# 一、版本控制Versioning

API 版本控制允许你在不同的客户端之间更改行为。REST framework 提供了许多不同的版本控制方案。

版本控制由传入的客户端请求决定，可以基于请求的URL中的版本部分，也可以基于请求头中属性定义的值。

## 使用DRF进行版本控制

当启用API版本控制后，`request.version` 属性中将包含与当前请求中版本相对应的字符串。

默认情况下，版本控制没有启用，`request.version` 将总是返回 `None`。

- **根据版本改变行为**

如何改变API的行为取决于你自己，但是一个常见的做法是在新版本中使用不同的序列化样式。例如:

```
1  def get_serializer_class(self):
2      if self.request.version == 'v1':
3          return AccountSerializerVersion1
4      return AccountSerializer
```

- **反向解析URLs中的版本APIs**

REST framework 包含的 `reverse` 函数与版本控制方案相关联。你需要确保将当前 `request` 作为关键字参数传递进去，如下所示。

```
1  from rest_framework.reverse import reverse
2
3  reverse('bookings-list', request=request)
```

上述函数将应用任何适用于请求版本的URL转换。例如:

- 如果使用 `NamespacedVersioning`，并且API的版本是'v1'，那么将会查找 `'v1:bookings-list'`，反向解析为类似 `http://example.org/v1/bookings/` 的URL。也就是版本字段形式。

- 如果使用 `QueryParameterVersioning` 并且API的版本是 `1.0` ，那么返回的URL可能像这样 `http://example.org/bookings/?version=1.0` 。也就是版本参数形式。
- **版本控制和超链接序列化器**

当将超链接序列化样式与基于URL的版本控制方案一起使用时，请确保将请求作为上下文包含在序列化程序中。

```
1   def get(self, request):
2       queryset = Booking.objects.all()
3       serializer = BookingsSerializer(queryset, many=True, context={'request':
    request})# 看这里
4       return Response({'all_bookings': serializer.data})
```

这样做将会在所有返回的URL中包含适当的版本参数或字段。

# 配置版本控制

使用 `DEFAULT_VERSIONING_CLASS` 进行版本相关的配置：

```
1   REST_FRAMEWORK = {
2       'DEFAULT_VERSIONING_CLASS':
    'rest_framework.versioning.NamespaceVersioning'
3   }
```

**如果没有配置，那么 `DEFAULT_VERSIONING_CLASS` 的默认值为 `None` ，这种情况时 `request.version` 的值 `None` ，也就相当于关闭了版本控制。**

还可以在单个视图上设置版本控制方案。但是通常我们不这么做，因为全局使用统一的版本控制方案更有意义。如果你非要这么做，请在视图类中使用 `versioning_class` 属性：

```
1   class ProfileList(APIView):
2       versioning_class = versioning.QueryParameterVersioning
```

# 其他版本相关设置

以下的配置项也用于版本管理:

- `DEFAULT_VERSION` .:当版本控制信息不存在时用于设置 `request.version` 的默认值，默认为 `None` 。

- `ALLOWED_VERSIONS`:如果设置了此值，将限制提供服务的版本范围，如果客户端请求的版本不在此范围中，则会引发错误。请注意，用于`DEFAULT_VERSION`的值应该总是在`ALLOWED_VERSIONS`设置的范围中（除非是`None`）。该配置默认为`None`，表示不限制。
- `VERSION_PARAM`:url中代表版本相关的参数名，默认值是`'version'`。

你还可以通过自定义版本类，并使用`default_version`，`allowed_versions`和`version_param`三类变量，在每个视图或每个视图集的基础上设置自定义的版本方案。例如，如果你想自定义`URLPathVersioning`的子类：

```
from rest_framework.versioning import URLPathVersioning
from rest_framework.views import APIView

class ExampleVersioning(URLPathVersioning): # 继承它
    default_version = ... #指定自己想要的设置
    allowed_versions = ...
    version_param = ...

class ExampleView(APIVIew):
    versioning_class = ExampleVersioning # 在视图中使用
```

# 二、API 参考

DRF的版本类都位于versioning模块中，这个模块很简单，不到200行代码，主要定义了几个类：

- BaseVersioning：基类，用于继承
- AcceptHeaderVersioning：继承了BaseVersioning
- URLPathVersioning：继承了BaseVersioning
- NamespaceVersioning：继承了BaseVersioning
- HostNameVersioning：继承了BaseVersioning
- QueryParameterVersioning：继承了BaseVersioning

**五个主要类的区别在于，你在哪个地方提供请求的版本信息！**是HTTP头部？还是URL的一部分？还是URL的参数？等等

## AcceptHeaderVersioning

此版本方案要求客户端将版本要求放在`Accept`头部信息中。

下面是一个HTTP请求示例：

```
GET /bookings/ HTTP/1.1
Host: example.com
Accept: application/json; version=1.0 #注意最后的部分
```

在上面的示例请求中， `request.version` 属性将返回字符串 `'1.0'` 。

**基于 accept 头部的版本控制通常被认为是最佳实践**，尽管其他版本控制方式可能更适合你的客户端需求。

---

严格地说 `json` 媒体类型并不算作包含的参数之一。如果要构建明确指定的公共API，则可以考虑使用vendor 媒体类型。为此，请将渲染器配置为使用自定义媒体类型的基于JSON的渲染器：

```
class BookingsAPIRenderer(JSONRenderer):
    media_type = 'application/vnd.megacorp.bookings+json'
```

这样的话，你的HTTP请求报头会是下面的样子:

```
GET /bookings/ HTTP/1.1
Host: example.com
Accept: application/vnd.megacorp.bookings+json; version=1.0
```

# URLPathVersioning

将版本要求作为URL路径的一部分。注意下面的 `/v1/` 。

```
GET /v1/bookings/ HTTP/1.1
Host: example.com
Accept: application/json
```

你的URL conf中必须包含一个使用 `'version'` 关键字参数的匹配模式，以便路由器可以获取对应的值，也就是版本信息。

```
urlpatterns = [
    re_path(
        r'^(?P<version>(v1|v2))/bookings/$',
        bookings_list,
        name='bookings-list'
    ),
    re_path(
        r'^(?P<version>(v1|v2))/bookings/(?P<pk>[0-9]+)/$',
        bookings_detail,
        name='bookings-detail'
    )
]
```

## NamespaceVersioning

对于客户端，此方案与 `URLPathVersioning` 相同。唯一的区别是，它是如何在 Django 应用程序中配置的，因为它使用URL conf中的命名空间而不是URL conf中的关键字参数。

```
GET /v1/something/ HTTP/1.1
Host: example.com
Accept: application/json
```

使用此方案， `request.version` 属性是根据与传入请求的路径匹配的 `namespace` 确定的。

在下面的示例中，我们给一组视图提供了两个可能出现的不同的URL前缀，每个前缀在不同的命名空间下:

```
# bookings/urls.py
urlpatterns = [
    re_path(r'^$', bookings_list, name='bookings-list'),
    re_path(r'^(?P<pk>[0-9]+)/$', bookings_detail, name='bookings-detail')
]

# urls.py
urlpatterns = [
    re_path(r'^v1/bookings/', include('bookings.urls', namespace='v1')),
    re_path(r'^v2/bookings/', include('bookings.urls', namespace='v2'))
]
```

如果你只需要一个简单的版本方案， `URLPathVersioning` 和 `NamespaceVersioning` 都是合适的。 `URLPathVersioning` 这种方法可能更适合小型项目，对于更大的项目来说 `NamespaceVersioning` 可能更容易管理。

## HostNameVersioning

通过主机名控制版本方案要求客户端将请求的版本指定为URL中主机名的一部分。 例如，以下是对 `http://v1.example.com/bookings/` 的HTTP请求：

```
GET /bookings/ HTTP/1.1
Host: v1.example.com
Accept: application/json
```

默认情况下，此实现期望主机名与下面的正则表达式匹配:

```
^([a-zA-Z0-9]+)\.[a-zA-Z0-9]+\.[a-zA-Z0-9]+$
```

注意，第一组用括号括起来，表示这是主机名的匹配部分。

## QueryParameterVersioning

这种版本方案是在 URL 中包含版本信息作为查询参数的方式，例如：

```
GET /something/?version=0.1 HTTP/1.1
Host: example.com
Accept: application/json
```

# 三、自定义版本方案

要实现自定义的版本方案，只需要继承 `BaseVersioning` 类并重写 `determine_version` 方法。

下面的例子中使用一个自定义的 `X-API-Version` 头部属性来确定所请求的版本。

```
class XAPIVersionScheme(versioning.BaseVersioning):
    def determine_version(self, request, *args, **kwargs):
        return request.META.get('HTTP_X_API_VERSION', None)
```

如果你的版本化方案基于请求的URL，你还需要改变带有版本信息的URL的确定方式。为此，你应该重写类中的 `.reverse()` 方法。有关示例，请参见源代码。