**Django Middleware - servicing the request and response processes**

**Introduction:**

The Django middleware provides the user with a mechanism to manipulate the raw HttpRequest and HttpResponse objects. The HttpRequest object can be manipulated by the middleware classes before they are captured by the views.py functions. The HttpResponse object can be manipulated after the views.py function has returned the object, but before it leaves the web server. There are a few cut and dried middleware classes that are available for usage in any django project and these are part of the django framework. The middleware classes to use can be configured in the django settings.py file. The specific config variable to set is the “MIDDLEWARE” (“MIDDLEWARE\_CLASSES” for old versions of django), which is a tuple of classes. The tuple of classes are called in top-down sequence for the manipulation of HttpRequest object. To manipulate the HttpResponse object, the tuple of classes are executed in bottom-top sequence.

**Available Middleware Classes:**

Each middleware class is represented in the settings.py file as a string in the 'MIDDLEWARE' list or tuple (depending on the version of Django you are using) varaible. From here on, we will refer to it (variable named 'MIDDLEWARE') as a list of classes. The pre-defined classes available in a standard Django installation are as follows:

'django.middleware.security.SecurityMiddleware',

'django.contrib.sessions.middleware.SessionMiddleware',

'django.middleware.common.CommonMiddleware',

'django.middleware.csrf.CsrfViewMiddleware',

'django.contrib.auth.middleware.AuthenticationMiddleware',

'django.contrib.messages.middleware.MessageMiddleware',

'django.middleware.clickjacking.XframeOptionsMiddleware'

As you can see, the predefined classes deal with Security, Session handling, Authentication mechanism, CSRF handling and some Messaging system. We will take a look at all these classes in a short while. Before we do that, let's note some of the features of the middleware framework.

**Points to Note Regarding Django Middleware:**

1) It is not necessary to use any middleware class in a Django project. However, using the out of the box middleware classes do provide the user with a lot of handy methods to do important things like session handling, authentication and CSRF manipulation. So, they are normally used in Django projects.

2) You can create your own middleware classes by creating a middleware factory, that takes a callable called 'get\_response' and returns a middleware. The middleware itself is a callable that takes a request object and returns a response object (very much like a view function).

3) Alternatively, you may create a middleware by defining a class and implementing the \_\_call\_\_ function to make its objects callable. This way, it is pretty neat and you can put your business logic in the \_\_call\_\_ function so that whenever called, it will handle the logic you have implemented.

Given below are stub implementations of the middleware as discussed in points 2 and 3 above.

*def my\_middleware(get\_response):*

***# This is an implementation of middleware as discussed in point #2 above.***

*def middleware(request):*

*# Code to be executed before the view function is called.*

*response = get\_response(request)*

*# Code to be executed after the view is called.*

*return response*

*return middleware*

*class MyMiddleware:*

***# This is an implementation of middleware as discussed in point #3 above.***

*def \_\_init\_\_(self, get\_response):*

*self.get\_response = get\_response*

*def \_\_call\_\_(self, request):*

*# Code to be executed before the view function is called.*

*response = self.get\_response(request)*

*# Code to be executed after the view function is called.*

*return response*

The “get\_response” callable used in the code above may be the view function itself or it might be the next middleware in the chain. The middleware being implemented doesn't need to know what exactly it is, but it only needs to keep in mind that it would be something that immediately succeeds the current middleware.

4) The sequence in which the middlewares are listed in settings.py (in the MIDDLEWARE list) do matter. Some middleware may be dependent on one of the earlier middlewares, so it cannot be listed before the middleware on which it depends. Django uses each middleware in the sequence in which they are listed, and hence this sequence is important. For example, a middleware that handles sessions may be called before a middleware that authenticates users. Without sessions, authentication would be useless, and hence the session middleware must be listed before the authentication middleware.

**A Brief Look at Some of the Available Middlewares in Django:**

Here, we will be discussing the following middlewares:

django.middleware.security.SecurityMiddleware

django.contrib.sessions.middleware.SessionMiddleware

django.middleware.common.CommonMiddleware

django.middleware.csrf.CsrfViewMiddleware

django.contrib.auth.middleware.AuthenticationMiddleware

SecurityMiddleware: The security middleware is responsible for a series of security enhancements over the request/response cycle. It makes HTTPS request/response possible in Django and prevents Man-in-the-Middle (MITM) attacks. Each enhancement can be independently switched on or off in the Django settings file. To see the available enhancement parameters, you should refer to the Django middleware documentation.

SessionMiddleware: The session middleware comes after the security middleware, and it is responsible for the session support provided by Django. Django allows you to store and retrieve arbitrary chunks of data on a per-site-visitor basis. This allows Django to implement sessions for each user on each site. The data is stored on the server side and it implements the sending and receiving of cookies in an abstract way. Mostly, you would be using database backed sessions, but there are some cases in which cache backed sessions would be better suited. This is especially so for sites that need a very fast session handling. For cache backed session, you would need to configure caching on Django, and this is done via the CACHE variable in settings.py. For details on how Django sessions are handled, please refer to the official Django documentation at <https://docs.djangoproject.com/en/2.2/topics/http/sessions/>

CommonMiddleware: The CommonMiddleware is responsible for various features like specific user-agent blocking, URL rewriting, content length setting for streaming responses, etc. This middleware is a conglomeration of various important features for each of which a special middleware component is not necessary.

CsrfViewMiddleware: This middleware handles Cross Site Request Forgery (CSRF) by adding a hidden field in every form that has the method attribute 'POST' (basically forms that get submitted by HTTP Post requests). Actually , the programmer has to add the form field 'csrfmiddlewaretoken' in the form as a hidden field. Once the request is made from the client side, the CsrfViewMiddleware has to ascertain that the value sent is correct by checking the request header field 'csrftoken'.

AuthenticationMiddleware: This middleware is responsible for authentication of users and it adds the 'user' attribute to every HttpRequest object. In case the user is not logged in, the value of the 'user' attribute is set to 'AnonymousUser', and in case of logged in users, the value is set to an instance of the 'User' model. To check if a user is logged in or not, one can use the 'is\_authenticated' attribute of the request.user object like so:

*if request.user.is\_authenticated:*

*# do something*

*else:*

*# show the user some message specifying the user is not logged in.*

Again, for more details regarding the AuthenticationMiddleware and the Django authentication process in general, please refer to the official Django documentation at [https://docs.djangoproject.com/en/2.2/topics/auth/default/#auth-web-requests](https://docs.djangoproject.com/en/2.2/topics/auth/default/" \l "auth-web-requests)

**Use Cases of Django Middleware**:

Here we will be looking at a few cases where a middleware could be used to solve a problem. Of course we cannot create an exhaustive list of use cases, but we can surely look at some of the typical cases where we can use middlewares to implement features that cannot be done better otherwise.

Case #1 : Let us suppose we have a web/mobile app that sells some products. Now, the seller of these products/services wants to implement a coupon discount system such that a buyer can apply a coupon on the price of an item/service and get a discount. The price of the commodity is dependent on several factors like the geographical location of the buyer, the age of the buyer, the activity of the buyer on previous sessions on the website, and so on. A single view function doesn't take care of all buyers, but rather, there are multiple view functions for various groups of buyers based on conditions like the ones mentioned above. To apply a coupon for each buyer, the programmer would need to apply logic in each of these view functions and that affects maintainability of the website. So how do we solve this problem?

The best method to solve this problem is to apply the coupon related logic in a response middleware (a middleware that handles HttpResponse objects). Since each view function eventually passes the price attribute of a product/service to the middleware chain, we can create a middleware and put it at the top of the chain to apply the logic to the prices computed by the different view functions. That way, we will have only a single place where the coupon logic is implemented and hence it would be easily maintainable later on.

Case #2 : Let us suppose we have a website (or a mobile app backend) that requires each request to all views to contain a certain parameter. Requests without the given parameter would be considered invalid, and hence there is a need to check each request for the given parameter before the request reaches its corresponding view function. One way to solve this problem is to create a decorator that scans each request to check for the parameter and return a boolean value depending on whether or not the parameter exists in the request. However, implementing it in this manner would require the programmer to use this decorator on every view function she creates. That is clumsy since a new developer might not know about this rule and she may create a view function for some purpose without using the decorator.

A better solution to this problem is to create a request middleware that scans every request for the parameter's existence in the request and pass only those requests that contain the desired parameter. That way, the checking code will be in one place and it gets applied to all requests, no matter which view is being called. A new programmer need not know the existence of this requirement, and still she would be able to write valid view functions, since the middleware scans the requests before passing it to any view function.

Case #3 : Another typical case where a middleware can be used is in the financial domain websites. For example, in a financial domain website, there is a need to convert all financial figures in terms of US dollar. Doing this in each view is a bad solution as maintainability is hampered. The programmer should do this conversion in a middleware that processes each request and converts all figures in different currencies to the standard US dollar.

**Conclusion:**

Django middleware allows the programmer to consolidate code that is repeated in all views.py functions in a single place. However, this should be done with great care since whatever happens in the middleware would affect all requests. Keeping this in mind is very important as it can adversely affect the entire app. Still, Django provides the programmer with this powerful feature so that a conscientious programmer would be able to write code that is concise and maintainable. Here, we have just scratched the tip of the proverbial iceberg, and should you want to write a middleware component, you should certainly check out the official documentation for it. This article just provides a platform to launch yourself in writing Django middleware components. Hope you have an enjoyable time writing middleware components correctly.