一、认证

身份验证是将传入的请求与一组鉴别凭据（例如请求来自的用户或与其签名的令牌）关联的机制。然后，可以使用权限和限流策略来确定是否允许请求进入。

DRF框架提供了许多现成的身份验证方案，还允许你实现自定义的方案。

在权限和限流检查发生之前，以及在执行任何其他代码之前，始终在视图的最开始处运行身份验证。也就是说，认证过程优先级最高，最先被执行。

`request.user` 属性通常设置为 `contrib.auth` 包的 `user` 类的实例。这是Django原生的做法，请参考Django教程。

`request.auth` 属性用于任何其他附加的身份验证信息，例如，它可以用来表示请求中携带的身份验证令牌。

上面两个位于request中的属性非常重要，是认证和权限机制的核心数据。

认证的机制

允许使用的认证模式一般以一个类的列表的配置形式存在。DRF会尝试使用列表中的每个类进行认证，并使用第一个成功通过的验证类的返回值设置 `request.user` 和 `request.auth` 。

如果所有认证类尝试了一遍，但还是没有通过验证。 `request.user` 将被设置为 `django.contrib.auth.models.AnonymousUser` 的实例，也就是匿名用户。 `request.auth` 也将被设置为 `None` 。可见，**DRF的认证机制依赖Django的auth框架！**

未认证用户请求过程中， `request.user` 和 `request.auth` 的值可以通过 `UNAUTHENTICATED_USER` and `UNAUTHENTICATED_TOKEN` 这两个配置项进行修改。

认证功能的核心源代码在这里：

```
# 位于request.py模块中

def _authenticate(self):
        """
        使用每个认证类的实例来验证请求
        """
        for authenticator in self.authenticators:  # 循环指定的所有认证类
            try: # 抓取异常
```

```
                # 执行认证类的authenticate方法，返回的是一个（user，auth）元组
                user_auth_tuple = authenticator.authenticate(self)
            except exceptions.APIException:  # 如果认证失败，或发生错误
                self._not_authenticated()  # 执行认证失败后的方法，用于处理后事
                raise  #继续向上抛出异常
            # 一旦某个认证类通过了验证，返回了正常的值，那么进入结束流程
            if user_auth_tuple is not None:
                self._authenticator = authenticator   # 将通过认证的认证类保存下来，备查
                self.user, self.auth = user_auth_tuple  # 将返回值赋值给request对象的user和auth属性
                return   # 结束方法，啥都不返回，因为上面一行已经将我们需要的值保存起来了。

        self._not_authenticated()  # 如果走完整个for循环都没有认证成功，进入后事处理阶段
```

配置认证方案

通过 `DEFAULT_AUTHENTICATION_CLASSES` 配置项，可以进行全局性认证方案设置，例如：

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

对于基于类的 `APIView` 视图，也可以设置视图或视图集级别的认证方案，这种粒度更细。

```
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,)

```

```
10      def get(self, request, format=None):
11          content = {
12              'user': unicode(request.user),  # `django.contrib.auth.User` instance.
13              'auth': unicode(request.auth),  # None
14          }
15          return Response(content)
```

对于使用 `@api_view` 装饰器转化来的视图，也可以指定视图级别的认证方案：

```
1   @api_view(['GET'])
2   @authentication_classes((SessionAuthentication, BasicAuthentication))
3   @permission_classes((IsAuthenticated,))
4   def example_view(request, format=None):
5       content = {
6           'user': unicode(request.user),  # `django.contrib.auth.User` instance.
7           'auth': unicode(request.auth),  # None
8       }
9       return Response(content)
```

如果想要某个视图不使用认证功能，可以设置：

```
1   authentication_classes = []
```

未认证和拒绝响应

当未经身份验证的请求被拒绝时，可能有两个不同的错误代码。

- HTTP 401 Unauthorized
- HTTP 403 Permission Denied

HTTP 401响应一般会包含一个 `WWW-Authenticate` 头部属性，用于指引用户如何认证。而HTTP 403响应则不会包含这个 `WWW-Authenticate` 属性。

具体产生哪种响应类型取决于身份认证方案。虽然同时可以使用多个身份认证方案，但最终只能使用一个方案来确定响应类型。**根据视图上设置的第一个身份认证类确定响应类型。**

请注意，当请求认证成功，但仍被拒绝执行时，无论身份验证方案如何，都将返回 `403 permission denied` 响应。

一切未通过认证的情况，都执行的是下面的源代码：

```
    # 位于request.py

    def _not_authenticated(self):
        """
        默认值为AnonymousUser 和 None.
        """
        self._authenticator = None  # request中指示认证器未None，也就是没有任何一个认证通过

        if api_settings.UNAUTHENTICATED_USER:  # 如果在settings中配置了这个参数，使用它
            self.user = api_settings.UNAUTHENTICATED_USER() # 默认叫做AnonymousUser
        else:
            self.user = None  # 否则设置user的值被设置为None

        if api_settings.UNAUTHENTICATED_TOKEN:  # 同上
            self.auth = api_settings.UNAUTHENTICATED_TOKEN() # 默认就是None
        else:
            self.auth = None
```

Apache+mod_wsgi 模式下的专用配置

注意，如果你使用 `Apache + mod_wsgi` 的方式部署项目，认证的头部字段将不会通过WSGI程序传递，它默认是通过Apache处理，而不是应用级别。

在Apache中，使用非会话的认证机制时，你需要显式地配置mode_wsgi，用于传递必须的头部信息。也就是进行如下地配置：

```
# this can go in either server config, virtual host, directory or .htaccess
WSGIPassAuthorization On
```

二、API参考

BasicAuthentication

DRF的认证模块非常简单，只提供了几个简单的认证类，主要还是依托Djangoyuans的auth认证框架。

BasicAuthentication使用HTTP基本的认证机制，通过用户名/密码的方式验证。它通常只适用于测试工作，尽量不要用于生成环境。

用户名和密码必须在HTTP报文头部，为 `Authorization` 属性提供值为 `Basic amFjazpmZWl4dWVsb3ZlMTAw` 的方式提供。其中 `Basic` 字符串是键，后面的一串乱码是通过 `base64` 库使用明文的用户名和密码计算出的密文，这一部分工作可以通过postman工具进行。

如果认证成功， `BasicAuthentication` 提供下面的属性：

- `request.user` ：设置为一个Django的 `User` 类的实例
- `request.auth` ：设置为None

未认证成功，将响应 `HTTP 401 Unauthorized` ，并携带下面的头部信息：

```
WWW-Authenticate: Basic realm="api"
```

可以看看它的核心方法authenticate的源代码：

```
    def authenticate(self, request):
        """
        使用HTTP的基本authentication属性，提供正确的用户名和密码，并返回一个user。否则返回None。
        """
        auth = get_authorization_header(request).split() # 从http报头读取密文，并分割字符串

        if not auth or auth[0].lower() != b'basic':  #如果分割后的第一部分不是以basic开头
            return None  # 认证失败

        if len(auth) == 1:  # 如果只有一个部分，说明没有提供用户名和密码部分，认证失败
            msg = _('Invalid basic header. No credentials provided.')
            raise exceptions.AuthenticationFailed(msg)
        elif len(auth) > 2: # 如果分割出了2个以上的部分，说明格式不对，空格太多，认证失败
            msg = _('Invalid basic header. Credentials string should not contain spaces.')
```

```
            raise exceptions.AuthenticationFailed(msg)
        # 只能分割成2个部分
        try:#获取第二个部分，用base64库进行解码
            auth_parts =
base64.b64decode(auth[1]).decode(HTTP_HEADER_ENCODING).partition(':')
        except (TypeError, UnicodeDecodeError, binascii.Error):
            msg = _('Invalid basic header. Credentials not correctly base64
encoded.')
            raise exceptions.AuthenticationFailed(msg)
        # 拿到明文的用户名和密码
        userid, password = auth_parts[0], auth_parts[2]
        # 进行密码比对，返回认证结果
        return self.authenticate_credentials(userid, password, request)
```

TokenAuthentication

TokenAuthentication是一种简单的基于令牌的HTTP认证。它适用于CS架构，例如普通的桌面应用程序或移动客户端。

要使用 `TokenAuthentication` 模式，你需要先配置认证类，并将 `rest_framework.authtoken` 添加到 `INSTALLED_APPS` 中，如下所示:

```
INSTALLED_APPS = (
    ...
    'rest_framework.authtoken'
)

REST_FRAMEWORK = {
    'DEFAULT_AUTHENTICATION_CLASSES': (
        ...
        'rest_framework.authentication.TokenAuthentication',
    )
}

```

注意:配置完成后，你需要运行 `python manage.py migrate` 命令，因为 `rest_framework.authtoken` 实际上是一个app或者说第三方模块，需要在数据库中生成它工作用的数据表。

接下来，你需要为你的用户创建令牌：

```
from rest_framework.authtoken.models import Token

token = Token.objects.create(user=...)
print(token.key)
```

对于要进行身份验证的客户端，令牌密钥应包含在 `authorization` HTTP头部属性中。键应该以字符串 `"token"` 作为前缀，用空格分隔两个字符串。例如：

```
Authorization: Token 9944b09199c62bcf9418ad846dd0e4bbdfc6ee4b
```

上面的工作也可以在postman中进行。

注意: 如果你想使用一个不同的头部关键字，比如 `Bearer` ，只需要简单地继承 `TokenAuthentication` 类，并设置 `keyword` 这个类变量的值为 `Bearer` 。

成功认证后 `TokenAuthentication` 提供下面的属性：

- `request.user` ： 设置为一个Django的 `User` 类的实例
- `request.auth` ： 设置为一个 `rest_framework.authtoken.models.Token` 的实例。

不成功将返回 `HTTP 401 Unauthorized` 响应，并携带下面的HTTP头部信息：

```
WWW-Authenticate: Token
```

可以使用 `curl` 命令行工具测试令牌认证API：

```
curl -X GET http://127.0.0.1:8000/api/example/ -H 'Authorization: Token
9944b09199c62bcf9418ad846dd0e4bbdfc6ee4b'
```

那么如何为用户生成令牌呢?主要有以下几种方式：

- **通过信号机制生成令牌**

如果你希望每个用户都有一个自动生成的令牌，可以简单地捕获用户的 `post_save` 信号。这个信号是Django原生为我们提供的，请参考Django教程相关内容。

```
from django.conf import settings
from django.db.models.signals import post_save
from django.dispatch import receiver
from rest_framework.authtoken.models import Token

@receiver(post_save, sender=settings.AUTH_USER_MODEL)
def create_auth_token(sender, instance=None, created=False, **kwargs):
    if created:
        Token.objects.create(user=instance)
```

如果你已经创建了一些用户，可以通过下面的方式为已创建的用户生成令牌：

```
from django.contrib.auth.models import User
from rest_framework.authtoken.models import Token

for user in User.objects.all():
    Token.objects.get_or_create(user=user)
```

- **提供获取令牌的API服务**

用户从客户端使用用户名和密码，往提供令牌服务的API发送表单或json数据。验证通过后，API将用户的令牌以json格式返回给客户端。**DRF提供了一个内置的视图 `obtain_auth_token` 用于实现这一功能！**

首先在你的URLconf中添加下面的路由:

```
from rest_framework.authtoken import views
urlpatterns += [
    path('api-token-auth/', views.obtain_auth_token),
]
```

路由的匹配字符串可以随意指定。

`obtain_auth_token` 视图会返回一个JSON响应，当用户名和密码通过验证后：

```
{ 'token' : '9944b09199c62bcf9418ad846dd0e4bbdfc6ee4b' }
```

请注意：默认情况下 `obtain_auth_token` 视图显式地使用JSON请求和响应，会忽略你在settings中关于渲染器和解析器的配置。

默认情况下，没有权限或限流机制应用于 `obtain_auth_token` 视图。如果要使用限流机制，则需要重写视图类，在其中添加 `throttle_classes` 属性。

如果你需要 `obtain_auth_token` 视图的自定义版本，可以继承 `ObtainAuthToken` 视图类，并在URL配置中使用新的子类来实现。

例如，你可以返回 `token` 值之外的其他用户信息：

```
from rest_framework.authtoken.views import ObtainAuthToken
from rest_framework.authtoken.models import Token
from rest_framework.response import Response

class CustomAuthToken(ObtainAuthToken):

    def post(self, request, *args, **kwargs):
        serializer = self.serializer_class(data=request.data,
                                           context={'request': request})
        serializer.is_valid(raise_exception=True)
        user = serializer.validated_data['user']
        token, created = Token.objects.get_or_create(user=user)
        return Response({
            'token': token.key,
            'user_id': user.pk,
            'email': user.email
        })
```

要同步修改 `urls.py`：

```
urlpatterns += [
    path('api-token-auth/', CustomAuthToken.as_view())
]
```

- **使用Admin后台生成令牌**

也可以使用Django的Admin后台手动生成令牌，如下所示：

`your_app/admin.py`：

```
from rest_framework.authtoken.admin import TokenAdmin

TokenAdmin.raw_id_fields = ('user',)
```

- **使用Django的manage.py命令**

从3.6.4版本开始，可以通过下面的命令为用户生成令牌

```
python manage.py drf_create_token <username>
```

结果如下：

```
Generated token 9944b09199c62bcf9418ad846dd0e4bbdfc6ee4b for user user1
```

如果你想为用户重新生成令牌，比如令牌已经不安全了或者泄露了的情况下，可以添加 `-r` 参数：

```
python manage.py drf_create_token -r <username>
```

SessionAuthentication

这种认证方式，使用了Django默认的会话后端，适合AJAX客户端等运行在同样会话上下文环境中的模式，这也是DRF默认的认证方式之一。

可以使用DRF提供的登录页面测试这一功能。

如果认证成功 `SessionAuthentication` 提供下面的属性

- `request.user` ： 设置为一个Django的 `User` 类的实例
- `request.auth` ： 设置为None

认证不成功将返回 `HTTP 403 Forbidden` 响应，没有额外的头部信息。

如果你正在使用类似AJAX风格的API，并采用SessionAuthentication认证，当使用不安全的HTTP方法，比如 `PUT` ，`PATCH` ，`POST` 或者 `DELETE` ，你必须确保提供了合法的CSRF 令牌。参考Django的CSRF相关章节。

DRF框架中的CSRF验证与标准Django的CSRF验证的工作方式略有不同，因为需要同时支持同一视图的会话和非会话身份验证。这意味着只有经过身份验证的请求才需要CSRF令牌，匿名请求可以在没有CSRF令牌的情况下发送。此行为不适用于登录视图，登录视图应始终应用CSRF验证。

RemoteUserAuthentication

使用Django的auth框架的认证功能。

首先你必须在你的 `AUTHENTICATION_BACKENDS` 配置中，使用 `django.contrib.auth.backends.RemoteUserBackend` (或者继承它) 。

如果认证成功， `RemoteUserAuthentication` 提供下面的属性：

- `request.user` ： 设置为一个Django的 `User` 类的实例
- `request.auth` ： 设置为None

三、自定义认证框架

自定义认证框架步骤：

1. 继承 `BaseAuthentication` 类
2. 重写 `.authenticate(self, request)` 方法，认证成功时返回一个 `(user, auth)` 二元元组，否则返回 `None` 。当然，某些情况下，你可能需要弹出一个 `AuthenticationFailed` 异常。

建议的认证机制：

- 如果未尝试身份验证，则返回None。继续进行任何其他正在使用的身份验证方案。
- 如果尝试身份验证但失败了，则引发 `AuthenticationFailed` 异常。立即返回错误响应，无论是否进行任何权限检查，也不继续进行任何其他身份验证方案。（这条可以讨论一下）

当认证失败时，你也可以重写 `.authenticate_header(self, request)` 方法，为 `WWW-Authenticate` 头部属性添加 `HTTP 401 Unauthorized` 响应。否则默认进行 `HTTP 403 Forbidden` 响应。

自定义认证类后，也可以将它作为配置参数，进行全局配置，只是在引用路径的时候，需要注意一下：

比如我们在某个app下创建一个auth模块，再写入MyAuthentication类：

```
REST_FRAMEWORK = {
    'DEFAULT_AUTHENTICATION_CLASSES': (
        ...
        'app.auth.MyAuthentication',
    )
}
```

下面的例子，对在请求头部中，将用户名保存在 'X_USERNAME'属性中的请求，进行认证：

```
from django.contrib.auth.models import User
from rest_framework import authentication
from rest_framework import exceptions

class MyAuthentication(authentication.BaseAuthentication):
    def authenticate(self, request):
        username = request.META.get('X_USERNAME') # 获取用户名信息
        if not username:
            return None
```

```
10
11          try:
12              user = User.objects.get(username=username) # 查找用户
13          except User.DoesNotExist:
14              raise exceptions.AuthenticationFailed('No such user') #如果用户
    不存在
15
16          return (user, None)
```

四、第三方模块

下面是一些可用的第三方认证模块：

- Django OAuth Toolkit

支持 OAuth 2.0，支持Python 2.7 和 Python 3.3+，文档很好，推荐使用。

```
1   pip install django-oauth-toolkit
```

需要做下面的配置：

```
1   INSTALLED_APPS = (
2       ...
3       'oauth2_provider',
4   )
5
6   REST_FRAMEWORK = {
7       'DEFAULT_AUTHENTICATION_CLASSES': (
8           'oauth2_provider.contrib.rest_framework.OAuth2Authentication',
9       )
10  }
```

- Django REST framework OAuth：`pip install djangorestframework-oauth`
- JSON Web Token Authentication
- Hawk HTTP Authentication
- HTTP Signature Authentication
- Djoser
- django-rest-auth
- django-rest-framework-social-oauth2
- django-rest-knox
- drfpasswordless