RELATÓRIO FINAL DE IMPLEMENTAÇÃO
João Carlos Barbosa dos Santos
Módulo V – Análise e Desenvolvimento de Sistemas

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

Todas as entidades pertencem a um micro sistema de documentação baseado na emissão de Alvarás de Construção, Autos de regularização e Habite-ses. As entidades utilizadas são:
Cidade: Além de herdar de um Model também utiliza choices geradas pela biblioteca *localflavor*, que gera todos os nomes de estado e uf.
Ref.: https://pypi.python.org/pypi/django-localflavor

```
core/models.py
from localflavor.br.br_states import STATE_CHOICES
class Cidade(models.Model):
(...)
    estado = models.CharField(max_length=2,
                              choices=STATE_CHOICES,
                              verbose_name='Estado',
                                      help_text='Informa o estado em que a cidade
pertence')

(...)
```

Bairro: Se relaciona com **Cidade**.
Requerente: Se relaciona com **Bairro.**
Cargo: Modelo que descreve o cargo de um Responsável Técnico.
ResponsavelTecnico: Se relaciona com **Cargo**.
Documento: Modelo que representa um documento genérico. Possui um campo do tipo *PositiveIntegerField* que é validado por uma função (*valida_ano*) que só permite que seja inserido um ano igual ou maior que 2013. Se relaciona com **Bairro, Requerente e ResponsavelTecnico.**

```
core/models.py
ano     =     models.PositiveIntegerField(verbose_name='Ano     do     documento',
validators=[valida_ano],

def valida_ano(valor):
    if valor < 2013:
        raise ValidationError(
            _('%(valor) não pode ser menor do que 2013'),
            params={'valor': valor},
        )
```

AlvaraConstrucao, AutoRegularizacao e HabiteSe: Herdam de Documento e possuem um campo *'numero'* que é validado por uma função que não permite que o número seja menor do que 1. Foi preciso fazer isso porque mesmo declarando um campo como `PositiveIntegerField`, constatei que pela API dava pra cadastrar números negativos. **HabiteSe** se relaciona com **AlvaraConstrucao** e **AutoRegularizacao**.

```
core/models.py
def valida_num(valor):
    if valor < 1:
        raise ValidationError(
            _('%(valor) não pode ser menor do que 1'),
            params={'valor': valor},
        )
```

Atendente: Modelo que se relaciona com o `User` do Django. Possui `'matricula'` como campo extra e `user` como atributo de relação com a classe *`django.contrib.auth.models.User`*.

```
core/models.py
class Atendente(models.Model):
      user     =     models.OneToOneField('auth.User',     related_name='atendente',
on_delete=models.CASCADE, )
      matricula    =    models.CharField(max_length=10,    help_text='Matrícula    do
atendente')

(…)
```

Todos os campos dos modelos possuem o parâmetro *`help_text`* definido pois o isso ajuda na geração da documentação da API pelo `Swagger`.

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

As classes mais importantes foram definidas como acessível somente para usuários autenticados, como Requerente, as classes que herdam de Documento, a classe responsável técnico e Atendente.

A rota para Atendente ainda tem uma limitação extra que só permite que a mesma seja vista ou alterada somente por um usuário cadastrado como superuser.

OBS.: Superuser padrão já cadastrado. Login: admin, Senha: password123

```
core/views.py
class CidadeViewSet(viewsets.ModelViewSet):
    (...)
    permission_classes = (permissions.IsAuthenticatedOrReadOnly,)

class RequerenteViewSet(viewsets.ModelViewSet):
    (...)
    permission_classes = (permissions.IsAuthenticated,)

class UserViewSet(viewsets.ReadOnlyModelViewSet):
   (...)
      # Somente o usuário com a flag is_superuser ativada pode visualizar os
usuários.
    permission_classes = (permissions.IsAuthenticated, permissions.IsAdminUser,)

class AtendenteViewSet(viewsets.ModelViewSet):
    (...)
      # Somente o usuário com a flag is_superuser ativada pode cadastrar e
visualizar novos atendentes.
    permission_classes = (permissions.IsAuthenticated, permissions.IsAdminUser,)
```

```
    (...)
```

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

Como biblioteca de documentação, foi adotado o `Swagger`, que é bastante flexível e personalizável.

Sua rota está disponível em `/documentacao/`. Sua configuração é bastante simples tendo somente que incluir *'rest_framework_swagger'* nas configurações do projeto. No arquivo *urls* da aplicação importar o método *get_swagger_view*, definir o título da página e inserindo na rota padrão.

Como foi adotado ViewSet nas classes de visão, foi preciso criar uma classe ViewSet para o swagger e depois chamá-la na rota padrão.

```
settings.py
EXTRA_APPS = [
    'rest_framework',
    'rest_framework_swagger',
    'django_filters',
]
```

Importações necessárias para criar a classe viewset do swagger.

```
core/view.py
from rest_framework import permissions, viewsets
from rest_framework.response import Response
from rest_framework.schemas import SchemaGenerator
from rest_framework_swagger import renderers

# Criando uma ViewSet para o Swagger, já que o uso de viewset inviabiliza o exibição padrão
# dos urls na 'api-root'. Para isso é preciso transformar uma 'APIView' em uma 'ViewSet'.
# Ref.1: http://marcgibbons.github.io/django-rest-swagger/schema/
# Ref.2: https://stackoverflow.com/questions/30389248/how-can-i-register-a-single-view-not-a-viewset-on-my-router
class SwaggerViewSet(viewsets.ViewSet):
    """
    list:   Retorna a documentação da Api
    """
    permission_classes = [permissions.AllowAny]
    renderer_classes = [
        renderers.OpenAPIRenderer,
        renderers.SwaggerUIRenderer
    ]
    def list(self, request):
        generator = SchemaGenerator()
        schema = generator.get_schema(request=request)
        return Response(schema)

core/urls.py
from rest_framework.routers import DefaultRouter
(...)
router.register(r'documentacao', views.SwaggerViewSet, base_name='swagger')
urlpatterns = [
    url(r'^', include(router.urls)),
```

```
]
```

Para documentar basta usar uma `docstring` logo abaixo da declaração da classe especificando o que cada método faz usando a nomenclatura `'nome_metodo': 'Descrição'`.

```
core/views.py
class CidadeViewSet(viewsets.ModelViewSet):
    """
    list:       Retorna uma lista de todas as cidades cadastradas no sistema.
    read:       Retorna uma cidade.
    create:     Cria uma nova cidade no banco do sistema.
    update:     Atualiza todos os campos.
    partial_update: Atualizar somente os campos alterados.
    delete:     Apaga uma cidade.
    """
(...)
```

Outra opção de documentação bastante interessante é a biblioteca built-in do DRF, que funcionar quase do mesmo modo que o Swagger, com a diferença que não precisa configurar nada no arquivo *`settings.py`*, somente importar o método *`include_docs_urls`*.

```
core/urls.py
from rest_framework.documentation import include_docs_urls
urlpatterns = [
    ...
    url(r'^docs/', include_docs_urls(title='My API title'))
]
```

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

A paginação foi configurada no arquivo *`settings.py`* e foi definida com um limite de 10 itens por página. O Throttling também foi configurado e com as opções de 50 requisições por hora para usuários anônimos e 200 requisições por hora para usuários autenticados.

```
settings.py
REST_FRAMEWORK = {
    # Ativando a paginação
    'DEFAULT_PAGINATION_CLASS':
        'rest_framework.pagination.LimitOffsetPagination',
    'PAGE_SIZE': 10,
    (...)
    # Ativando o Throttle
    'DEFAULT_THROTTLE_CLASSES': [
        'rest_framework.throttling.AnonRateThrottle',
        'rest_framework.throttling.UserRateThrottle',
    ],
    'DEFAULT_THROTTLE_RATES': {
        'anon': '50/hour',
        'user': '200/hour',
    },
}
```

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

Para filtros foi utilizado a biblioteca *'djangorestframework-filters'* no lugar do *django-filters*.
O `django-rest-framework-filters`, é uma biblioteca do Django REST framework e do Django-filter que facilita o filtro em todos os relacionamentos. Na variável `'fields'`, é possível especificar o tipo de argumento que será feita na consulta, como `'exacts'`, `'in'`, `'startswith'` e todas as outras opções dos field-lookups do Django. Esses argumentos precisam ser checados, pois podem ser diferentes de acordo com o banco de dados utilizado.
Ref.1: https://docs.djangoproject.com/pt-br/1.11/ref/models/querysets/#field-lookups
Ref.2: https://github.com/philipn/django-rest-framework-filters

```
settings.py
EXTRA_APPS = [
    (...)
    'django_filters',
]

REST_FRAMEWORK = {
    (...)
    'DEFAULT_FILTER_BACKENDS': [
        'rest_framework_filters.backends.DjangoFilterBackend',
    ],
    (...)
}

core/views.py
import rest_framework_filters as filters

class CidadeFilter(filters.FilterSet):
    class Meta:
        model = Cidade
        fields = {'nome_cidade': ['icontains'],
                  'estado': ['exact']
                  }
class BairroFilter(filters.FilterSet):
                cidade    =    filters.RelatedFilter(CidadeFilter,    name='cidade',
queryset=Cidade.objects.all())
    class Meta:
        model = Bairro
        fields = {'nome_bairro': ['icontains']}
```

Para se criar um filter basta importar o módulo e criar a classe correspondente ao modelo que se quer pesquisar, como no exemplo acima.
Para uso com ViewSet é preciso criar classes do tipo FitlerSet. Depois basta declarar a variável *filter_class* e passar a classe filtro.

```
core/views.py
class CidadeViewSet(viewsets.ModelViewSet):
(...)
    filter_class = CidadeFilter
(...)
```

Criar testes unitários e de cobertura.
Os testes estão no tests.py do projeto. Bastando rodar o comando python manager.py test core.tests para executá-los.

Informações adicionais.
ViewSet

Usar viewset no lugar das classes genéricas do DRF trás uma grande facilidade e redução na quantidade de códigos implementados, já que as viewsets geram as 'LISTS' e 'DETAILS' por padrão, sem contar que com isso ganhamos os `'routers'` automaticamente.

As rotas foram definidas automaticamente com o `DefaultRouter` do DRF. Isto é possível porque estou usando viewsets nas views. Isso facilita, pois as urls são geradas automaticamente, não precisando repetir código e usando o `'defaultrouter'` eu ganho o api-root.
Ref.1: http://www.django-rest-framework.org/api-guide/routers/#routers
Ref.2: http://www.django-rest-framework.org/api-guide/routers/#defaultrouter
Ref.3: http://www.tomchristie.com/rest-framework-2-docs/tutorial/6-viewsets-and-routers

TokenAuthentication

A implementação da autenticação por token foi feita com a biblioteca `rest_framework.authtoken`, que já vem embutida no DRF.
Ref.: http://www.django-rest-framework.org/api-guide/authentication/

```
settings.py
EXTRA_APPS = [
      ...
    'rest_framework.authtoken',
      ...
]

REST_FRAMEWORK = {
    (...)
    'DEFAULT_AUTHENTICATION_CLASSES': [
        # Ativando a autenticação por token
        'rest_framework.authentication.TokenAuthentication',
        # 'rest_framework.authentication.BasicAuthentication',
        'rest_framework.authentication.SessionAuthentication',
    ],

    (...)

core/urls.py
from rest_framework.authtoken.views import obtain_auth_token

urlpatterns = [
(...)
    url(r'^api-token-auth/$', obtain_auth_token),
(...)
]
```

```
core/models.py
@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)
```

Cross-Origin Resource Sharing

O Envio do cabeçalho de proteção `cors` é feito com a biblioteca `django-cors-headers`. É fácil de usar e configurar.
O CorsMiddleware deve ser colocado o mais alto possível, especialmente antes de qualquer middleware que possa gerar respostas, como o CommonMiddleware do Django ou o WhiteNoiseMiddleware do Whitenoise. Se não for anterior, não poderá adicionar os cabeçalhos CORS a essas respostas.
Ref.: https://github.com/ottoyiu/django-cors-headers/

```
settings.py
EXTRA_MIDDLEWARE = [
    'corsheaders.middleware.CorsMiddleware',
]
MIDDLEWARE = EXTRA_MIDDLEWARE + DEFAULT_MIDDLEWARE

settings.py
CORS_ORIGIN_ALLOW_ALL = False
CORS_ORIGIN_WHITELIST = (
    'localhost:8000',
)
```

Setando a opção CORS_ORIGIN_ALLOW_ALL para True, permite o acesso de todos os hosts a API.