Anatomy of a Web Framework through Django

By Matt Allen

April 20, 2020

The purpose of this presentation is to describe what a web framework is through Django. The Django Project has a lot of great documentation and tutorials. Instead of attempting to repeat already great tutorials by the Django team, the current presentation will attempt to explain concepts that you may encounter when researching web frameworks. There are many links in this presentation to provide more context, and make the intended meaning clear.

The Django project's motto is "The web framework for perfectionists with deadlines." I think we all know what perfectionism and deadlines are, but what is a web framework? There are many web frameworks out there written in lots of different programming languages. We used one for R called Shiny. Some have more features than others do. There is a tradeoff between ease of use and features. At the core, a web framework should facilitate building web applications.

Two of the major Python frameworks are Django and Flask. There are others Python Web Frameworks, but these two are the ones that I personally have seen mentioned the most. Furthermore, they illustrate a full stack framework versus part of the stack (Non-full stack). Django is a full stack web framework, and Flask is part of the stack. Both being web frameworks, the most important part is being able to handle web requests and responses.

A web application ultimately is a software application used in a web browser. The major technologies that web browsers support and consume are HTML, CSS, and JavaScript. Web browsers support other technologies, but at the heart are the basics of HTML, CSS, and JavaScript. To get these resources, we navigate to URLs. This action is a Request. The two most common types of Request are a GET, and a POST. After completing a Request, the web application will make a Response, and send it back to your browser. The fundamental thing that a web application framework should handle is this Request and Response cycle. The web framework provides an abstraction over this cycle. Both Flask and Django provide a framework to map URLs to your Python methods.

Another thing that we might want a web framework to handle is saving and retrieving data from a database, and maybe even generating and updating database schemas. Some other framework would handle data access in a part stack framework like Flask. Using whatever data access provider you want, gives flexibility to plug in parts. Django being a full stack framework has the data access capability built into it. You could use a different model layer framework, but Django has a nice one built into it. This is a major difference between the two.

So far, we have covered the basics of a web framework, the rest of the presentation will describe three parts of a web framework through Django: Model, View, and Template layer. I am emphasizing the concept of a web framework, because it is a general concept and has some common design patterns and use cases around it. A common web application pattern is Model View Controller. Django supports a similar pattern namely the Model View Template pattern. In addition, Django has some common use cases built in like an Admin Application for designers to manage data. This is a very common use case in web development and supports "the perfectionist with deadlines" that is in the motto. Django is not only a full stack web framework, but also has a common use cases built into it.

The model layer in Django is in fact an Object Relational Mapping (ORM) framework. You can think of ORMs as an abstraction of SQL. It allows you to think of objects in Python versus database tables. The example polls application has a database table called Question. In Django, you can access all the questions by typing

```
>>> Question.objects.all()
```

This is equivalent to the SQL

```
select * from Questions
```

In fact, you can do all Create, Read, Update, and Delete statements (CRUD) statements, which are equivalent to SQL's Insert, Select, Update, and Delete statements. The statements are Python, so you do not even have to know SQL.

Here is an example of finding and deleting a Question:

```
>>> q = Question.objects.get(pk=1)
>>> c.delete()
```

This would be equivalent to writing something like this in SQL

```
Delete from Questions where Id = 1
```

Under the covers, the Django framework translates your Python code into SQL.

Beyond this Django also has the capability to do Migrations. Migration support is limited to a few databases out of the box. The best support is in PostgreSQL followed by MySQL and SQLite. Migrations allow you to create a Python class and have it automatically generate a SQL Schema. As in the sample application, there are Question and Choice classes. A migration would generate the SQL statements to create a Question and Choice table in your database.

The View layer in Django handles the Request and Response pattern of a Web Application. Its purpose is to route URLs to Python methods. It takes a Web Request and returns a Web Response. The response can be anything that you want to return from calling a URL. This is typically a web page, but it could also do a redirect to another page, return JSON, an error code, or whatever content you want to return. A simple example of a View is below. This view just returns the text "Hello World!"

```
from django.http import HttpResponse

def index(request):
    return HttpResponse("Hello World!")
```

The URL mapping system in Django makes for very clean looking URLs. The method above is accessed by going to a URL like http://mysite/greet. The Django routing system examines the URL and attempts to map it to a Python function. The mapping is very flexible and can use regular expressions. The URLs can include parameters for a Python method to use. For example, you could pass a name in the URL like

http://mysite/greet/Matt. Note in the example below the <str:name> maps to the parameter **name** in the method named_greet.

```
def named_greet(request, name):
    return HttpResponse("Hello " + name + " !")
```

The routing is handled in a file called urls.py, which for the examples above would look like

```
from django.urls import path
from . import views

urlpatterns = [
    # ex: http://mysite/greet/
    path('', views.index),
    # ex: http://mysite/greet/Matt/
    path('<str:name>/', views.named_greet),
   ]
```

The above example would make a trivial web site. To make reusable components to display dynamic HTML, Django provides a built in template system called Django Template Language. Templates can be composed of other templates, which makes them reusable. Templates allow you to inject bits of Python code into HTML pages, which makes pages dynamic. A simple example taken from the Django Tutorial is below.

```
<h1>{{ question.question_text }}</h1>
<ul>
{% for choice in question.choice_set.all %}
    <li>{{ choice.choice_text }}</li>
{% endfor %}
</ul>
```

The double brackets indicate that Python code is inside. In this example, an object question has an attribute question_text. The string question_text will be printed inside the <h1> tag. Below the question_text a list of choices for a question is generated from a Python for loop.

This presentation covered some general aspects of Web Frameworks and how they are implemented in Django. I have just begun using Django, but have successfully created a simple application with a PostgreSQL database. Web applications can be daunting, but Django has a lot to offer to simplify web application development. I like that it is all in Python, so it would be relatively straight forward to use all the great Statistics and Machine Learning libraries available in Python in a web application. This would allow making models we build more widely available. If your web application does not need a database or if you have another model framework that you prefer, then a simpler option for a web application would be to use Flask.