一、元数据Metadata

REST框架包含一个可配置的机制，用于确定API如何响应返回API模式或其他资源信息。`OPTIONS`方式的HTTP请求。这允许你

目前还没有任何广泛采用的约定来明确为`OPTIONS`请求返回哪种类型的响应，因此我们提供了一种特殊风格来返回一些有用的信息。

下面的示例演示了默认情况下响应的信息：

```
HTTP 200 OK
Allow: GET, POST, HEAD,  OPTIONS
Content-Type: application/json

{
    "name": "To Do List",
    "description": "List existing 'To Do' items, or create a new item.",
    "renders": [
        "application/json",
        "text/html"
    ],
    "parses": [
        "application/json",
        "application/x-www-form-urlencoded",
        "multipart/form-data"
    ],
    "actions": {
        "POST": {
            "note": {
                "type": "string",
                "required": false,
                "read_only": false,
                "label": "title",
                "max_length": 100
            }
        }
    }
}
```

可以通过`'DEFAULT_METADATA_CLASS'`设置全局性的元数据类：

```
REST_FRAMEWORK = {
'DEFAULT_METADATA_CLASS': 'rest_framework.metadata.SimpleMetadata' 3
}
```

或者通过 `metadata_class` 类属性，为每个类视图单独设置元数据:

```
class APIRoot(APIView):
    metadata_class =  APIRootMetadata

    def get(self, request, format=None):
        return Response({
            ...
        })
```

REST框架目前只有一个元数据类，也就是 `SimpleMetadata` 。

如果您对创建使用常规的“get”请求访问的模式端点有特定的要求，那么您可能会考虑使用元数据API来实现这一点。

例如，可以在视图集中使用以下附加路由来提供可链接的模式端点。

```
@action(methods=['GET'], detail=False)
def schema(self, request):
    meta = self.metadata_class()
    data = meta.determine_metadata(request, self)
    return Response(data)
```

二、自定义元数据

如果你想要自定义元数据类：

1. `BaseMetadata`
2. 实现 `determine_metadata(self, request, view)` 方法

下面的额类可以限制给 `OPTIONS` 请求返回的信息：

```
class MinimalMetadata(BaseMetadata):
    """
    Don't include field and other information for `OPTIONS` requests.
    Just return the name and description.
    """
    def determine_metadata(self, request, view):
        return {
            'name': view.get_view_name(),
        'description': view.get_view_description() 10 }
```

自定义类后，如果想使用它，不要忘了在settings中进行配置：

```
REST_FRAMEWORK = {
'DEFAULT_METADATA_CLASS': 'myproject.apps.core.MinimalMetadata' 3  }
```

格式后缀Format suGixes

在Web APIs中，经常使用 `http://example.com/api/users.json` 这种，以文件扩展名为后缀的形式，要求响应返回的内容类型。

既然url有后缀，那么Django中也必须为url路由的编写，添加相应的后缀部分。但是，向URLConf中的每个条目添加格式后缀的方式，是容易出错、机械重复、可能数量较大、且不够简洁的，因此DRF提供了一种快捷的方式，也就是 `format_suffix_patterns` 函数。

format_suGix_patterns函数

签名: format_suVix_patterns(urlpatterns, suVix_required=False, allowed=None)

该方法返回一个附加了后缀的URL路由列表。

参数说明：

- urlpatterns: 必填！初始的URL路由列表。通常是你本来写好的路由列表。
- suGix_required: 可选！一个布尔值！指示URL中的后缀是可选的还是必需的。默认为False，表示后缀在默认情况下是不需要的。
- allowed: 可选！有效格式后缀的列表或元组。如果没有提供，将使用通配符格式后缀模式。

例如：

```
from rest_framework.urlpatterns import format_suffix_patterns    # 导入方法
from blog import views

urlpatterns = [
    path('/', views.apt_root),
    path('comments/',  views.comment_list),
    path('comments/<int:pk>/',  views.comment_detail)
]

urlpatterns = format_suffix_patterns(urlpatterns, allowed=['json', 'html'])
# 对原路由进行扩展
```

如果你使用了 `format_suffix_patterns` ，你必须在对应的视图上添加 关键字参数:

`'format'`

```
@api_view(('GET', 'POST'))
def comment_list(request, format=None):    # 看这里，第二个参数
    # do stuff...
```

对于类视图，则要在每个类的方法中添加format参数：

```
class CommentList(APIView):
    def get(self, request, format=None):
        # do stuff...

    def post(self, request,  format=None):
        # do stuff...
```

可以通过settings中的 `FORMAT_SUFFIX_KWARG` 设置参数的名字，比如从 `format` 改为 `response_type` 。

另外要注意的是， `format_suffix_patterns` 不支持 `include` 路由转发。

如果你同时还使用了Django提供的 `i18n_patterns` 函数。 `format_suffix_patterns` 要做如下处理：

```
url patterns = [
    …
]

urlpatterns = i18n_patterns(
    format_suffix_patterns(urlpatterns, allowed=['json', 'html'])
)# 看这里
```

以url参数的模式指定类型

格式后缀的另一种做法是在url参数中包含请求的格式。REST框架默认情况下开启了这一功能，并且已经在可浏览API中应用，用于在不同的展示方式之间进行切换。

一个URL的例子： `http://example.com/organizations/?format=csv` 。

可以通过 `URL_FORMAT_OVERRIDE` 来设置format这个参数的名字。如果设置为None，将关闭这一功能。

HTTP头部 VS 格式后缀

在一些Web社区中，似乎有一种这样的观点：将文件扩展名放在url中，不符合RESTful的精神，应该使用 `HTTP Accept` 的属性来携带文件扩展名。

DRF官方认为这是一种误解，并给出了理由，而你怎么看？我认为应该是老板说怎么办就怎么办！

一、异常Exceptions

想象一下这么个场景，用户发送过来一个请求，DRF的视图在处理请求的过程中发生了异常。这时，你希望看到的是什么？服务器直接弹出异常，然后卡住或结束进程，等待管理员处理？客户端堵塞等待服务器响应数据？

No！No!这肯定是错误的。正确的做法是，DRF的视图抓住弹出的异常，并根据异常的类型，返回对应的响应，响应内容中要包含异常的信息。客户端仍然能够获得响应，不过内容不是期望的数据，而是发生错误的提示和信息。客户端可以根据提示，重新修改并发送请求。

DRF的视图是如何处理异常的

DRF内置了许多种异常的处理方法，并返回对应的响应。

比如下面几类：

- `APIException` 的子类
- Django的 `Http404` 异常.
- Django的 `PermissionDenied` 异常

在每种情况下，REST框架都将返回带有适当状态代码和内容类型的响应。响应的内容将包含错误原因的详细信息。

大多数错误响应将在响应正文中包含一个 `detail` 键。

例如，对于下面的请求：

```
DELETE http://api.example.com/foo/bar HTTP/1.1
Accept: application/json
```

有可能你会收到一个响应，并告诉你不允许删除请求的对象：

```
HTTP/1.1 405 Method Not  Allowed
Content-Type: application/json
Content-Length: 42

{"detail": "Method 'DELETE' not  allowed."}  # 注意detail这个键
```

验证错误的处理方式略有不同，这时会将字段名作为键包含在响应的内容中。如果验证错误不属于特定字段，则使用`NON_FIELD_ERRORS_KEY`作为键名，或者使用settings文件中`NON_FIELD_ERRORS_KEY`配置项设置的键名。

一个验证错误的例子如下：

```
HTTP/1.1 400 Bad Request
Content-Type: application/json
Content-Length: 94

{"amount": ["A valid integer is required."], "description": ["This field may not be blank."]}  # 注意返回的内容主体
```

自定义异常处理的方法

你可以自己创建处理函数来实现自定义异常处理，该函数将API视图中引发的异常转换为响应对象。这允许您控制API使用的错误响应样式。

函数必须接受两个参数，第一个是要处理的异常，第二个是包含额外上下文（如当前正在处理的视图）的字典。正常情况下，异常处理函数应该返回一个`Response`对象，如果无法处理异常，则返回None。如果处理程序返回None，则将重新引发异常，Django将返回标准的HTTP`500 'server error'`响应。

例如，你可能想要所有的错误响应都在响应主体中包含HTTP状态代码，例如：

```
HTTP/1.1 405 Method Not Allowed
Content-Type: application/json
Content-Length: 62

{"status code": 405, "detail": "Method 'DELETE' not allowed."}
```

要实现上面的功能，你可以这么做：

```python
from rest_framework.views import  exception_handler

def custom_exception_handler(exc, context):
    # Call REST framework's default exception handler first,
    # to get the standard error response.
    response = exception_handler(exc, context)

    # Now add the HTTP status code to the response.
    if response is not None:
        response.data['status_code'] =  response.status_code

    return response
```

默认的处理函数不使用上下文参数，但如果异常处理程序需要更多的信息（如当前正在处理的视图），则该参数很有用，该视图可以通过 `context['view']` 变量获得。

自定义异常处理程序写完了，还必须使用 `EXCEPTION_HANDLER` 在settings中配置异常处理程序：

```python
REST_FRAMEWORK = {
    'EXCEPTION_HANDLER': 'my_project.my_app.utils.custom_exception_handler'
#注意这个路径
}
```

如果你不做上面的配置，那么DRF默认使用的异常处理程序是：

```python
REST_FRAMEWORK = {
'EXCEPTION_HANDLER': 'rest_framework.views.exception_handler' 3
}
```

注意，异常处理程序只由引发异常生成的响应进行调用。它不会用于任何视图直接返回的响应，例如序列化类验证失败时由常规视图返回的 `HTTP_400_BAD_REQUEST` 响应。

二、API参考

除了处理异常的函数，DRF当然还为我们内置了一些常见的异常。

APIException

```
APIException()
```

签名:

所有从 `APIView` 或者 `@api_view` 视图中弹出的异常的基类。

要自定义异常，需要先继承 `APIException` ，然后设置 `status_code` 、 `default_detail` 和 `default_code` 这三个类属性。

例如，如果您的API依赖于有时可能无法访问的第三方服务，那么您可能想要自定义一个 `"503 Service Unavailable"` 的HTTP响应代码的异常。你可以这样做：

```
from rest_framework.exceptions import  APIException

class ServiceUnavailable(APIException):
    status_code = 503
    default_detail = 'Service temporarily unavailable, try again later.'
    default_code = 'service_unavailable'
```

有一些属性可用于查看API异常的状态：

The available attributes and methods are:

- `.detail` - 返回错误的文本格式描述
- `.get_codes()` - 返回错误的标识
- `.get_full_details()` - 同时返回错误标识和文本描述

大多数情况下，错误的信息都比较简单：

```
>>> print(exc.detail)
You do not have permission to perform this  action.
>>> print(exc.get_codes())
permission_denied
>>> print(exc.get_full_details())
{'message':'You do not have permission to perform this
action.','code':'permission_denied'}
```

在验证错误的时候，错误内容可能是字典格式：

```
>>> print(exc.detail)
{"name":"This field is required.","age":"A valid integer is  required."}
>>> print(exc.get_codes())
{"name":"required","age":"invalid"}
>>> print(exc.get_full_details())
{"name":{"message":"This field is  required.","code":"required"},"age":
{"message":"A valid integer is required.","code":"invalid"}}
```

ParseError

签名: `ParseError(detail=None, code=None)`

解析请求发生错误!

当访问 `request.data` 时，如果请求包含了格式错误的数据，则引发该异常。

默认情况下，这会返回 `"400 Bad Request"` 类型的响应。

AuthenticationFailed

`AuthenticationFailed(detail=None, code=None)`

签名:

认证失败!

当进入的请求包含不正确的认证身份时弹出该异常。

默认情况下返回 `"401 Unauthenticated"` ，也有可能返回 `"403 Forbidden"` ，这取决于你使用的认证模式。

NotAuthenticated

签名: `NotAuthenticated(detail=None, code=None)`

未认证。

当一个未认证的请求进行权限检查时失败了，弹出该异常。要和 `AuthenticationFailed` 区分开来。

默认情况下返回 `"401 Unauthenticated"` ，也有可能返回 `"403 Forbidden"` ，这取决于你使用的认证模式。

PermissionDenied

```
PermissionDenied(detail=None, code=None)
```

签名:

无权访问!

当一个认证过的请求在权限检查时却未通过，弹出该异常。

默认情况下返回 `"403 Forbidden"` 。

NotFound

签名: `NotFound(detail=None, code=None)`

资源未找到!

当请求的资源对象不存在的时候，弹出该异常。 等同于Django标准的 `Http404` 异常。

默认情况下返回 `"404 Not Found"` 。

MethodNotAllowed

`MethodNotAllowed(method, detail=None, code=None)`

签名:

不支持请求的方法!

视图不支持当前请求使用的HTTP方法时，弹出该异常。

默认情况下返回 `"405 Method Not Allowed"` 。

NotAcceptable

`NotAcceptable(detail=None, code=None)`

签名:

客户端想要服务器返回的内容类型不支持!

当请求头部的 `Accept` 属性定义的类型，无法被视图渲染时，弹出该异常。

默认情况下返回 `"406 Not Acceptable"` 。

UnsupportedMediaType

签名: `UnsupportedMediaType(media_type, detail=None, code=None)`

不支持的媒体类型!

当后端所有的解析器都无法处理请求数据 `request.data` 中的内容时，弹出该异常。

默认情况下返回 `"415 Unsupported Media Type"` 。

Throttled

签名: `Throttled(wait=None, detail=None, code=None)`

被限流了!

当请求被限流，不允许访问时，弹出该异常。

默认情况下返回 `"429 Too Many Requests"` 。

ValidationError

签名: `ValidationError(detail, code=None)`

验证失败!

`ValidationError` 异常和其它的 `APIException` 异常有点区别:

- `detail` 参数是必须的，不可空缺。
- `detail` 可能是一个列表、字典或者嵌套的数据结构，用于容纳错误信息。
- By convention you should import the serializers module and use a fully qualified `ValidationError` style, in order to diVerentiate it from Django's built-in validation error. For example. `raise serializers.ValidationError('This field must be an integer value.')`
- 按照惯例，您应该导入序列化类并使用完全限定的 `ValidationError` 模块，以便将其与 Django 内置的验证错误区分开来。例如要这么写： `raise serializers.ValidationError('This field must be an integer value.')` ，而不能这么写： `raise ValidationError('This field must be an integer value.')`

默认情况下，这个异常会返回 `"400 Bad Request"` 。

三、通用的错误视图

DRF提供了两个错误视图，用于提供通用的500和400错误，并返回JSON类型的响应。（Django默认的错误视图，提供的是HTML类型的响应，这可能不适合那些只使用API接口的程序）

- `rest_framework.exceptions.server_error`

返回 `500` 响应，以 `application/json` 的内容类型。

作为 `handler500` 进行设置（参考liujiangblog.com的Django教程）：

```
1   handler500  = 'rest_framework.exceptions.server_error'
```

- `rest_framework.exceptions.bad_request`

返回 `400` 响应，以 `application/json` 的内容类型。

作为 `handler400` 进行设置：

```
handler400  = 'rest_framework.exceptions.bad_request'
```

测试Testing

留待以后吧。

REST framework includes a few helper classes that extend Django's existing test framework, and improve support for making API requests.

APIRequestFactory

Extends Django's existing `RequestFactory` class.

Creating test requests

The `APIRequestFactory` class supports an almost identical API to Django's standard `RequestFactory` class. This means that the standard `.get()` , `.post()` , `.put()` , `.patch()` , `.delete()` , `.head()` and `.options()` methods are all available.

```
from rest_framework.test import  APIRequestFactory

# Using the standard RequestFactory API to create a form POST request
factory = APIRequestFactory()
request = factory.post('/notes/', {'title': 'new idea'})
```

Using the `format` argument

Methods which create a request body, such as `post` , `put` and `patch` include a `format` argument, which make it easy to generate requests using a content type other than multipart form data. For example:

```
# Create a JSON POST request
factory = APIRequestFactory()
request = factory.post('/notes/', {'title': 'new idea'},  format='json')
```

By default the available formats are `'multipart'` and `'json'` . For compatibility with Django's existing `RequestFactory` the default format is `'multipart'` .

To support a wider set of request formats, or change the default format, see the configuration section.

Explicitly encoding the request body

If you need to explicitly encode the request body, you can do so by setting the `content_type` flag. For example:

```
request = factory.post('/notes/', json.dumps({'title': 'new idea'}), content_type='application/json')
```

PUT and PATCH with form data

One diVerence worth noting between Django's `RequestFactory` and REST framework's is`APIRequestFactory` ta will be encoded for methods other than just `.post()` .

For example, using `APIRequestFactory` , you can make a form PUT request like so:

```
factory = APIRequestFactory()
request = factory.put('/notes/547/', {'title': 'remember to email  dave'})
```

Using Django's `RequestFactory` , you'd need to explicitly encode the data yourself:

```
from django.test.client import encode_multipart, RequestFactory

factory = RequestFactory()
data = {'title': 'remember to email dave'}
content = encode_multipart('BoUnDaRyStRiNg', data)
content_type = 'multipart/form-data; boundary=BoUnDaRyStRiNg'
request = factory.put('/notes/547/', content, content_type=content_type)
```

Forcing authentication

When testing views directly using a request factory, it's ohen convenient to be able to directly authenticate the request, rather than having to construct the correct authentication credentials.

To forcibly authenticate a request, use the `force_authenticate()` method.

```
from rest_framework.test import  force_authenticate

factory = APIRequestFactory()
user = User.objects.get(username='olivia')
view = AccountDetail.as_view()

# Make an authenticated request to the view...
request = factory.get('/accounts/django-superstars/')
force_authenticate(request, user=user)
response = view(request)
```

The signature for the method is `force_authenticate(request, user=None, token=None)` making the call, either or both of the user and token may be set.

For example, when forcibly authenticating using a token, you might do something like the following:

```
user = User.objects.get(username='olivia')
request = factory.get('/accounts/django-superstars/')
force_authenticate(request, user=user, token=user.auth_token)
```

Note: `force_authenticate` directly sets `request.user` to the in-memory `user` instance. If you are re-using the same `user` instance across multiple tests that update the saved `user` state, you may need to call **`refresh_from_db()`** between tests.

Note: When using `APIRequestFactory`, the object that is returned is Django's standard `HttpRequest` and not REST framework's `Request` object, which is only generated once the view is called.

This means that setting attributes directly on the request object may not always have the eVect you expect. For example, setting `.token` directly will have no eVect, and setting `.user` directly will only work if session authentication is being used.

```
# Request will only authenticate if `SessionAuthentication` is in  use.
request = factory.get('/accounts/django-superstars/')
request.user = user
response = view(request)
```

Forcing CSRF validation

By default, requests created with `APIRequestFactory` will not have CSRF validation applied when passed to a REST framework view. If you need to explicitly turn CSRF validation on, you can do so by setting the `enforce_csrf_checks` flag when instantiating the factory.

```
factory = APIRequestFactory(enforce_csrf_checks=True)
```

Note: It's worth noting that Django's standard `RequestFactory` doesn't need to include this option, because when using regular Django the CSRF validation takes place in middleware, which is not run when testing views directly. When using REST framework, CSRF validation takes place inside the view, so the request factory needs to disable view-level CSRF checks.

APIClient

Extends Django's existing `Client` class.

Making requests

The `APIClient` class supports the same request interface as Django's standard `Client` class. This means that the standard `.get()`, `.post()`, `.put()`, `.patch()`, `.delete()`, a`.head()` `ons()` methods are all available. For example:

```
from rest framework.test import  APIClient

client = APIClient()
client.post('/notes/', {'title': 'new idea'}, format='json')
```

To support a wider set of request formats, or change the default format, see the configuration section.

Authenticating

.login(**kwargs)

The `login` method functions exactly as it does with Django's regular `Client` class. Thisallows you to authenticate requests against any views which include `SessionAuthentication` .

```
# Make all requests in the context of a logged in  session.
client = APIClient()
client.login(username='lauren', password='secret')
```

To logout, call the `logout` method as usual.

```
# Log out
client.logout()
```

The `login` method is appropriate for testing APIs that use session authentication, for example web sites which include AJAX interaction with the API.

.credentials(**kwargs)

The `credentials` method can be used to set headers that will then be included on all subsequent requests by the test client.

```
from rest_framework.authtoken.models import  Token
from rest_framework.test import APIClient

# Include an appropriate `Authorization:` header on all requests.
token = Token.objects.get(user_username='lauren')
client = APIClient()
client.credentials(HTTP_AUTHORIZATION='Token ' + token.key)
```

Note that calling `credentials` a second time overwrites any existing credentials. You can unset any existing credentials by calling the method with no arguments.

```
# Stop including any credentials
client.credentials()
```

The `credentials` method is appropriate for testing APIs that require authentication headers, such as basic authentication, OAuth1a and OAuth2 authentication, and simple token authentication schemes.

.force_authenticate(user=None, token=None)

Sometimes you may want to bypass authentication entirely and force all requests by the test client to be automatically treated as authenticated.

This can be a useful shortcut if you're testing the API but don't want to have to construct valid authentication credentials in order to make test requests.

```
user = User.objects.get(username='lauren')
client = APIClient()
client.force_authenticate(user=user)
```

To unauthenticate subsequent requests, call `force_authenticate` setting the user and/or token to `None` .

```
client.force_authenticate(user=None)
```

CSRF validation

By default CSRF validation is not applied when using `APIClient` . If you need to explicitly enable CSRF validation, you can do so by setting the `enforce_csrf_checks` flag when instantiating the client.

```
client = APIClient(enforce_csrf_checks=True)
```

As usual CSRF validation will only apply to any session authenticated views. This means CSRF validation will only occur if the client has been logged in by calling `login()` .

RequestsClient

REST framework also includes a client for interacting with your application using the popular Python library, `requests` . This may be useful if:

- You are expecting to interface with the API primarily from another Python service, and want to test the service at the same level as the client will see.
- You want to write tests in such a way that they can also be run against a staging or live environment. (See "Live tests" below.)

This exposes exactly the same interface as if you were using a requests session directly.

```
from rest_framework.test import  RequestsClient

client = RequestsClient()
response = client.get('http://testserver/users/')
assert response.status_code == 200
```

Note that the requests client requires you to pass fully qualified URLs.

RequestsClient and working with the database

The `RequestsClient` class is useful if you want to write tests that solely interact with the service interface. This is a little stricter than using the standard Django test client, as it means that all interactions should be via the API.

If you're using `RequestsClient` you'll want to ensure that test setup, and results assertions are performed as regular API calls, rather than interacting with the database models directly. For example, rather than checking that `Customer.objects.count() == 3` you would list the customers endpoint, and ensure that it contains three records.

Headers & Authentication

Custom headers and authentication credentials can be provided in the same way as when using a standard `requests.Session` instance.

```
from requests.auth import HTTPBasicAuth

client.auth = HTTPBasicAuth('user', 'pass')
client.headers.update({'x-test': 'true'})
```

CSRF

If you're using `SessionAuthentication` then you'll need to include a CSRF token for any `POST`, `PUT`, `PATCH` or `DELETE`

You can do so by following the same flow that a JavaScript based client would use. First make a `GET` request in order to obtain a CRSF token, then present that token in the following request.

For example...

```
client = RequestsClient()

# Obtain a CSRF token.
response = client.get('http://testserver/homepage/')
assert response.status_code == 200
csrftoken = response.cookies['csrftoken']

# Interact with the API.
response = client.post('http://testserver/organisations/',
json={ 'name': 'MegaCorp',
'status': 'active'
}, headers={'X-CSRFToken': csrftoken})
assert response.status_code == 200
```

Live tests

With careful usage both the `RequestsClient` and the `CoreAPIClient` provide the ability to write test cases that can run either in development, or be run directly against your staging server or production environment.

Using this style to create basic tests of a few core piece of functionality is a powerful way to validate your live service. Doing so may require some careful attention to setup and teardown to ensure that the tests run in a way that they do not directly aVect customer data.

CoreAPIClient

The CoreAPIClient allows you to interact with your API using the Python `coreapi` client library.

```
    # Fetch the API schema
    client = CoreAPIClient()
    schema = client.get('http://testserver/schema/')

    # Create a new organisation
    params = {'name': 'MegaCorp', 'status': 'active'}
    client.action(schema, ['organisations', 'create'], params)

    # Ensure that the organisation exists in the listing
    data = client.action(schema, ['organisations', 'list'])
    assert(len(data) == 1)
    assert(data == [{'name': 'MegaCorp', 'status': 'active'}])
```

Headers & Authentication

Custom headers and authentication may be used with `CoreAPIClient` in a similar way as with `RequestsClient` .

```
    from requests.auth import  HTTPBasicAuth

    client = CoreAPIClient()
    client.session.auth = HTTPBasicAuth('user', 'pass')
    client.session.headers.update({'x-test': 'true'})
```

API Test cases

REST framework includes the following test case classes, that mirror the existing Django test case classes, but use `APIClient` instead of Django's default `Client` .

- `APISimpleTestCase`
- `APITransactionTestCase`
- `APITestCase`
- `APILiveServerTestCase`

Example

You can use any of REST framework's test case classes as you would for the regular Django test case classes. The `self.client` attribute will be an `APIClient` instance.

```
from django.urls import reverse
from rest_framework import  status
from rest_framework.test import  APITestCase
from myproject.apps.core.models import  Account

class AccountTests(APITestCase):
def test_create_account(self):
"""
Ensure we can create a new account object. """
url = reverse('account-list') data =
{'name': 'DabApps'}
response = self.client.post(url, data, format='json')
self.assertEqual(response.status_code, status.HTTP_201_CREATED)
self.assertEqual(Account.objects.count(), 1)
self.assertEqual(Account.objects.get().name, 'DabApps')

```

URLPatternsTestCase

REST framework also provides a test case class for isolating `urlpatterns` on a per-class basis. Note that this inherits from Django's `SimpleTestCase` and will most likely need to be mixed with another test case class.

Example

```
from django.urls import include, path, reverse
from rest framework.test import APITestCase, URLPatternsTestCase


class AccountTests(APITestCase, URLPatternsTestCase):
urlpatterns = [
path('api/', include('api.urls')),
]

def test_create_account(self): """

```

```
        Ensure we can create a new account object.
        """
        url = reverse('account-list')
        response = self.client.get(url, format='json')
        self.assertEqual(response.status_code, status.HTTP_200_OK)
        self.assertEqual(len(response.data), 1)
```

Testing responses

Checking the response data

When checking the validity of test responses it's ohen more convenient to inspect the data that the response was created with, rather than inspecting the fully rendered response.

For example, it's easier to inspect `response.data` :

```
response = self.client.get('/users/4/')
self.assertEqual(response.data, {'id': 4, 'username': 'lauren'})
```

Instead of inspecting the result of parsing `response.content` :

```
response = self.client.get('/users/4/')
self.assertEqual(json.loads(response.content), {'id': 4, 'username':
'lauren'})
```

Rendering responses

If you're testing views directly using `APIRequestFactory` , the responses that are returned will not yet be rendered, as rendering of template responses is performed by Django's internal request-response cycle. In order to access `response.content` , you'll first need to render the response.

```
view = UserDetail.as_view()
request = factory.get('/users/4')
response = view(request, pk='4')
response.render()  # Cannot access `response.content` without this.
self.assertEqual(response.content, '{"username": "lauren", "id": 4}')
```

Configuration

Setting the default format

The default format used to make test requests may be set using the setting key `TEST_REQUEST_DEFAULT_FORMAT` For example, to always use JSON for test requests by default instead of standard multipart form requests, set the following in your file: `settings.py`

```
REST_FRAMEWORK = {
    ...
    'TEST_REQUEST_DEFAULT_FORMAT': 'json'
}
```

Setting the available formats

If you need to test requests using something other than multipart or json requests, you can do so by setting the `TEST_REQUEST_RENDERER_CLASSES` setting.

For example, to add support for using `format='html'` in test requests, you might have something like this in your `settings.py` file.

```
REST_FRAMEWORK = {
    ...
    'TEST_REQUEST_RENDERER_CLASSES': (
        'rest_framework.renderers.MultiPartRenderer',
        'rest_framework.renderers.JSONRenderer',
        'rest_framework.renderers.TemplateHTMLRenderer'
    )
}
```

API客户端

API客户端为开发人员提供了一个方便的API测试接口，而不需要使用浏览器。　　客户端通常有两种，基于命令行的和图形化界面的

一、命令行客户端

命令行客户端常用的有curl、wget、HTTPie和coreapi。这里介绍coreapi。

注意coreapi的命令行客户端，不等同于python客户端模块，需要单独安装：

```
$ pip install  coreapi-cli
```

下面是一个使用的例子：

```
$ coreapi get http://api.example.org/
<Pastebin API "http://127.0.0.1:8000/">
snippets: {
    create(code, [title], [linenos], [language], [style])
    destroy(pk)
    highlight(pk)
    list([page])
    partial_update(pk, [title], [code], [linenos], [language],  [style])
    retrieve(pk)
update(pk, code, [title], [linenos], [language], [style]) 11}
users: {
    list([page])
retrieve(pk) 15
}
```

使用coreapi action命令和API进行交互：

```
$ coreapi action users list
[
    {
        "url":  "http://127.0.0.1:8000/users/2/",
        "id": 2,
        "username": "aziz",
        "snippets": []
    },
    ...
]
```

使用 `--debug` 标识，查看具体的交互信息：

```
$ coreapi action users list --debug
> GET /users/ HTTP/1.1
> Accept: application/vnd.coreapi+json, */*
> Authorization: Basic bWF4Om1heA==
> Host: 127.0.0.1
> User-Agent: coreapi
< 200 OK
< Allow: GET, HEAD, OPTIONS
< Content-Type: application/json
< Date: Thu, 30 Jun 2016 10:51:46 GMT
< Server: WSGIServer/0.1 Python/2.7.10
< Vary: Accept, Cookie
<
< [{"url":"http://127.0.0.1/users/2/","id":2,"username":"aziz","snippets":
[]},{"url":"http://127.0.0.1/users/3/","id":3,"username":"amy","snippets":
["http://127.0.0.1/snippets/3/"]},
{"url":"http://127.0.0.1/users/4/","id":4,"username":"max","snippets":
["http://127.0.0.1/snippets/4/","http://127.0.0.1/snippets/5/","http://127.
0.0.1/snippets/6/","http://127.0.0.1/snippets/7/"]},
{"url":"http://127.0.0.1/users/5/","id":5,"username":"jose","snippets":[]},
{"url":"http://127.0.0.1/users/6/","id":6,"username":"admin","snippets":
["http://127.0.0.1/snippets/1/","http://127.0.0.1/snippets/2/"]}]

[
    ...
]
```

使用选项或必填的参数：

```
$ coreapi action users create --param  username=example
```

如果你想让命令的意思更加明确，使用 `--data` 标识用于空、数字、布尔、列表或者对象的输入，使用 `--string` 标识用于字符串输入：

```
$ coreapi action users edit --string username=tomchristie --data
is_admin=true
```

认证和头部属性

加入证书：

```
$ coreapi credentials add <domain> <credentials string>
```

例如：

```
$ coreapi credentials add api.example.org "Token
9944b09199c62bcf9418ad846dd0e4bbdfc6ee4b"
```

`--auth` 标识用于选择认证类型：

```
$ coreapi credentials add api.example.org tomchristie:foobar --auth basic
```

使用 `headers` 命令添加头部属性：

```
$ coreapi headers add api.example.org x-api-version 2
```

使用 `coreapi help` 查看更多的帮助。

编码解码器Codecs

默认情况下，coreapi只支持JSON格式，但是可以安装别的插件来增加支持的类型：

```
$ pip install openapi-codec jsonhyperschema-codec hal-codec
$ coreapi codecs show
Codecs
corejson         application/vnd.coreapi+json encoding, decoding
hal              application/hal+json         encoding, decoding
openapi          application/openapi+json     encoding, decoding
jsonhyperschema  application/schema+json      decoding
json             application/json             data
text             text/*                       data
```

Utilities

书签功能:

```
$ coreapi bookmarks add accountmanagement
```

查看历史记录:

```
$ coreapi history show
$ coreapi history back
```

更多帮助查看 `coreapi bookmarks --help` 或者 `coreapi history --help`.

其它命令

显示当前的 `Document`:

```
$ coreapi show
```

重载当前的 `Document` :

```
$ coreapi reload
```

从本地硬盘加载概要:

```
$ coreapi load my-api-schema.json --format corejson
```

dump当前文档:

```
$ coreapi dump --format openapi
```

删除当前文档:

```
$ coreapi clear
```

二、Python的coreapi模块

`coreapi` 这个Python模块允许你以编程的方式与API交互。这不同于前面的coreapi命令行使用环境。

安装：

```
$ pip install coreapi
```

导入模块后，首先需要实例化一个 `Client` 对象，如下所示：

```
import coreapi
client = coreapi.Client()
```

然后可以通过这个client对象访问API了：

```
schema = client.get('https://api.example.org/')
```

返回的是一个 `Document` 实例，也就是API的概要。

认证方式

使用 `TokenAuthentication` 类进行认证：

```
auth = coreapi.auth.TokenAuthentication(
    scheme='JWT',
token='<token>' 4
)
client = coreapi.Client(auth=auth)
```

或者使用 "Django REST framework JWT" 模块

```
client = coreapi.Client()
schema = client.get('https://api.example.org/')

action = ['api-token-auth', 'create']
params = {"username": "example", "password": "secret"}
result = client.action(schema, action, params)

auth =
    coreapi.auth.TokenAuthentication( scheme='JWT',
    token=result['token']
)
client = coreapi.Client(auth=auth)
```

或者使用最基础的 `BasicAuthentication` 方式，使用用户名和密码进行认证：

```
auth = coreapi.auth.BasicAuthentication(
    username='<username>',
password='<password>' 4
)
client = coreapi.Client(auth=auth)
```

与API交互

使用action方法进行交互：

```
users = client.action(schema, ['users', 'list'])
```

添加参数：

```
new_user = client.action(schema, ['users', 'create'], params={"username":
"max"})
```

Codecs

配置Codecs：

```
from coreapi import codecs,  Client

decoders = [codecs.CoreJSONCodec(), codecs.JSONCodec()]
client = Client(decoders=decoders)
```

可以直接使用：

```
input_file = open('my-api-schema.json', 'rb')
schema_definition = input_file.read()
codec = codecs.CoreJSONCodec()
schema = codec.load(schema_definition)
```

或者这样：

```
schema_definition = codec.dump(schema)
output_file = open('my-api-schema.json', 'rb')
output_file.write(schema_definition)
```

三、图形化API工具Postman

Postman是google开发的一款功能强大的API测试工具，可以作为Chrome浏览器的插件使用。 其主要功能包括：

- 允许用户发送任何类型的 HTTP 请求，例如 GET，POST，HEAD，PUT、DELETE等
- 允许任意的参数和 Headers
- 支持不同的认证机制，包括 Basic Auth，Digest Auth，OAuth 1.0，OAuth 2.0等
- 可以响应数据是自动按照语法格式高亮的，包括 HTML，JSON和XML

官方主页： https://www.getpostman.com/

下载链接： https://www.getpostman.com/downloads/

本来最简单的方法是在Chrome中以浏览器插件的方式安装postman，但是你懂的，所以只能下载app，并安装。

（在Linux命令行下老老实实用wget、curl、httpie和coreapi吧！别想图形界面的事了）

使用AJAX，CSRF和CORS

> "仔细观察你自己的网站上的可能的CSRF/XSRF漏洞。它们是最糟糕的漏洞 — 非常容易被攻击者利用。但对于软件开发人员来说，直到你被攻击了你才能有深入理解。"
>
> — JeG Atwood

Javascript客户端

如果你正在构建一个JavaScript客户端来和Web API对接，你将需要考虑客户端是否可以使用网站其他部分使用的身份验证策略，并确定是否需要使用CSRF令牌或CORS标头。

与正在交互的API相同的上下文中发起的AJAX请求通常将使用 `SessionAuthentication` 。 这确保一旦用户登录，任何AJAX请求都可以使用与网站其他部分相同的基于会话的身份验证。

与其通信的API在不同站点上发起的AJAX请求通常需要使用基于非会话的身份验证方案，例如 `TokenAuthentication` 。

CSRF保护

Cross Site Request Forgery防止特定类型的攻击，当用户没有注销登出网站并且具有有效会话时，这种攻击可能发生。在这种情况下，恶意站点可能会在登录会话的上下文中针对目标站点执行攻击操作。

为了防范这些类型的攻击，你需要做两件事情：

1. 确保"安全"的HTTP方法（如： `GET` ， `HEAD` 和 `OPTIONS` ）不用于更改服务器端的任何状态。
2. 确保"不安全"的HTTP方法（如： `POST` ， `PUT` ， `PATCH` 和 `DELETE` ）始终需要有效的CSRF令牌。

如果你使用的是 `SessionAuthentication` ，则需要为任何 `POST` ， `PUT` ， `PATCH` 或 `DELETE` 操作包含有效的CSRF令牌。

为了使用AJAX请求，你需要在HTTP报头中包含CSRF令牌，具体方法参考Django教程（liujiangblog.com）。

CORS

Cross-Origin Resource Sharing，是允许客户端与托管在不同域上的API交互的机制。CORS的工作原理是要求服务器包含一组特定的报头信息，允许浏览器确定是否以及何时允许跨域请求。

在REST框架中处理CORS的最佳方法是在中间件中添加所需的响应头信息。这样可以确保CORS得到透明支持，而无需更改视图中的任何行为。

`Otto Yiu` 维护着能在REST框架下使用的CORS支持工具，也就是django-cors-headers包。

1. 所有DRF能做的，Django都能做，只不过更麻烦。你可以不使用DRF的序列化等所有功能，但你必须自己实现，你可以认为DRF的功能不够强大，提供的类比较单薄，甚至所有的东西都自定义，而不使用DRF的任何已有类，那么你还用DRF干什么呢？你实际上已经创建了一个新模块。反过来说，如果只用DRF的类，而不做任何自定义或者修改，又无法满足实际工作的需求，甚至连基本功能可能都无法实现，这也说明DRF不够强大，不够体系。
2. RESTful只是个建议的规范，不是强制的
3. 以前，解析请求和渲染结果，都是Django帮我们做了，都是黑盒子。
4. 做前后端分离的项目的后端开发，就是做API或者接口的开发。后端将url暴露给前端，前端使用url访问后端的API。
5. FBV:基于函数的视图；CBV:基于类的视图
6. 关于CSRF：

```python
# 对于普通的基于函数的视图，如果想取消CSRF限制，直接使用csrf_exempt装饰器即可

from django.views.decorators.csrf import  csrf_exempt

@csrf_exempt
def someview(request):
    pass


# 对于基于类的视图，如果想取消CSRF限制，第一种方法是重写dispatch方法，添加csrf_exempt装饰器
from django.views import View
from django.utils.decorators import method_decorator


class SomeView(View):

    @method_decorator(csrf_exempt)
    def dispatch(self, request, *args, **kwargs):
        pass

    def get(self, request, *arg, **kwargs):
        pass

    pass


```

```python
# 对于基于类的视图，如果想取消CSRF限制，第二种方法是直接在类上添加csrf_exempt装饰器，注意参数
@method_decorator(csrf_exempt, name='dispatch')
class AnotherView(View):

    def get(self, request, *arg, **kwargs):
        pass

    pass


# 对于Django REST framework 它的APIView中的as_view()方法，帮我们使用了csrf_exempt()方法。

from django.views.decorators.csrf import csrf_exempt
class APIView(View):

    pass

    @classmethod
    def as_view(cls, **initkwargs):
        if isinstance(getattr(cls, 'queryset', None), models.query.QuerySet):
            def force_evaluation():
                raise RuntimeError(
                    'Do not evaluate the `.queryset` attribute directly, '
                    'as the result will be cached and reused between requests. '
                'Use `.all()` or call `.get_queryset()` instead.'
                )
            cls.queryset._fetch_all = force_evaluation

        view = super(APIView, cls).as_view(**initkwargs)
        view.cls = cls
        view.initkwargs = initkwargs

        # Note: session based authentication is explicitly CSRF validated,
        # all other authentication is CSRF exempt.
        return csrf_exempt(view)  # 看这里！！！！！！！！！！！！！！！
```

7. 阅读DRF的源码，从APIView类的dispatch方法开始。

8. 从DRF的request对象中获取Django原生的request对象：`requst._request`
9. 从Django的ORM的查询集中获取第一个对象使用`first()`方法

10. 可以在Pycharm里直接使用Sqlite3的插件对数据库内的数据进行CRUD，不用进入admin后台。
11. 在执行 `json.dumps` 的时候，如果提供参数 `ensure_ascii=False` ，不会将中文编码成ASCII码，更方便和直观。
12. serializer中的字段的名字，必须和对应的model中字段的名字一样。但是你如果使用了source参数，那么就可以不一样了。
13. 使用Django原生FBV能力编写API接口

```
import json
from django.views.decorators.csrf import csrf_exempt
from django.http import JsonResponse

user_dict = {
    "name": 'tom',
    "sex": " male",
    "age": 18
}

@csrf_exempt
def userList(request):
    if request.method ==  'GET':
        # return HttpResponse(json.dumps(user_dict),
content type='application/json')
    if request.method == 'POST':
        # print(request.POST)
        # print(request.body)
        user_form = json.loads(request.body.decode('utf-8'))
        print(user_form)
        # return JsonResponse(user_form, safe=False)
        return HttpResponse(json.dumps(user_form),


content_type='application/json')
```

14. 使用Django的类视图CBV编写API：

```
url.py:
###################
path('users/',  views.UserList.as view())


```

```
views.py
######################
from django.views import View
from django.utils.decorators import  method_decorator


@method_decorator(csrf_exempt, name='dispatch')
class UserList(View):

    def get(self,request):
        return JsonResponse(user_dict)

    def post(self,request):
        data = json.loads(request.body.decode('utf-8'))
        print(data, type(data))
        # return HttpResponse(json.dumps(data), content_type='application/json')
        return JsonResponse(data)
```

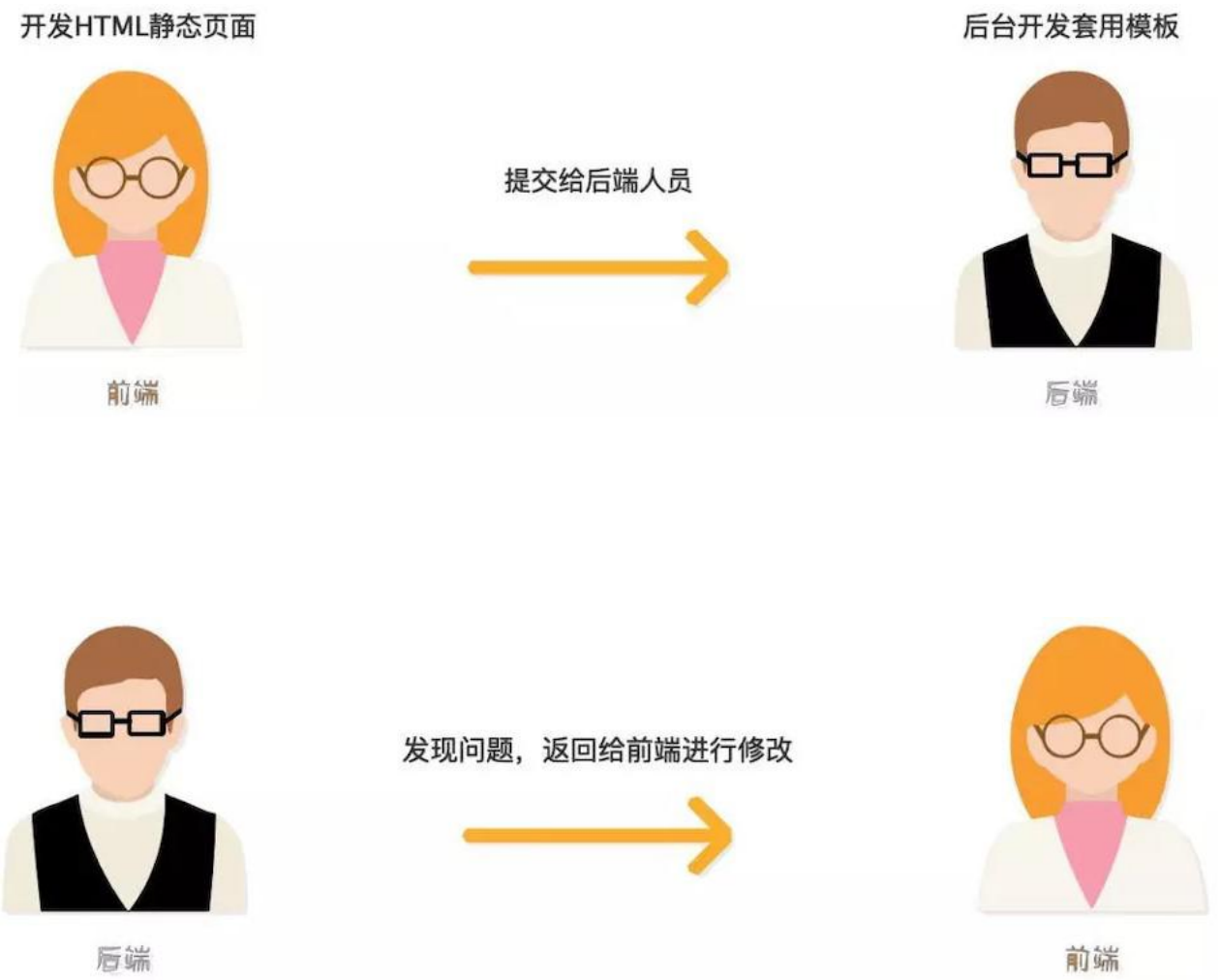
15. Django2.0之后的urls.py文件中的url模式，是完全匹配，所以'/users/'不会短路'/users/high/'。

从目前应用软件开发的发展趋势来看

- - 越来越注重用户体验，随着互联网的发展，开始多终端化。
- - 大型应用架构模式正在向云化、微服务化发展。

传统的开发模式

前后端分离前我们的开发协作模式一般是这样的：



前端写好静态的HTML页面交付给后端开发。静态页面可以本地开发，也无需考虑业务逻辑只需要实现View即可。

后端使用模板引擎去套模板，当年使用最广泛的就是jsp，freemarker等等，同时内嵌一些后端提供的模板变量和一些逻辑操作。

然后前后端集成对接，遇到问题，前台返工，后台返工。

然后在集成，直至集成成功。

这种模式的问题：

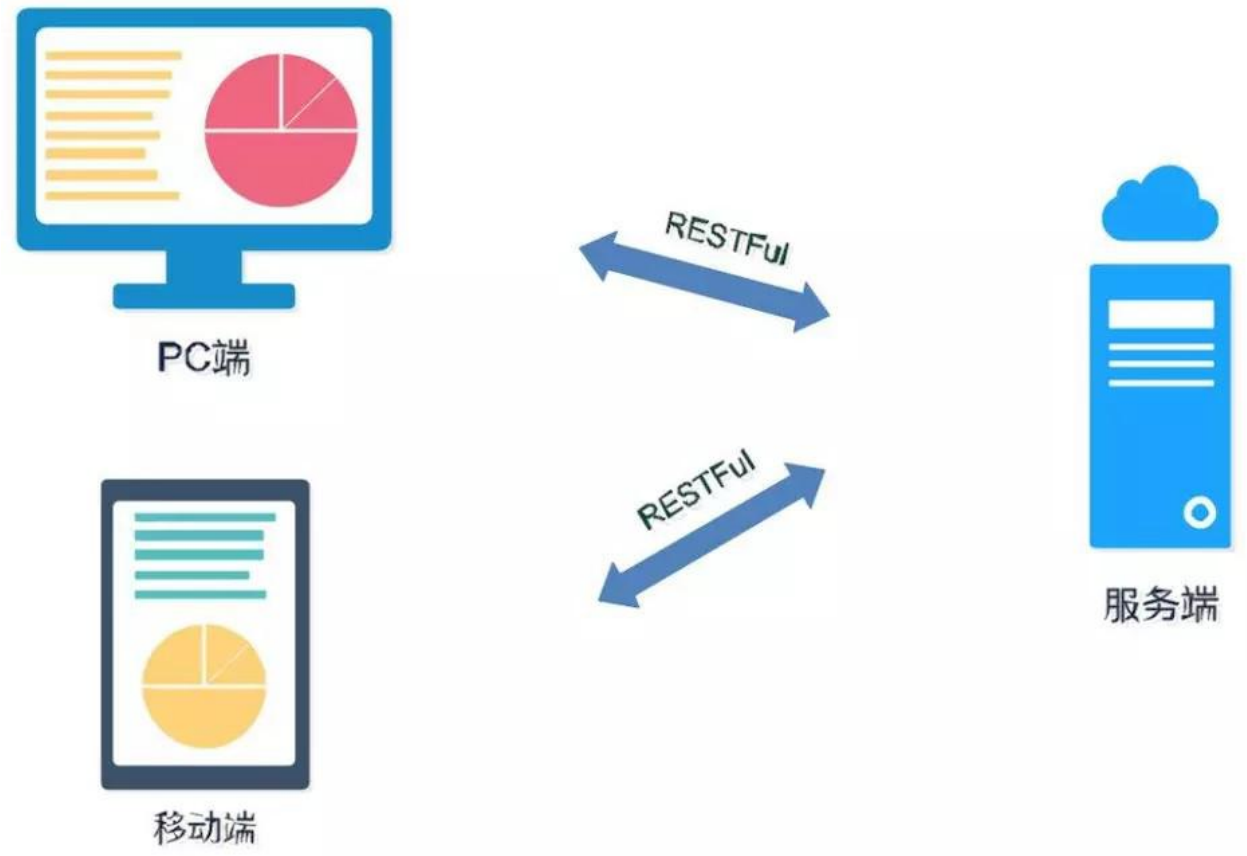
在前端调试的时候要安装完整的一套后端开发工具，要把后端程序完全启动起来。遇到问题需要后端开发来帮忙调试。我们发现前后端严重耦合，还要要求后端人员会一些HTML，JS等前端语言。前端页面里还嵌入了很多后端的代码。一旦后端换了一种语言开发，简直就要重做。

像这种增加了大量的沟通成本，调试成本等，而且前后端的开发进度相互影响，从而大大降低了开发效率。

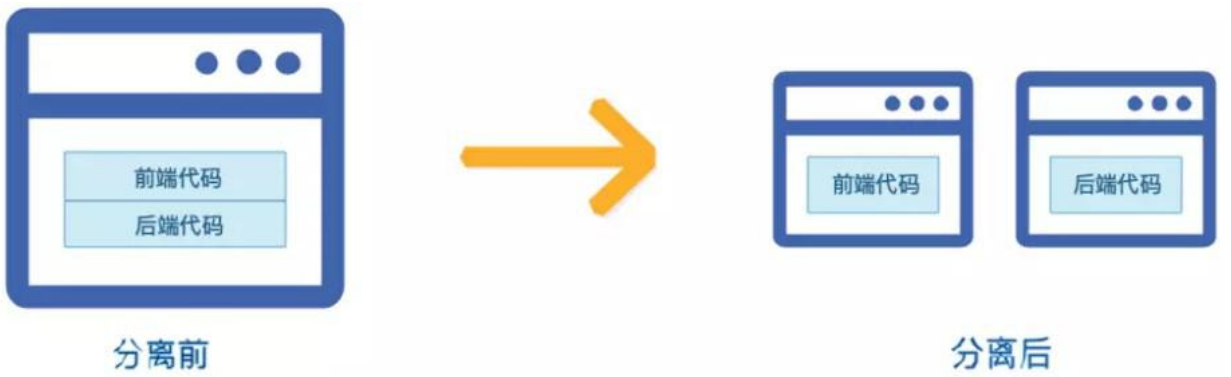
前后端分离

前后端分离并不只是开发模式，而是web应用的一种架构模式。在开发阶段，前后端工程师约定好数据交互接口，实现并行开发和测试；在运行阶段前后端分离模式需要对web应用进行分离部署，前后端之前使用HTTP或者其他协议进行交互请求。

1. 客户端和服务端采用RESTFul API的交互方式进行交互



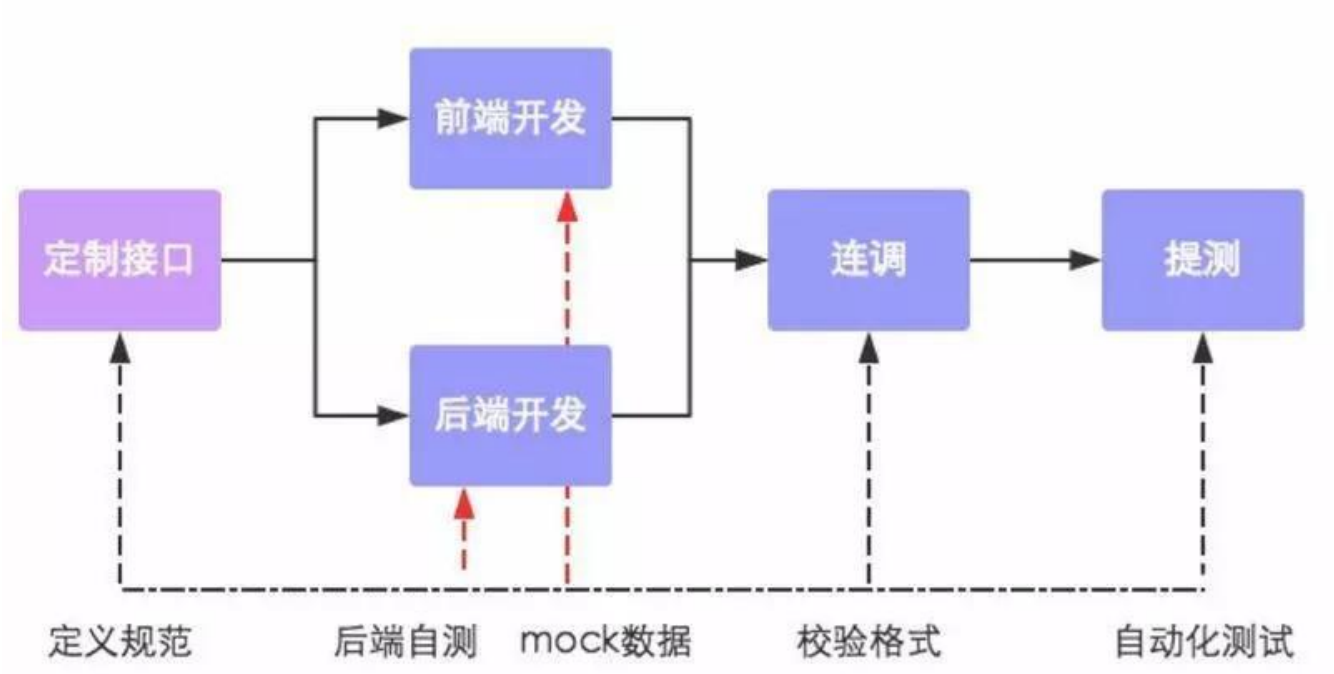
2. 前后端代码库分离



在传统架构模式中，前后端代码存放于同一个代码库中，甚至是同一工程目录下。页面中还夹杂着后端代码。前后端工程师进行开发时，都必须把整个项目导入到开发工具中。

前后端代码库分离，前端代码中有可以进行Mock测试(通过构造虚拟测试对 象以简化测试环境的方法)的伪后端，能支持前端的独立开发和测试。而后端代码中除了功能实现外，还有着详细的测试用例，以保证API的可用性，降低集成风险。

3. 并行开发



在开发期间前后端共同商定好数据接口的交互形式和数据格式。然后实现前后端的并行开发，其中前端工程师在开发完成之后可以独自进行mock测试，而后端也可以使用Postman等接口测试软件进行接口自测，然后前后端一起进行功能联调并校验格式，最终进行自动化测试。

分离后，开发模式是这样的：

1. 接收数据，返回数据
2. 处理渲染逻辑
3. Client-side MV* 架构
4. 代码跑在浏览器上
5. 独立开发

前端

模拟测试

1. 提供数据
2. 处理业务逻辑
3. Server-side MVC架构
4. 代码跑在服务器上
5. 独立开发

Postman

模拟测试

后端

4. 前后端分离架构后的优点：

为优质产品打造精益团队

通过将开发团队前后端分离化，让前后端工程师只需要专注于前端或后端的开发工作，是的前后端工程师实现自治，培养其独特的技术特性，然后构建出一个全栈式的精益开发团队。

提升开发效率

前后端分离以后，可以实现前后端代码的解耦，只要前后端沟通约定好应用所需接口以及接口参数，便可以开始并行开发，无需等待对方的开发工作结束。与此同时，即使需求发生变更，只要接口与数据格式不变，后端开发人员就不需要修改代码，只要前端进行变动即可。如此一来整个应用的开发效率必然会有质的提升。

完美应对复杂多变的前端需求

如果开发团队能完成前后端分离的转型，打造优秀的前后端团队，开发独立化，让开发人员做到专注专精，开发能力必然会有所提升，能够完美应对各种复杂多变的前端需求。

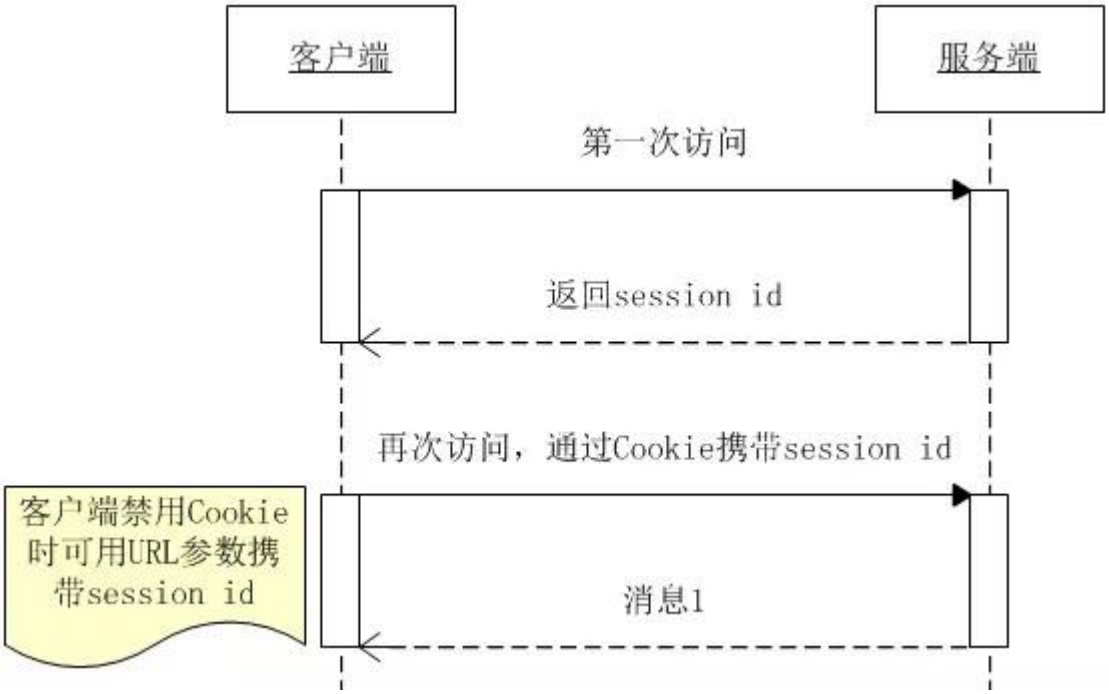
增强代码可维护性

前后端分离后，应用的代码不再是前后端混合，只有在运行期才会有调用依赖关系。应用代码将会变得整洁清晰，不论是代码阅读还是代码维护都会比以前轻松。

使用了前后端分离架构后，除了开发模式的变更，我们在开发的过程中还会遇到哪些问题呢？我们接着往下看。

用户认证

我们先来看看传统开发，我们是如何进行用户认证的：



前端登录，后端根据用户信息生成一个jsessionid，并保存这个 jsessionid 和对应的用户id到Session中，接着把 jsessionid 传给用户，存入浏览器 cookie，之后浏览器请求带上这个cookie，后端根据这个cookie值来查询用户，验证是否过期。

HTTP有一个特性：无状态的，就是前后两个HTTP事务它们并不知道对方的信息。而为了维护会话信息或用户信息，一般可用Cookie和Session技术缓存信息。

- Cookie是存储在客户端的

- Session是存储在服务端的

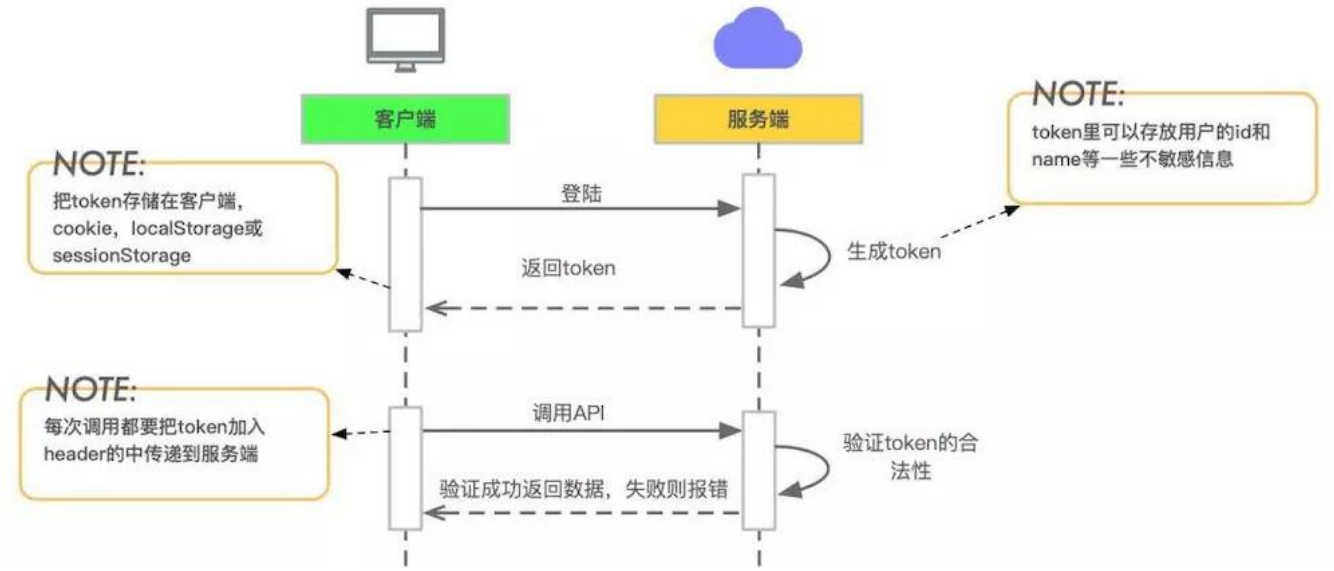
但这样做问题就很多，如果我们的页面出现了 XSS 漏洞，由于 cookie 可以被 JavaScript 读取，XSS 漏洞会导致用户 token 泄露，而作为后端识别用户的标识，cookie 的泄露意味着用户信息不再安全。尽管我们通过转义输出内容，使用 CDN 等可以尽量避免 XSS 注入，但谁也不能保证在大型的项目中不会出现这个问题。

在设置 cookie 的时候，其实你还可以设置 httpOnly 以及 secure 项。设置 httpOnly 后 cookie 将不能被 JS 读取，浏览器会自动的把它加在请求的 header 当中，设置 secure 的话，cookie 就只允许通过 HTTPS 传输。secure 选项可以过滤掉一些使用 HTTP 协议的 XSS 注入，但并不能完全阻止。

httpOnly 选项使得 JS 不能读取到 cookie，那么 XSS 注入的问题也基本不用担心了。但设置 httpOnly 就带来了另一个问题，就是很容易的被 XSRF，即跨站请求伪造。当你浏览器开着这个页面的时候，另一个页面可以很容易的跨站请求这个页面的内容。因为 cookie 默认被发了出去。

JWT解决方案

JWT （Json Web Token）



JWT 是一个开放标准(RFC 7519)，它定义了一种用于简洁，自包含的用于通信双方之间以 JSON 对象的形式安全传递信息的方法。该信息可以被验证和信任，因为它是数字签名的。JWTS可以使用秘密（使用HMAC算法）或公钥/私钥对使用RSA或ECDSA来签名。

- 简洁(Compact)：可以通过URL，POST 参数或者在 HTTP header 发送，因为数据量小，传输速度快。

- 自包含(Self-contained)：负载中包含了所有用户所需要的信息，避免了多次查询数据库。

JWT 组成

JWT由3个子字符串组成，分别为Header，Payload以及Signature，结合JWT的格式即：Header.Payload.Signature。（Claim是描述Json的信息的一个Json，将Claim转码之后生成Payload）。

Header

Header是由下面这个格式的Json通过Base64编码（编码不是加密，是可以通过反编码的方式获取到这个原来的Json，所以JWT中存放的一般是不敏感的信息）生成的字符串，Header中存放的内容是说明编码对象是一个JWT以及使用“SHA-256”的算法进行加密（加密用于生成Signature）

{ "typ":"JWT

",

"alg":"HS256"

}

- 头部包含了两部分，token 类型和采用的加密算法

- Base64是一种编码，也就是说，它是可以被翻译回原来的样子来的。它并不是一种加密过程。

JWS	算法名称	描述
HS256	HMAC256	HMAC with SHA-256
HS384	HMAC384	HMAC with SHA-384
HS512	HMAC512	HMAC with SHA-512
RS256	RSA256	RSASSA-PKCS1-v1_5 with SHA-256
RS384	RSA384	RSASSA-PKCS1-v1_5 with SHA-384
RS512	RSA512	RSASSA-PKCS1-v1_5 with SHA-512
ES256	ECDSA256	ECDSA with curve P-256 and SHA-256
ES384	ECDSA384	ECDSA with curve P-384 and SHA-384
ES512	ECDSA512	ECDSA with curve P-521 and SHA-512

Payload

Payload是通过Claim进行Base64转码之后生成的一串字符串，Claim是一个Json，Claim中存放的内容是JWT自身的标准属性，所有的标准属性都是可选的，可以自行添加，比如：JWT的签发者、JWT的接收者、JWT的持续时间等；同时Claim中也可以存放一些自定义的属性，这个自定义的属性就是在用户认证中用于标明用户身份的一个属性，比如用户存放在数据库中的id，为了安全起见，一般不会将用户名及密码这类敏感的信息存放在Claim中。将Claim通过Base64转码之后生成的一串字符串称作Payload。

{

"iss":"Issuer —— 用于说明该JWT是由谁签发的",

"sub":"Subject —— 用于说明该JWT面向的对象",

"aud":"Audience —— 用于说明该JWT发送给的用户",

"exp":"Expiration Time —— 数字类型，说明该JWT过期的时间",

"nbf":"Not Before —— 数字类型，说明在该时间之前JWT不能被接受与处理",

"iat":"Issued At —— 数字类型，说明该JWT何时被签发",

"jti":"JWT ID —— 说明标明JWT的唯一ID",

"user-definde1":"自定义属性举例",

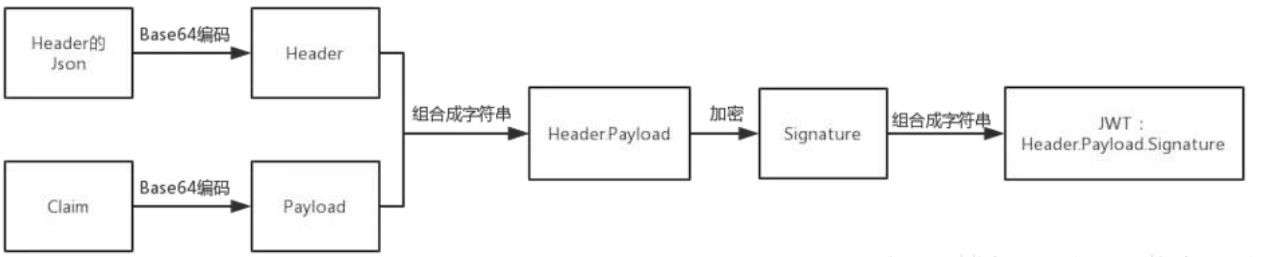
"user-definde2":"自定义属性举例"

```
}
```

Signature

Signature是由Header和Payload组合而成，将Header和Claim这两个Json分别使用Base64方式进行编码，生成字符串Header和Payload，然后将Header和Payload以Header.Payload的格式组合在一起形成一个字符串，然后使用上面定义好的加密算法和一个密匙（这个密匙存放在服务器上，用于进行验证）对这个字符串进行加密，形成一个新的字符串，这个字符串就是Signature。

签名的目的：最后一步签名的过程，实际上是对头部以及负载内容进行签名，防止内容被窜改。如果有人对头部以及负载的内容解码之后进行修改，再进行编码，最后加上之前的签名组合形成新的JWT的话，那么服务器端会判断出新的头部和负载形成的签名和JWT附带上的签名是不一样的。如果要对新的头部和负载进行签名，在不知道服务器加密时用的密钥的话，得出来的签名也是不一样的。



三个部分通过.连接在一起就是我们的 JWT 了：

eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJpZCI6IjU3ZmVmMTY0ZTU0YWY2NGZmYzUzZGJkNSIsInhzcmYiOiI0ZWE1YzUwOGE2NTY2ZTc2MjQwNTQzZjhmZWIwNmZkNDU3Nzc3YmUzOTU0OWM0MDE2NDM2YWZkYTY1ZDIzMzBlIiwiaWF0IjoxNDc2NDI3OTMzfQ.PA3QjeyZSUh7H0GfE0vJaKW4LjKJuC3dVLQiY4hii8s

JWT认证

服务器在生成一个JWT之后会将这个token发送到客户端机器，在客户端再次访问受到JWT保护的资源URL链接的时候，服务器会获取到这个token信息，首先将Header进行反编码获取到加密的算法，在通过存放在服务器上的密匙对Header.Payload 这个字符串进行加密，比对token中的Signature和实际加密出来的结果是否一致，如果一致那么说明该token是合法有效的，认证成功，否则认证失败。

JWT使用总结

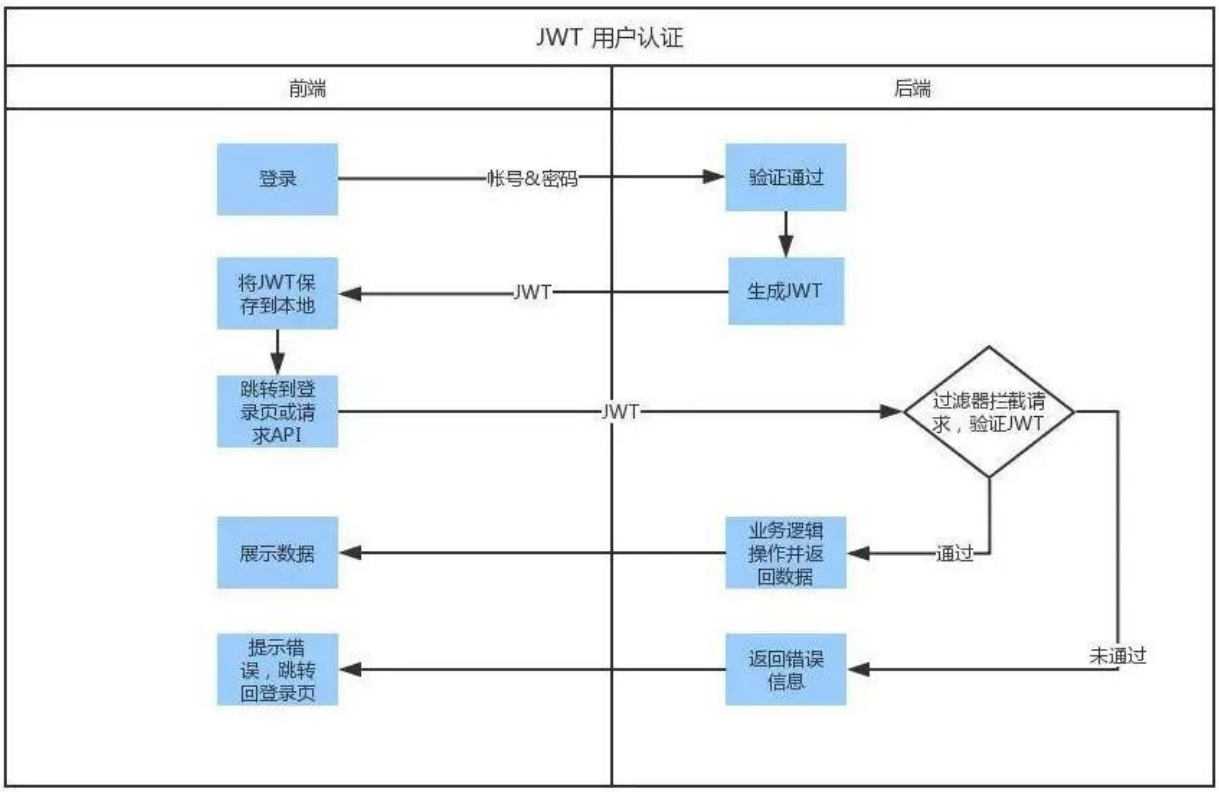
1. 首先，前端通过Web表单将自己的用户名和密码发送到后端的接口。这一过程一般是一个HTTP POST请求。建议的方式是通过SSL加密的传输（https协议），从而避免敏感信息被嗅探。
2. 后端核对用户名和密码成功后，将用户的id等其他信息作为JWT Payload（负载），将其与头部分别进行Base64编码拼接后签名，形成一个JWT。形成的JWT就是一个形同lll.zzz.xxx的字符串。
3. 后端将JWT字符串作为登录成功的返回结果返回给前端。前端可以将返回的结果保存在Cookie或localStorage或sessionStorage上，退出登录时前端删除保存的JWT即可。

特性	Cookie	LocalStorage	sessionStorage
数据的声明周期	一般由服务器生成，可设置失效时间。如果在浏览器生成，默认是关闭浏览器之后失效	除非被清楚，否则永久保存	仅在当前会话有效，关闭页面或浏览器后被清除
存放数据大小	4KB	*一般 5MB*	*一般 5MB*
与服务端通信	*每次都会携带在HTTP头中，如果使用 cookie 保存过多数据会带来性能问题*	仅在客户端中保存，不参与和服务器的通信。~~也可有脚本选择性提交到服务器端?~~	*同 LocalStorage*
用途	一般由服务器端生成，用于标识用户身份	用于浏览器端缓存数据	*同 LocalStorage*
共享			

4. 前端在每次请求时将JWT放入HTTP Header中的Authorization位。(解决XSS和XSRF问题)

5. 后端检查是否存在，如存在验证JWT的有效性。例如，检查签名是否正确；检查Token是否过期；检查Token的接收方是否是自己（可选）。

6. 验证通过后后端使用JWT中包含的用户信息进行其他逻辑操作，返回相应结果。



JWT和Session方式存储id的差异

Session方式存储用户id的最大弊病在于Session是存储在服务器端的，所以需要占用大量服务器内存，对于较大型应用而言可能还要保存许多的状态。一般而言，大型应用还需要借助一些KV数据库和一系列缓存机制来实现Session的存储。

而JWT方式将用户状态分散到了客户端中，可以明显减轻服务端的内存压力。除了用户id之外，还可以存储其他的和用户相关的信息，例如该用户是否是管理员、用户所在的分组等。虽说JWT方式让服务器有一些计算压力（例如加密、编码和解码），但是这些压力相比磁盘存储而言可能就不算什么了。

单点登录

Session方式来存储用户id，一开始用户的Session只会存储在一台服务器上。对于有多个子域名的站点，每个子域名至少会对应一台不同的服务器，例如：

www.taobao.com，nv.taobao.com，nz.taobao.com，login.taobao.com。所以如果要实现在login.taobao.com登录后，在其他的子域名下依然可以取到Session，这要求我们在多台服务器上同步Session。使用JWT的方式则没有这个问题的存在，因为用户的状态已经被传送到了客户端。

跨域问题

当客户端和服务端分开部署到不同服务器的时候，就会遇到跨域访问的问题，是由浏览器同源策略限制的一类请求场景。

URL	说明	是否允许通信
http://www.a.com/a.js http://www.a.com/b.js	同一域名下	允许
http://www.a.com/lab/a.js http://www.a.com/script/b.js	同一域名下不同文件夹	允许
http://www.a.com:8000/a.js http://www.a.com/b.js	同一域名，不同端口	不允许
http://www.a.com/a.js https://www.a.com/b.js	同一域名，不同协议	不允许
http://www.a.com/a.js http://70.32.92.74/b.js	域名和域名对应ip	不允许
http://www.a.com/a.js http://script.a.com/b.js	主域相同，子域不同	不允许
http://www.a.com/a.js http://a.com/b.js	同一域名，不同二级域名（同上）	不允许（cookie这种情况下也不允许访问）
http://www.cnblogs.com/a.js http://www.a.com/b.js	不同域名	不允许

跨域解决方案

推荐使用Nginx反向代理的方案：

反向代理

代理访问其实在实际应用中有很多场景，在跨域中应用的原理做法为：通过反向代理服务器监听同端口，同域名的访问，不同路径映射到不同的地址，比如，在nginx服务器中，监听同一个域名和端口，不同路径转发到客户端和服务器，把不同端口和域名的限制通过反向代理，来解决跨域的问题：

server

{ listen

80;

```
server_name domain.com;
```

```
#charset koi8-r;

#access_log logs/host.access.log main;

location /client { #访问客户端路径

proxy_pass http://localhost:81 ;

proxy_redirect default;

}

location /apis { #访问服务器路径

rewrite ^/apis/(.*) 1 break;

proxy_pass http://localhost:82 ;

}

}
```