Kayak Python Web Framework

Table of Contents

Summary

Kayak is a python web framework implemented as part of learning how the Django web framework works.Currently the framework includes the basic features like to be able to implement views, linking views with urls and to render the basic html with css templates using the views.

Basics of Python Web applications

We all know that generally web applications are served to the client through the servers. So it makes sense to start exploring from the point “how the python web application interact with the servers?” For any python web application to be able to served by the servers there is a contract called WSGI(Web Server Gateway Interface).

WSGI is a specification which serves as an interface between the python web application and the servers.So both the server and the python web application must implement the WSGI interface. From the application side this is the starting point. At application side WSGI interface we get the requests from the server in a predefined format. And the clients which interact with the servers (generally web browsers) also expects a predefined format of response. So as an application(framework) developer we need to understand the request and the client expecting response formats to process the request and send back the response.We need to specify the path to WSGI interface of our application to the server, so that it can interact with it. Basically it happens by configuring it in config files of server or passing as command line arguments while running the server.

How Django Works

Now let’s explore the Django specifically. We know that the Django is a python web framework, which serves as a toolkit and having defined structure to implement the web applications in python.Basically the Django is minimizing the overhead for us by implementing the common web application needs.So most of the web application common needs like talking to servers, databases, rendering templates are taken care by the framework to facilitate the user to concentrate on the specific application requirements he/she want to implement.

The Django framework combined with user implementation serves as one single web application. So the starting point in Django to serve the requests lies in the file ***“****wsgi.py”* in generated Django project. The server must know the path to this file so that it can talk to the Django application. So i basically started my exploration from “*wsgi.py” to understand how Django works.* In this file “*application = get_wsgi_application()” specifies the path to wsgi interface, which is implemented in Django source code at “core/handlers/wsgi.py” as “WSGIHandler” class “__call__” method.* Which takes “*environ”* and “*start_response*” as two args, which will be replaced with the server inputs a request as python dictionary and a call back function respectively.

The callback function “*start_response”* from server *should be called with a status string(Ex: ‘200 OK’*) and list of response headers(Ex: Content-Type: ‘text/html’). And after finished with request processing and came up with the response data, wsgi interface should return an iterator(Ex: list) with the response data in binary format. While processing the request Django uses its toolkit to talk to databases, processing templates, middlewares and doing all the other stuff. So that’s how the Django processes a request and sends a response back to server.

WSGI

WSGI interface is implemented as function *“app”* in *“core/wsgi.py”,* Which will call the request handler function called “*handler*”. Which will take the server request and callback functions as arguments.

Request and Response Cycle

The “*handler*” function implemented in “core/*base.py*” processes the server request by creating HTTPRequest class object from the server request dict. And calls the *"view_resolver”* function with the requested url path as argument to resolve the view function to be called. And view function will be called with HTTPRequest class object as argument and it will return an HTTPResponse class object. From this response object we will form the response headers and status string to pass as arguments to the server callback function. And finally an iterator with binary encoded response body is returned.

URLS

Urls are handled in “*core/urls.py*”, which contains “*urlpattens*” an array of tuples of url regular expression pattern and its corresponding view function. And this module also includes a “*view_resolver.py”* .

Views

Views can be implemented in module “*core/views.py*”. Which should be linked with url patterns in “core/urls.py”. View should take HTTPRequest class object as an argument and return HTTPResponse class object. And if we want to return html as response, then there is an utility function called “*render”* implemented in “core/utils.py”, which can be used to send html as response.To use the “render” function with HTML template, template should be placed in “*templates/*” folder. The “*render*” function takes http status code, relative path to the html template, and the context dict to use to replace the variables in template.

Templates

Basic HTML Templates with css included can be used to render the html content. Templates should be placed in “/*templates*” folder. And “render” function in “core/utils.py” replaces the passed context with the variables in the template and returns an HTTPResponse class object.

References

http://lucumr.pocoo.org/2007/5/21/getting-started-with-wsgi/
https://medium.com/zeitcode/django-middlewares-and-the-request-response-cycle-fcbf8efb903f