**(1): What does your custom middleware do, and how does it work?**

My custom middleware is called **PromoCodeMiddleware**, and its purpose is to generate and return a unique promotional **code** for a user’s first API visit, while tracking whether the user has already seen the promo. It works by intercepting requests to the /api/promo-code/ endpoint.

When a request is made to /api/promo-code/, the middleware checks for a cookie named has\_seen\_promo. If this cookie does not exist (or is false), it generates a random alphanumeric code and returns it inside a JSON response, while setting the cookie in the response to indicate that the promo has been shown. For subsequent requests where has\_seen\_promo is already set to true, it returns null for the promo code.

This approach allows the frontend to display a promo banner only once per user within a certain timeframe (30 days in this case), without needing user authentication or database storage for tracking.

**(2): What problems did you face with authentication or the API? How did you fix them?**

One major issue I encountered was with cross-site cookies being blocked when making API requests from the frontend running on a different origin (e.g., frontend on localhost:5371 and backend on 127.0.0.1:8000). Modern browsers block cookies unless the **SameSite** attribute is explicitly set to **None** and the cookie is marked as **Secure.** This caused the has\_seen\_promo cookie not to be stored in the browser, so the backend kept generating a new promo code every time, thinking it was a new user.

To fix this, I updated the **set\_cookie** method to include **samesite='None'** and **secure=True**. Additionally, I configured CORS\_ALLOW\_CREDENTIALS = True and CORS\_ALLOWED\_ORIGINS in Django settings, and ensured withCredentials: true was set in my frontend Axios requests. This allowed cookies to be accepted and sent in cross-origin requests, resolving the issue.  
  
 I also faced a problem where the **authenticate()** function failed because it wasn’t receiving the correct **request** context. Without it, session-based authentication didn’t work as expected. I fixed this by passing self.context.get('request') into the authenticate() call inside the serializer’s **validate** method, ensuring Django could properly handle the authentication backend.

**(3): Pick one of the functions in the views.py that you wrote and explain line-by-line what it does.**

I will explain the **post** method inside the **UserAPILogin** view:

Defines the post method, which handles HTTP POST requests sent to this view.

def post(self, request):

Creates a serializer instance, passing the incoming request data (the username and password) and the current request object in the context.

serializer = UserLoginSerializer(data=request.data, context={'request': request})

Checks if the submitted data passes validation logic defined in the serializer (e.g., checking credentials, authenticating the user).

if serializer.is\_valid():

Retrieves the authenticated user object that was added to validated\_data during serializer validation.

user = serializer.validated\_data['user']

If validation was successful, returns a JSON response containing serialized user information and a set of authentication tokens (refresh and access tokens). The response status is HTTP 200 OK.

return Response({  
 'user': UserSerializer(user).data,  
 'tokens': serializer.get\_tokens(serializer.validated\_data)  
 }, status=status.HTTP\_200\_OK)

If validation failed, returns a JSON response with the serializer's error messages and a status code of HTTP 400 Bad Request.

return Response(serializer.errors, status=status.HTTP\_400\_BAD\_REQUEST)

**(4️):** **If you had more time, what would you improve or what additional functionality would you add?**

If I had more time, I would improve the security and user experience of the promo code middleware by:

*1): Saving promo codes in a database table tied to user sessions or IP addresses, instead of only relying on cookies (which can be deleted or blocked).*

*2): Implement rate-limiting and CAPTCHA challenges for login requests to protect against brute-force attacks.*

*3): Add email-based registration with verification emails sent to confirm the address.*

*4): Real-time flight tracking integration using external APIs.*

*5): A real-time flight booking system with live seat availability updates.*

*6): An improved user dashboard showing booking history, status updates, and notifications.*