[AI Marketer V2] Authentication System

1. Backend Structure

The AI Marketer V2 backend is organized into several Django apps, each with a specific purpose:

- **users**: Handles user authentication, registration, and profile management
- **businesses**: Manages business profiles and settings
- **posts**: Handles social media post creation, scheduling, and management
- **promotions**: Manages marketing promotions and campaigns
- **social**: Connects businesses to social media platforms (NOT for user authentication)
- **ai**: Provides AI-powered features like image analysis and caption generation
- **sales**: Manages sales data and reporting

This modular structure separates concerns and makes the codebase more maintainable.

2. Authentication System

The authentication system is primarily contained in the `users` app. It handles user registration, login, logout, and profile management, with placeholders for features like social login and password reset.

2.1 Key Files in the `users` App

2.1.1 `models.py`

This file defines the User data structure:

```
class User(AbstractBaseUser, PermissionsMixin):
    email = models.EmailField(unique=True, db_index=True)
    name = models.CharField(max_length=255)
    role = models.CharField(max_length=20, choices=...)
    is_active = models.BooleanField(default=True)
    is_staff = models.BooleanField(default=False)
    date_joined = models.DateTimeField(auto_now_add=True)
    requires_2fa = models.BooleanField(default=False)
    secret_2fa = models.CharField(max_length=255, blank=True, null=True)
    access_token = models.CharField(max_length=512, blank=True, null=True)
```

The model uses email instead of username for login, and includes fields for 2FA support and storing access tokens.

2.1.2 `managers.py`

Contains the custom user manager for creating users:

```
class UserManager(BaseUserManager):
    def create_user(self, email, name, password=None, role=DEFAULT_ROLE,
**extra_fields):
        # Creates and returns a regular user

    def create_superuser(self, email, name, password=None, **extra_fields):
        # Creates and returns a superuser
```

This manager handles user creation, including field validation and password hashing.

2.1.3 `serializers.py`

Handles data validation and processing for authentication:

- `RegisterSerializer`: Validates registration data and creates new users
- `TraditionalLoginSerializer`: Validates email/password login
- `SocialLoginSerializer`: (Placeholder) For authenticating via social providers
- `PasskeyLoginSerializer`: (Placeholder) For biometric/security key authentication
- `TwoFactorVerificationSerializer`: Validates 2FA codes
- `ForgotPasswordSerializer`: (Placeholder) For password reset requests
- `ResetPasswordSerializer`: (Placeholder) For setting a new password

2.1.4 `views.py`

Contains API endpoints for authentication:

- `RegisterView`: Handles user registration
- `LoginView`: Processes login requests and issues JWT tokens
- `UserProfileView`: Returns authenticated user profile
- `LogoutView`: Handles user logout and token invalidation
- `ForgotPasswordView`: (Placeholder) For password reset requests
- `ResetPasswordView`: (Placeholder) For setting a new password
- `Check2FA`, `Remove2FA`, `Enable2FA`: Manage two-factor authentication

2.1.5 `urls.py`

Defines URL routes for authentication endpoints:

```
urlpatterns = [
    path('register/', RegisterView.as_view(), name='register'),
    path('login/', LoginView.as_view(), name='login'),
    path('me/', UserProfileView.as_view(), name='user-profile'),
    path('logout/', LogoutView.as_view(), name='logout'),
    path('password/forgot/', ForgotPasswordView.as_view(), name='forgot-
password'),
    path('password/reset/', ResetPasswordView.as_view(), name='reset-password'),
    path('2fa-check/', Check2FA.as_view(), name='check2fa'),
    path('2fa-remove/', Remove2FA.as_view(), name='remove2fa'),
```

```
    path('2fa-qr/', Enable2FA.as_view(), name='qr2fa'),
]
```

2.1.6 `authentication.py`

Handles JWT authentication from cookies:

```
class CustomJWTAuthentication(JWTAuthentication):
    def authenticate(self, request):
        # Get token from cookie instead of Authorization header
        token = request.COOKIES.get(settings.SIMPLE_JWT["AUTH_COOKIE"])
        if token is None:
            return None

        try:
            validated_token = self.get_validated_token(token)
            user = self.get_user(validated_token)
            return user, validated_token
        except Exception as e:
            return None
```

2.2 Authentication Flow

2.2.1 Registration Flow

1. Frontend sends user data to `/api/users/register/`
2. `RegisterView` uses `RegisterSerializer` to validate data
3. User is created with hashed password via `UserManager.create_user()`
4. Success message is returned

2.2.2 Login Flow

1. Frontend sends credentials to `/api/users/login/`
2. `LoginView` uses correct serializer based on login method
3. Credentials are validated and user is authenticated
4. JWT tokens are generated (access token in cookie, refresh token in response)
5. Success message is returned

2.2.3 Auth Verification

1. Frontend makes authenticated requests with the cookie
2. `CustomJWTAuthentication` extracts token from cookie
3. Token is validated and user is retrieved
4. View receives authenticated user in `request.user`

2.2.4 Logout Flow

1. Frontend sends logout request to `/api/users/logout/`
2. Access token cookie is cleared
3. Success message is returned

3. Project Settings

Authentication configuration is set in `config/settings.py`:

```
# JWT Settings
SIMPLE_JWT = {
    "ACCESS_TOKEN_LIFETIME": timedelta(days=1),
    "REFRESH_TOKEN_LIFETIME": timedelta(days=7),
    "ROTATE_REFRESH_TOKENS": True,
    "BLACKLIST_AFTER_ROTATION": True,
    "AUTH_HEADER_TYPES": ("Bearer",),
    "AUTH_COOKIE": "access_token",
    "AUTH_COOKIE_HTTP_ONLY": True,
    "AUTH_COOKIE_SECURE": True,
    "AUTH_COOKIE_PATH": "/",
    "AUTH_COOKIE_SAMESITE": "None",
}

# Authentication setup
REST_FRAMEWORK = {
    'DEFAULT_AUTHENTICATION_CLASSES': (
        "users.authentication.CustomJWTAuthentication",
    ),
}

AUTH_USER_MODEL = 'users.User'
```

This configures JWT tokens to be stored in secure HTTP-only cookies and sets the custom user model.