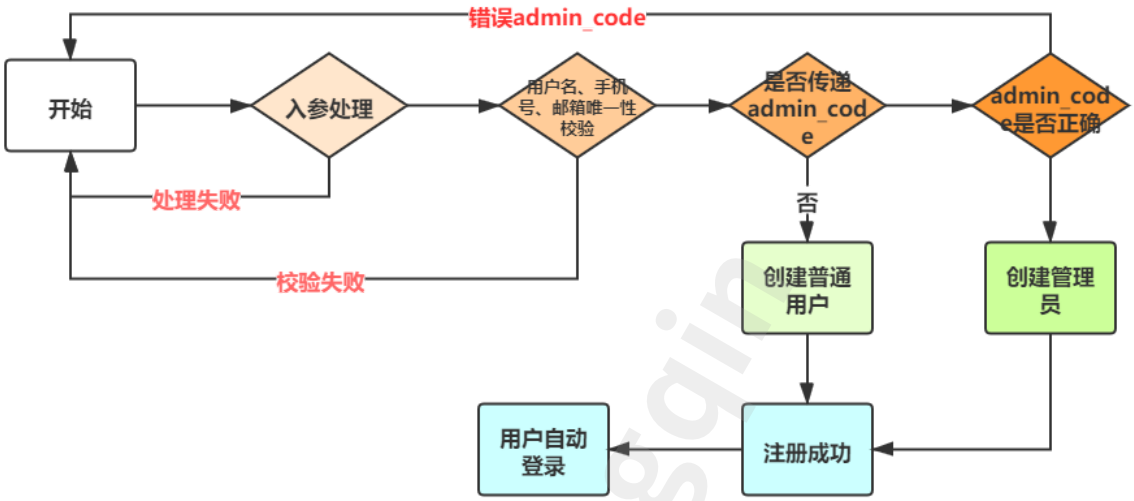
用户注册

上节课，我们完成了用户认证的基础：用户模型的创建，接下来，围绕该用户模型将用户认证机制搭建起来。

首先要实现的就是用户注册功能--创建用户

不同系统有不同的注册流程，但是大体上都差不都，都是提交信息给服务器，然后服务器判断没有问题后创建用户。结合前端，我们当前的注册流程可以参考下图：



我们可以结合REST框架帮助我们完成验证的工作

注册序列化器

```
# 注册序列器
class RegisterSerializer(serializers.ModelSerializer):
    # admin_code 不在User字段中，需要单独定义
    admin_code = serializers.CharField(default='sqtp')  # 字符串-sqtp
    class Meta:
        model = User
        fields = ['username','password','email','phone','realname','admin_code']

    # 校验入参是否合法
    def validate(self, attrs):  # attrs为入参的字典形式
        # 检查admin_code
        if 'admin_code' in attrs and attrs['admin_code']!='sqtp':
            raise ValidationError('错误的admin_code')
        return attrs

    # 重写序列化器的保存方法
    def register(self):
        in_param = self.data
        if 'admin_code' in in_param:
            in_param.pop('admin_code') # 创建用户数据不需要admin_code
            user=User.objects.create_superuser(**in_param) #创建管理员
        else:
            user=User.objects.create_user(**in_param) #创建普通用户
        return user
```

注册视图

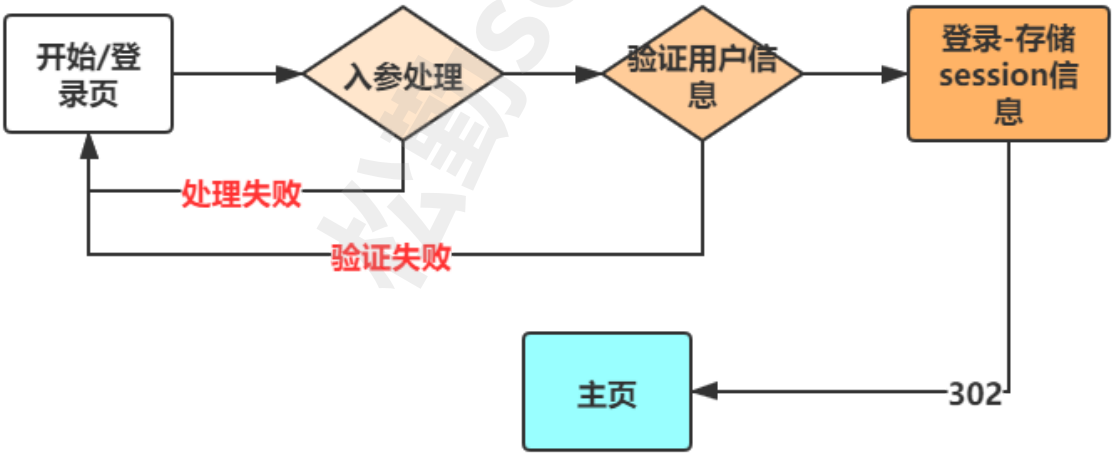
```
# 注册视图
@api_view(['POST'])
def register(request):
    # 获取注册信息--序列化器
    serializer = RegisterSerializer(data=request.data)
    if serializer.is_valid(): #自动根据模型定义来判断数据入参是否合法
        user = serializer.register()
        auth.login(request, user)
        return Response(status=status.HTTP_201_CREATED,
                        data={'msg': 'register success', 'is_admin':
user.is_superuser, 'retcode':status.HTTP_201_CREATED})
    return Response({'msg': 'error', 'retcode': status.HTTP_400_BAD_REQUEST,
'error': serializer.errors},status=status.HTTP_400_BAD_REQUEST)
```

注册URL

```
path('register/',views.register),
```

用户登录

登录流程可参考下图，同样数据验证部分可以写进序列化器中



登录序列器

与注册不同的是，登录不需要默认的序列化器校验数据，因为登录使用的是已有的数据，因此应该序列号其调用validate方法返回验证后的user对象即可

```
# 登录序列器
class LoginSerializer(serializers.ModelSerializer):
    class Meta:
        model = User
        fields = ['username', 'password']

    def validate(self, attrs):
```

```
        # 检查必填参数
        position_params = ['username', 'password']
        for param in position_params:
            if param not in attrs:
                raise ValidationError(f'缺少参数:{param}')
        # 验证用户名和密码
        user = auth.authenticate(**attrs)
        if not user:
            raise ValidationError('用户名或密码不正确')
        return user
```

登录视图

```
@api_view(['POST'])
def login(request):
    # 获取登录信息--序列化器
    serializer = LoginSerializer(data=request.data)
    user = serializer.validate(request.data)
    if user:
        auth.login(request, user)
        return Response(status=status.HTTP_302_FOUND, data={'msg': 'login
success', 'to': 'index.html'})
    return Response({'msg': 'error', 'retcode': status.HTTP_400_BAD_REQUEST,
'error': serializer.errors},status=status.HTTP_400_BAD_REQUEST)
```

登录URL

```
path('login/',views.login),
```

渲染器-异常捕获重写

异常返回的信息需要重新处理

```
from rest_framework.exceptions import ValidationError
from rest_framework.views import exception_handler  #REST框架异常处理器
# 处理异常返回
def my_exception_handler(exc,context):
    response = exception_handler(exc,context)  # 获取标准的错误响应
    if response:
        # 成功捕获到异常
        if isinstance(exc,ValidationError):
            error_msg = exc.detail[0]
        else:
            error_msg = exc
        response.data =
{'msg':'error','retcode':response.status_code,'error':str(error_msg)}

    return response
```

用户登出

这个比较简单，没有请求参数，范围对应的视图就可以完成登出流程，因此无需序列化器

登出视图

```
@api_view(['GET'])
def logout(request):
    # 当前用例处于登录状态
    if request.user.is_authenticated:
        auth.logout(request)  # 清除登录信息
    return Response(status=status.HTTP_302_FOUND, data={'msg': 'logout', 'to':
'login.html'})
```

登出URL

```
path('logout/',views.logout),
```

获取当前用户信息视图

前后端分离的系统前端部分需要同步后端的访问状态，因此构造一个请求用于检查是否登录，这样在切换页面时可以根据当前用户状态选择是否重定向到登录界面。

用于跑马灯，以及前端判断当前用户是否登录的依据

```
@api_view(['GET'])
def current_user(request):
    # 如果当前用户处于登录状态
    if request.user.is_authenticated:
        serializer = UserSerializer(request.user)
        # 返回当前用户登录信息
        return Response(serializer.data)
    else:
        return Response({'retcode': 403, 'msg': '未登录', 'to':
'login.html'},status=status.HTTP_403_FORBIDDEN)
```

用户状态校验

django默认的用户信息存储方案为session机制,有内置的装饰器可以验证当前请求是否携带用户认证信息，

```
from django.contrib.auth.decorators import login_required
@login_required
def list_user(request):
    ...

# 如果用户未登录就会被重定向到一个默认的URL(/accounts/login/) 状态码为403
```

但是如果我们用DRF，则可以用

DRF设置认证方案

可以使用 `DEFAULT_AUTHENTICATION_CLASSES` 设置全局的默认身份验证方案。比如：

```
REST_FRAMEWORK = {
    'DEFAULT_AUTHENTICATION_CLASSES': (
        'rest_framework.authentication.BasicAuthentication',
        'rest_framework.authentication.SessionAuthentication',
    )
}
```

你还可以使用基于 `APIView` 类视图的方式，在每个view或每个viewset基础上设置身份验证方案。

```
from rest_framework.authentication import SessionAuthentication,
BasicAuthentication
from rest_framework.permissions import IsAuthenticated
from rest_framework.response import Response
from rest_framework.views import APIView

class ExampleView(APIView):
    authentication_classes = (SessionAuthentication, BasicAuthentication)
    permission_classes = (IsAuthenticated,)

    def get(self, request, format=None):
        content = {
            'user': unicode(request.user),  # `django.contrib.auth.User` 实例。
            'auth': unicode(request.auth),  # None
        }
        return Response(content)
```

或者，如果你使用基于函数的视图，那就使用 `@api_view` 装饰器。

```
@api_view(['GET'])
@authentication_classes((SessionAuthentication, BasicAuthentication))
@permission_classes((IsAuthenticated,))
def example_view(request, format=None):
    content = {
        'user': unicode(request.user),  # `django.contrib.auth.User` 实例。
        'auth': unicode(request.auth),  # None
    }
    return Response(content)
```

另外，DRF的认证是在定义有权限类（permission_classes）的视图下才有作用，且权限类（permission_classes）必须要求认证用户才能访问此视图。如果没有定义权限类（permission_classes），那么也就意味着允许匿名用户的访问，自然牵涉不到认证相关的限制了。所以，一般在项目中的使用方式是在全局配置`DEFAULT_AUTHENTICATION_CLASSES`认证，然后会定义多个base views，根据不同的访问需求来继承不同的base views即可。

DRF权限方案

我们已经在DRF的认证方案中看到了如何使用权限，接下来扩展下权限的范围

只有当前项目的管理员才有读写当前项目的权限

定义权限

sqtp/permissions

```
from rest_framework import permissions


class IsOwnerOrReadOnly(permissions.BasePermission):
    """
    自定义权限只允许对象的所有者编辑它。
    """

    def has_object_permission(self, request, view, obj):
        # 读取权限允许任何请求，
        # 所以我们总是允许GET，HEAD或OPTIONS请求。
        if request.method in permissions.SAFE_METHODS:
            return True

        # 只有该snippet的所有者才允许写权限。
        return obj.owner == request.user
```

加入权限

```
class ProjectViewSet(viewsets.ModelViewSet):
    queryset = Project.objects.all()
    serializer_class = ProjectSerializer
    # 权限
    authentication_classes = (SessionAuthentication, BasicAuthentication)
    permission_classes = (IsAuthenticated,IsOwnerOrReadOnly)
```

使用非当前项目的管理员用户进行修改测试，发现被拒绝

```
{
  "msg": "error",
  "retcode": 403,
  "error": "You do not have permission to perform this action."
}
```