /admin page, in which the entire database can be shown if we so desire (we don't!). Finally, if we want to display a page to a user with data from the example model, we need to create a few things: firstly, we need a function in views.py, we need a url in urls.py to call that function, and we need a template to give to an end user.

The views.py function can range from super simple, such as if you are just rendering a page without wanting a user to give you information, to complex and 'loopy' (lots of for loops). For example:

```
from django.shortcuts import render
def example_render (request): # request is a required argument here! It represents the user!
    render(request, 'template_name.html')
```

Simply, this passes a request variable to the html template, which may seem a little strange, passing a variable to a html document, but when django complies it actually has 'tags' that are codded in the template that are typed in Python, but django converts it to HTML, it's all quite nice. The request variable is rather important, it contains information about if the user is logged in, what page the user is on, what groups of authorization the user belongs to, and user fields (yes! The built-in user of django is a model as well!). The urls are rather easy, as long as you can brush up

some on regular expression (regular expression is what allows the computer to match patterns, it's how we verify emails and in this case how we are going to verify the url pattern exists).

```
from django.conf.urls import url

from . import views

urls = [

        url(r'^$', views.example_render(), name='example_url)

]
```

The url() function does magic behind the scene to register the url with the website, the r'^$' just makes the parent url (say if the app was named example_app and we register it in the root urls.py file as /example, r'^$; will make it websitename.extension/example), and the name is what allows us to reference the url in templates, so that we can redirect to other parts of our website without having to type in the actual url. urls is the required name for the list, it's what django looks for in the file.

Finally, when a user accesses the website.com/example, it will need to deliver a html document to the users end so that the website actually exists and has things for the user to view. This is rather simple, in a folder called templates in the app folder we have to create a file titled 'template_name.html' to match what we named the template in the views function. This can be a simple html document.

```
<!DOCTYPE html>

<html>

<head>
```

```
<title>Template</title>

</head>

<body>

{% if request.user_isauthenticated %}

<p> the user is logged in! </p>

{% else %}

<p> the user is not logged in </p>

{% endif %}

</body>

</html>
```

This document tests with the variable that is passed if a user is logged in or not, changing the webpage contents based on the result. This is why Django's template engine is so powerful, it allows easy access to database management in a familiar language: python (though there are many other frameworks that do this for many other languages, it's just I like python so I made it with Django). And that's the basics of Django, take a moment, make sure you understand what the relationship between the models, database, views, and urls file is, and when you're ready let's look at the Discovery Database.

III. The Discovery Database

The Database has a lot of simple parts, but also contains some complex parts. Let's jump into the simple. The Senate app (remember apps are just sections of the website) only has a single

model, it represents the senators. Each model instance contains the name of a senator, a photo of them, their email, and if they are current or not. These models are loaded into the senate template and it displays the active members loading in each senator model instance. The constitution portion of the senate app is individual and has to be updated manually, I didn't think it really made much sense to make the constitution into a separate model in the database because it is updated infrequently, and there are not multiple versions of it. There's room for improvement here, perhaps splitting the constitution up into its parts, clauses, introduction, etc, or storing previous versions for archival reasons. Overall, the senate app is fairly self-contained and isolated, which makes it as an easy introduction.

The Community Database is perhaps the most complex. The Community models contains a few models: Game, Community Instance, Community Games, Community Game Ratings, Community Extra Ratings, Community Pacing Ratings, and Community Extra Ratings. I'm going to start by explaining why the ratings are separate: it's because validating the amount of games and game models, by ensuring that the numbers line up, is impossible with a single community survey model. This may not seem like a big deal, but it really is. So instead, the games and community models are completely separate entities, tied together by another model that serves as a sort of go-between. It makes the actual process of adding a new community somewhat tedious, but it also is an extremely robust database. The two other ratings, that being pacing and extras exist separately from game ratings as they serve their own purpose outside of the games. Each one of these has a modelForm, which creates a html form from the model. This streamlines the process of creating surveys, as they are automatically filled in with game names and descriptors, making the only job of a senator to add each community. The way in which

these are loaded in and validated, then stored back into the database is rather self-explanatory and I won't go into detail here, see the code for that.

IV. Editing Survey Data

You don't.

V. Upgrading The Website

If you are the person who decided to take up maintain the website, it must mean you have some drive to see the school improve. Excellent, you should share the same desire with the website. As long as everything goes as planned, you should have access to the Discovery Github and the Heroku account. It's rather easy once you've made changes to update the website. Run these commands from the command prompt in the folder:

```
git add –all
git commit -a -m "Put commit message here"
git push origin master
git push heroku master
```

Don't worry about pushing error code, the herkou will catch or the website just won't work until you fix it. Ensure that the environment variable debug is always off, if turned on, it will create security vulnerabilities that would jeopardize the website.