1. **A API deve possuir pelo menos 4 entidades relevantes e relacionadas via mapeamento objeto relacional.**

As entidades utilizadas foram de um sistema de vendas, onde existem as entidades de Cliente(`Client`), Produto(`Product`), Funcionário(`Employee`), Venda(`Sale`) e Item de Venda(`SaleItem`). Uma Venda tem um Funcionário e um Cliente relacionados, e um Item de Venda tem uma Venda e um Produto relacionados. Uma Venda ao ser criada pega automaticamente a data atual e o usuário logado, e um usuário só poderá adicionar Itens de Venda nas Vendas que ele criou.

2. **Pelo menos uma entidade deve ser integrada ao esquema de autenticação do Django.**

A entidade `Employee` está integrada ao esquema de autenticação do Django, tendo na instancia da classe um atributo `user` que é uma relação com a classe `User` do `django.contrib.auth.models`.

```
core/models.py
#...
from django.contrib.auth.models import User

class Employee(models.Model):
  name = models.CharField(max_length=128)
  user = models.OneToOneField(User, related_name='user')
#...
```

3. **Parte da API deve ser somente leitura e parte deve ser acessível apenas para usuários autenticados.**

As rotas relacionadas a `Employee` são somente leitura e apenas acessíveis a usuários autenticados. As outras rotas são acessíveis a todos os usuários, mas apenas os usuários autenticados podem fazer alterações nos dados. Essas permissões foram feitas primeiramente utilizando as views `rest_framework.generics.ListAPIView` e `rest_framework.generics.RetrieveAPIView`, que permitem apenas métodos somente leitura. Para as outras views, foram utilizadas as views `rest_framework.generics.ListCreateAPIView` e `rest_framework.generics.RetrieveUpdateDestroyAPIView`, usando as classes do `rest_framework.permissions` e com permissões customizadas feitas no arquivo `permissions.py` do módulo `core`. As permissões são inseridas em cada view na variável `permission_classes`.

```
core/views.py
#...
from rest_framework import generics, permissions
from .permissions import *

#...

class SaleDetail(generics.RetrieveUpdateDestroyAPIView):

  #...

  permission_classes = (permissions.IsAuthenticatedOrReadOnly,
IsOnwerOrReadOnly,)
#...
```

4. **A API deve ser documentada com Swagger ou alguma outra sugestão da página: http://www.django-rest-framework.org/topics/documenting-your-api/.**

Foi utilizado o próprio Swagger, disponível na rota `/swagger/`. Após a instalação via `pip install django-rest-swagger`, é necessário adicionar `'rest_framework_swagger'` na configuração

INSTALLED_APPS do settings.py do projeto. Após isso, é só criar uma rota de url no arquivo urls.py do módulo core.

```
core/urls.py
#...
from rest_framework_swagger.views import get_swagger_view

schema_view = get_swagger_view(title='Final Project API')

urlpatterns = [
  url(r'^$', APIRoot.as_view(), name='root'),
  url(r'^swagger/$', schema_view),
#...
```

A documentação foi feita via comentário em cada view no arquivo views.py do módulo core. O Swagger faz a conversão automática da documentação para cada método explicitado e o mostra em sua página.

Exemplo de documentação:

```
core/views.py
#...
class ProductDetail(generics.RetrieveUpdateDestroyAPIView):
  """
  get: Return the given product.
  delete: Delete the given product.
  put: Update the given product.
  patch: Update the given product.
  """
 #...
```

5. Definir e usar critérios de paginação e Throttling. Esse último deve diferenciar usuários autenticados de não autenticados.

A paginação atual é de 5 itens por página, e a taxa de requisições é de 50 por hora para anônimos e 200 por hora para usuários autenticados. Paginação e Throttling são itens que apenas precisam ter suas referências adicionadas na variável REST_FRAMEWORK do settings.py do projeto.

```
finalproject/settings.py
#...
REST_FRAMEWORK = {
    'DEFAULT_PAGINATION_CLASS':
    'rest_framework.pagination.LimitOffsetPagination',
    'PAGE_SIZE': 5,

    'DEFAULT_THROTTLE_CLASSES': (
        'rest_framework.throttling.AnonRateThrottle',
        'rest_framework.throttling.UserRateThrottle',
    ),
    'DEFAULT_THROTTLE_RATES': {
        'anon': '50/hour',
        'user': '200/hour',
    },
 #...
```

6. Implementar para pelo menos 2 entidades: filtros, busca e ordenação.

Os itens citados estão implementados nas entidades Client e Product. Primeiro, se deve instalar via pip install django-filter, referenciar no arquivo settings.py do projeto

```
'DEFAULT_FILTER_BACKENDS':
('django_filters.rest_framework.DjangoFilterBackend',)
```
na variável `REST_FRAMEWORK`.

```
finalproject/settings.py
#...
REST_FRAMEWORK = {
    'DEFAULT_FILTER_BACKENDS':
('django_filters.rest_framework.DjangoFilterBackend',)
#...
```

Para criar os itens citados na entidade, basta referenciar os filtros necessários na variável `filter_backends` da view, e referenciar os atributos da entidade que serão filtrados/procurados/ordenados nas variáveis `filter_fields`, `search_fields` e `ordering_fields`, respectivamente.

```
core/views.py
#...
from django_filters.rest_framework import DjangoFilterBackend
from rest_framework import filters

#...

class ClientList(generics.ListCreateAPIView):
  filter_backends = (filters.SearchFilter, filters.OrderingFilter,
DjangoFilterBackend)
  ordering_fields = ('name',)
  search_fields = ('name',)
  filter_fields = ('name',)
#...
```

7. Criar testes unitários e de cobertura.

Os testes estão no `tests.py` do módulo `core` do projeto, e cobrem os casos de uso possíveis da API. Eles fazem requisições e comparam os códigos de status para verificar se as requisições estão retornando o que se espera. Para rodar os testes, apenas precisa rodar no terminal o comando `python manage.py test core.tests` estando dentro da pasta do projeto.