Documentatie backend
Persoonlijkheidstest

Jens Weyen 19/04/2021

Verantwoording Django

Voor de ontwikkeling van onze backend gebruiken wij Django, een python framework dat snel en efficiënt websites/API's kan bouwen. Django erft de sterktepunten van Python over en is dus zeer snel, gemakkelijk om te leren en heeft de mogelijkheid om cutting-edge technologieën zoals machine learning en artificial intelligence te implementeren.

Django bleek dus de geschikte kandidaat te zijn voor onze back end, overigens werd er van ons ook verwacht onze applicatie te integreren in een andere applicatie die in de backend gebaseerd was op Django.

Doel backend

Het doel van onze backend is om onze frontend te voorzien van de vereiste data en om de data te beheren, concreet zullen we dit op 2 manieren aanpakken:

- Admin tool
- API

Dit document zal de backend documentatie van het persoonlijkheidstest project inhouden en verwijst dus naar het plan van aanpak van het persoonlijkheidstest project.

Admin tool

De admin tool zal enkel gebruikt worden door de administrator(s) van de applicatie, deze zal wegens veiligheidsredenen dus niet opengesteld worden aan de normale gebruikers van de persoonlijkheidstest.

API

De API of Application Programming Interface van onze applicatie zal instaan voor de communicatie tussen de frontend en backend van onze applicatie. Het API deel van de backen zal dus data verzenden op aanvraag van onze

frontend, dit zal gebeuren via GET & POST requests. PUT & DELETE requests zullen niet gebruikt worden door onze frontend.

Python

Aangezien ons framework voornamelijk uit python bestaat is het niet onbelangrijk welke versie van python we gaan gebruiken. De huidige applicatie Jobsite.hr gebruikt momenteel python 2.7, het is van groot belang dat wanneer we de huidige applicatie gaan uitbreiden met onze persoonlijkheidstest dat onze python versie overeenkomt en dat al onze modules compatibel zijn met python 2.7. Daarom hebben wij de opdracht gekregen om onze applicatie te ontwikkelen met python 2.7, hierdoor zullen we minder problemen krijgen bij de deployment van onze applicatie.

Python modules

Onze backend zal verschillende modules nodig hebben om correct te kunnen functioneren, deze modules zullen gebruikt worden om bijvoorbeeld connectie te kunnen maken met onze PostgreSQL server.

Lijst van de belangrijkste modules en versies:

- Django==1.11.17
- django-cors-headers==3.0.0
- django-modeltranslation==0.12.2
- psycopg2==2.8.6

Django 1.11.17

Deze module is het hart van ons framework, deze module zal zo goed als overal in onze applicatie gebruikt worden en is dus van groot belang. We kozen voor Django 1.11.17 omdat dit zal zorgen voor een simpele integratie naar de andere applicatie die ook Django 1.11 gebruikt. Dit was overigens een vereiste van de klant.

Django-cors-headers 3.0.2

CORS of Cross-origin resource sharing is een mechanisme waarmee beperkte bronnen op een webpagina kunnen worden opgevraagd vanuit een ander domein buiten het domein van waaruit de eerste bron werd geserveerd. Het Django framework declareert een aantal CORS regels, om communicatie tussen onze front -en backend te maken moeten we deze aanpassen. Het aanpassen van deze regels wordt geregeld door deze module. We hebben gekozen voor versie 3.0.2 omdat deze de nieuwste versie is die compatibel is met zowel Django 1.11 als Python 2.7. Nieuwe versies van deze module zijn alleen compatibel met Python 3.X . Deze module zal enkel gebruikt worden tijdens de ontwikkeling van onze applicatie en zal verwijdert worden voor het einde van ons project.

Django-modeltranslation 0.12.2

Django-modeltranslation is een module die wordt gebruikt om onze models van onze applicatie automatisch aan te passen. Deze module zal er bijvoorbeeld voor zorgen dat wanneer we een extra taal willen toevoegen aan onze applicatie dat er met 1 regel code de nodige velden extra zullen worden aangemaakt in onze modellen zonder we er iets extra voor moeten doen.

Psycopg2 2.8.6

Om data uit te wisselen met onze frontend moeten we natuurlijk ook verbinding kunnen maken met onze PostgreSQL server, dit zal gebeuren door het gebruik van de Psycopg2 module. Op het moment van schrijven is versie 2.8.6 de allernieuwste versie, deze versie is compatibel met zowel Python 2.7 als Django 1.11

PostgreSQL

Onze applicatie zal een databank nodig hebben, wij gebruiken een PostgreSQL server omdat de applicatie waar we verder op moeten bouwen dit ook gebruikt. Dit zal dus zorgen voor een zeer simpele integratie in de andere applicatie.

Projectstructuur

In dit hoofdstuk van de documentatie zullen we over de belangrijkste folders/bestanden gaan van onze Django backend, hierin bespreken we wat deze bestanden in de folder werkelijk doen met als doel een beter overzicht van onze applicatie te krijgen.

Mappenstructuur

- Persoonlijkheidstestproject
    - assessment
        - mysite
            - settings.py
            - urls.py
        - Persoonlijkheidstest
            - urls.py
            - migrations
            - admin.py
            - models.py
            - translation.py
            - views.py
        - persoonlijkheidstest - Vue (zie documentatie Vue)

Assessment

In deze folder vindt heel ons project plaats, deze folder bevat dus zowel onze backend als onze frontend. In dit hoofdstuk zullen enkel de bestanden/folders met betrekking tot onze backend besproken worden.

Mysite

Deze folder bevat voornamelijk bestanden die automatisch door Django aangemaakt zijn tijdens de initialisatie van het project. Enkele bestanden uit deze folder zijn wel belangrijk voor de verdere configuratie van ons project

Settings.py

In dit bestand kunnen we de algemene instellingen vinden van ons project, deze instellingen zijn hoofdzakelijk auto-generated en worden niet amper gewijzigd tijdens de ontwikkeling van ons project.
INSTALLED_APPS: Zoals we kunnen zien in de screenshot staan er verschillende zaken in de lijst, wij hebben daar 3 dingen aan toegevoegd: 'Corsheaders', 'modeltranslation' en 'persoonlijkheidstest' deze zaken zorgen er voor dat de modules Django-cors-headers en Django-modeltranslation werken. Het zorgt ook voor de integratie van de persoonlijkheidstest folder.

```
INSTALLED_APPS = [
    'corsheaders',
    'modeltranslation',
    'django.contrib.admin',
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.messages',
    'django.contrib.staticfiles',
    'persoonlijkheidstest',

]
```

Urls.py

Door het aan te passen van dit bestand kunnen we voor verschillende routes zorgen m.a.w. dit bestand zorgt er voor dat wanneer iemand een url oproept ze naar de juiste pagina worden doorverwezen. In dit bestand zorgen we er voor dat het urls.py bestand van de map persoonlijkheidstest geïmporteerd zal worden.

Persoonlijkheidstest

In deze folder vinden we onze werkelijke applicatie die we gebouwd hebben, de bestanden in deze folder zullen zorgen dat onze backend data kan versturen naar onze database en frontend.

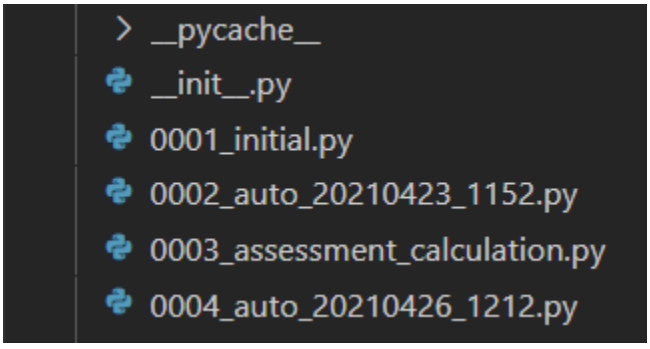
Urls.py

In dit bestand zullen we de routing van onze applicatie behandelen, hier zullen enkele routes gedefinieerd worden, een voorbeeld van dit bestand kan u in de screenshot hieronder vinden:

```python
from django.conf.urls import url
from django.contrib import admin
from persoonlijkheidstest.views import index,QuestionsView,AssessmentsView,TypesView,TextTranslationView
from django.conf.urls.i18n import i18n_patterns
from django.views.i18n import set_language
from django.contrib.auth.decorators import login_required
from django.views.decorators.csrf import csrf_exempt

urlpatterns = [
    url('api/admin/', admin.site.urls),
    url('api/questions/',QuestionsView.as_view()),
    url('api/assessments/',csrf_exempt(AssessmentsView.as_view())),
    url('api/types/',TypesView.as_view()),
    url('api/textTranslation/',TextTranslationView.as_view()),
    url('', index),

]
```

Migrations

Ons systeem genereert migrations, dit zijn bestanden die onze modellen in de postgreSQL server aanmaken, deze bestanden worden in de migrations folder. Hieronder kan u een voorbeeld vinden van hoe deze folder er uit ziet.



Elk van deze bestanden is verantwoordelijk voor een bepaalde aanpassing aan de postgreSQL server, een voorbeeld van zo'n bestand kan u hieronder vinden. (0004_auto_20210426_1212.py)

```python
# -*- coding: utf-8 -*-
# Generated by Django 1.11.26 on 2021-04-26 12:12
from __future__ import unicode_literals

from django.db import migrations


class Migration(migrations.Migration):

    dependencies = [
        ('persoonlijkheidstest', '0003_assessment_calculation'),
    ]

    operations = [
        migrations.AlterModelOptions(
            name='question',
            options={},
        ),
    ]

```

Admin.py

In dit bestand zal de registratie van onze modellen gebeuren, dit zal er voor zorgen dat we CRUD operaties kunnen doen voor deze modellen in de automatisch gegenereerde admin kant van de Django backend. Hieronder

kan u de inhoud van dit bestand vinden.

```python
from django.contrib import admin
from .models import Application,Question,Assessment,Type

# Register your models here.

admin.site.register(Application)
admin.site.register(Question)
admin.site.register(Assessment)
admin.site.register(Type)
```

Models.py

Het aanmaken van onze models gebeurt in models.py, hierin kan u de 3 modellen van onze backend vinden. Elk van deze modellen heeft zijn attributen en de declaratie hiervan gebeurt ook in dit bestand. We hebben 3 verschillende modellen in onze applicatie:

1. Application
2. Question
3. Assessment
4. Type

Application model

Dit model zal de informatie van de gebruiker opslaan, dit model zal concreet de voor -en achternaam, email en geboortedatum van de gebruiker opslaan.

Question model

Het question model zal alle vragen van onze persoonlijkheidstest opslaan, elke instantie van dit model zal een vraag en twee antwoorden bevatten in verschillende talen. Lijst alle vragen: NL, FR, EN, DE

Assessment model

Na dat de gebruiker de test heeft afgenomen worden de resultaten hiervan opgeslagen in het assesment model, in dit model worden alle antwoorden op de vragen van de gebruiker opgeslagen samen met het persoonlijkheidstype van de gebruiker en de berekeningen hiervan. Het object zal ook een verwijzing naar de gebruiker bevatten.

Type model

Wanneer een gebruiker een test af heeft gelegd dan wordt hij doorverwezen naar een resultaatspagina op deze pagina zal de gebruiker zijn resultaat kunnen vinden maar omdat het resultaat van zo'n test (bv. "ENTJ") niet veel betekent zullen we extra tekst over elk persoonlijkheidstype moeten voorzien. Dit is waar ons Type model voor dient, het voorziet extra informatie rond onze persoonlijkheidstypen en dat in 4 verschillende talen.

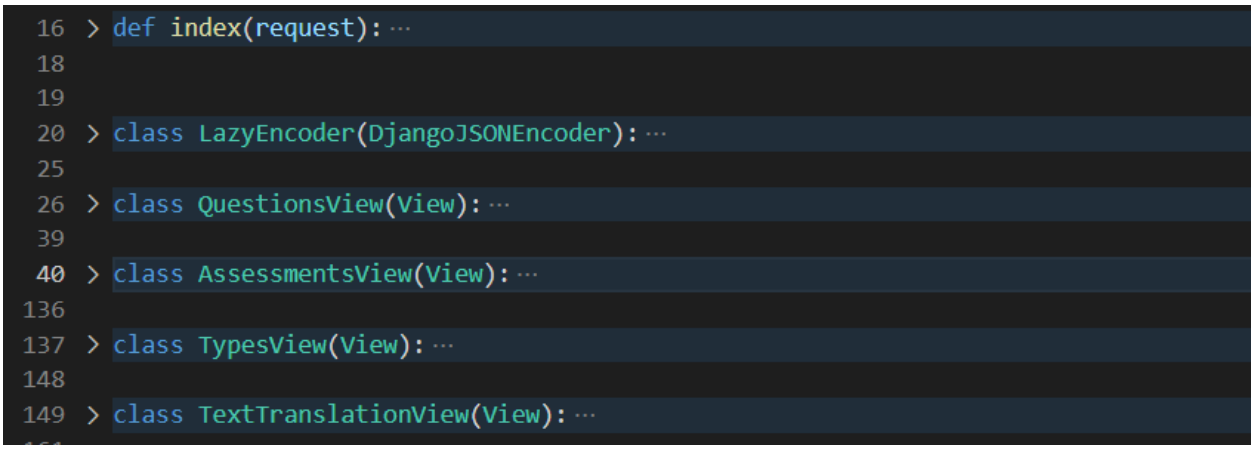
Translation.py

In dit bestand zal onze modeltranslation module toegepast worden op bepaalde modellen. Dit bestand zal onze modellen aanpassen door alle gekozen attributen in de in settings.py gedefinieerde talen te toe te voegen. In ons geval zal dit bijvoorbeeld op basis van ons question model het attribuut question nemen, het zal dan een attribuut toevoegen aan ons question model genoemd question_fr. Zo kunnen we dus voor een dynamische hoeveelheid talen zorgen zonder dat we deze attributen handmatig zouden moeten aanmaken. Hieronder kan u een screenshot vinden van translation.py.

```python
from modeltranslation.translator import translator, TranslationOptions
from .models import Question,Type


class QuestionTranslationOptions(TranslationOptions):
    fields = ('question','answer_a','answer_b')

translator.register(Question, QuestionTranslationOptions)

class TypeTranslationOptions(TranslationOptions):
    fields = ('full_kts','text','temperament')

translator.register(Type,TypeTranslationOptions)
```

Views.py

In het views.py bestand zullen we er voor zorgen dat onze data opgeroepen en aangemaakt kan worden door het gebruik van een API-requests. In dit bestand kan u verschillende klassen vinden die zullen worden gebruikt om deze API-requests mogelijk te maken. Hieronder kan u een screenshot vinden van de klassen die zich in dit bestand bevinden.



Handleiding

In dit hoofdstuk zullen we de admin kant van onze applicatie overlopen maar ook hoe onze API werkt vanuit het standpunt van onze frontend.

Opstarten backend

Het opstarten van onze backend kan door de volgende commando uit te voeren in de root folder van ons project: python manage.py runserver

Dit zal wanneer je omgeving goed geconfigureerd is de backend opstarten.

Admin

Onze backend heeft dankzij Django een automatisch geïntegreerde managing tool, deze tool maakt het mogelijk om objecten te bekijken, aan te maken, aan te passen en te verwijderen. Toegang tot de admin: http://127.0.0.1:8000/api/admin/

Inloggen

Om toegang te krijgen tot de admin kant van onze backend moeten we ten eerste administrators aanmaken, dit kan enkel gebeuren via een command in de python omgeving. Deze command zal dan enkele gegevens vragen. Na het aanmaken van de user kan deze zich inloggen op de admin kant van onze applicatie. Het aanmaken van een administrator gebeurt zo:

```
(venv) C:\Stage\assessment>python manage.py createsuperuser
Username: Handleiding
Email address: Handleiding@handleiding.be
Password:
Password (again):
Superuser created successfully.

(venv) C:\Stage\assessment>
```

Nadat er een administrator toegevoegd is dan kunnen we ons aanmelden bij de admin view, dit zal gebeuren via het gebruik van de door jou opgegeven username en wachtwoord.

Wachtwoord wijzigen

Wanneer een administrator zijn wachtwoord vergeten is kan dit wachtwoord gewijzigd worden volgens deze command:

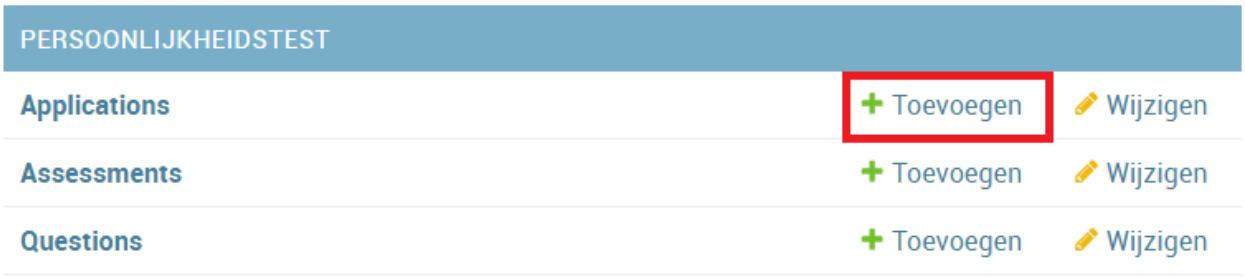
```
(venv) C:\Stage\assessment>python manage.py changepassword Handleiding
Changing password for user 'Handleiding'
Password:
Password (again):
Password changed successfully for user 'Handleiding'
```

Django-beheer

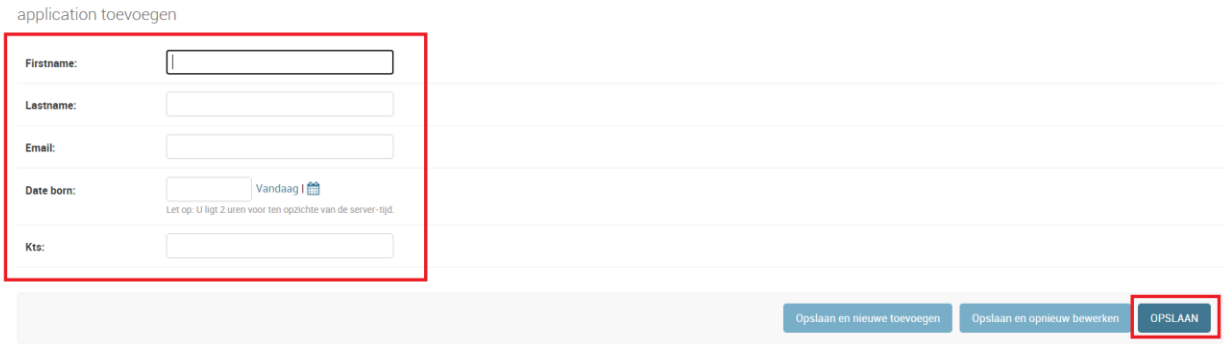
Nadat de administrator zich succesvol heeft aangemeld verwijst de applicatie de administrator naar de beheerpagina. Op deze beheerpagina zullen alle in Admin.py geregistreerde modellen beheerd kunnen worden. Deze pagina zal ook wijzigingen tot de authenticatie en autorisatie toelaten. De pagina ziet er zo uit:

Aanmaken van een object

Een administrator kan van eender welk model een object aanmaken door op de “Toevoegen” knop te drukken als voorbeeld zullen we nu een Applications object aanmaken.

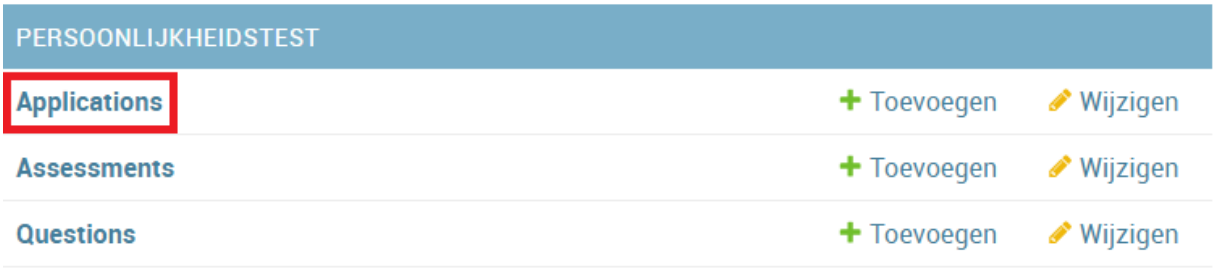


Vervolgens krijgt de administrator een scherm te zien waar hij alle informatie van het object kan invullen. Vervolgens klikt de administrator op “Opslaan” aan, en wordt hij verwezen naar de lijst van objecten.



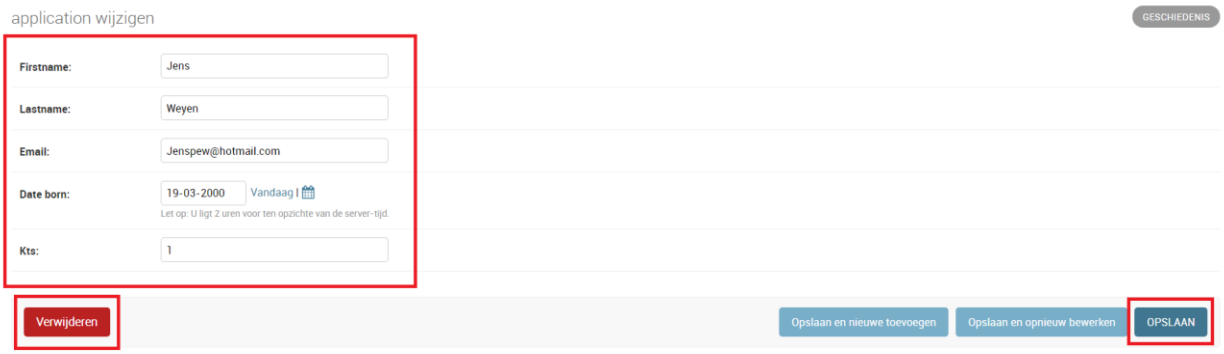
Aanpassen/Verwijderen van een object

Nadat we een object hebben aangemaakt zullen we demonstreren hoe we een object kunnen aanpassen. We beginnen terug bij de beheer pagina en klikken op de naam van ons model



Vervolgens zien we een overzicht van alle Applications-objecten en klikken we op ons object:

Nadat we op ons object geklikt hebben krijgen we een overzicht te zien van de gegevens van dat object. Op deze pagina kunnen we alle gegevens aanpassen en opslaan door op de knop rechtsonder te klikken. We kunnen het object ook verwijderen door op de rode knop linksonder te drukken.



API

Onze backend zal ook enkele API endpoints openstellen zodat we onze frontend kunnen linken aan onze backend. Deze API voorziet voor elk model in ons Models.py bestand GET en POST endpoints, de endpoints zullen enkel lokaal opengesteld worden. GET requests staan in voor het ophalen van onze data en POST requests zullen verantwoordelijk zijn voor het aanmaken van objecten.

PUT en DELETE endpoints werden in het begin van ons project ook geïmplementeerd maar deze zijn overbodig en werden niet meer up-to-date gehouden, wanneer we deze zouden willen gebruiken staan deze zo goed als

klaar mits enkele aanpassingen waar nodig. We besloten om deze niet meer te onderhouden omdat nadat we deze ontwikkeld hadden we te horen kregen dat we deze niet gingen gebruiken.

Los van deze endpoints zal er ook een endpoint voorzien worden om alle vra
De code van de endpoints kan teruggevonden worden in Views.py

Questions

De GET-request voor al onze Questions vereist een parameter genoemd "lang", deze parameter zorgt er voor dat je alle Questions krijgt in één van de 4 geïmplementeerde talen (NL,EN,DE,FR).

De code die deze GET-request mogelijk maakt:

```
class QuestionsView(View):
    def get(self, request):
        body =request.GET
        activate(body['lang'])
        data = Question.objects.all()
        data = serialize('json',data , cls=LazyEncoder, fields=('question','answer_a','answer_b'))
        return JsonResponse(json.loads(data),safe=False)
```

Voorbeeld GET-request : http://127.0.0.1:8000/api/questions?lang=en

Resultaat:


```
{
    "model": "persoonlijkheidstest.question",
    "pk": 70,
    "fields": {
        "question": "Do you tend to be more",
        "answer_a": "deliberate than spontaneous",
        "answer_b": "spontaneous than deliberate"
    }
},
```

Assessments:

De GET-request voor ons Assessment model vereist een parameter genoemd “app_id”, deze parameter zorgt er voor dat je alle assessments krijgt van één application.

De code die deze GET-request mogelijk maakt:

```
def get(self, request):
    body =request.GET
    data = Assessment.objects.filter(application__id=body['app_id']).order_by('-created')
    data = serialize('json',data , cls=LazyEncoder)
    return JsonResponse(json.loads(data),safe=False)
```

Voorbeeld GET-request : http://127.0.0.1:8000/api/assessments/?app_id=2

Resultaat:


```
{
    "model": "persoonlijkheidstest.assessment",
    "pk": 29,
    "fields": {
        "application": 1,
        "kts_type": "ISTJ",
        "data": { … }, // 70 items
        "calculation": { … }, // 12 items
        "created": "2021-04-29T13:18:01.886Z",
        "last_updated": "2021-04-29T13:18:01.886Z"
    }
}
```

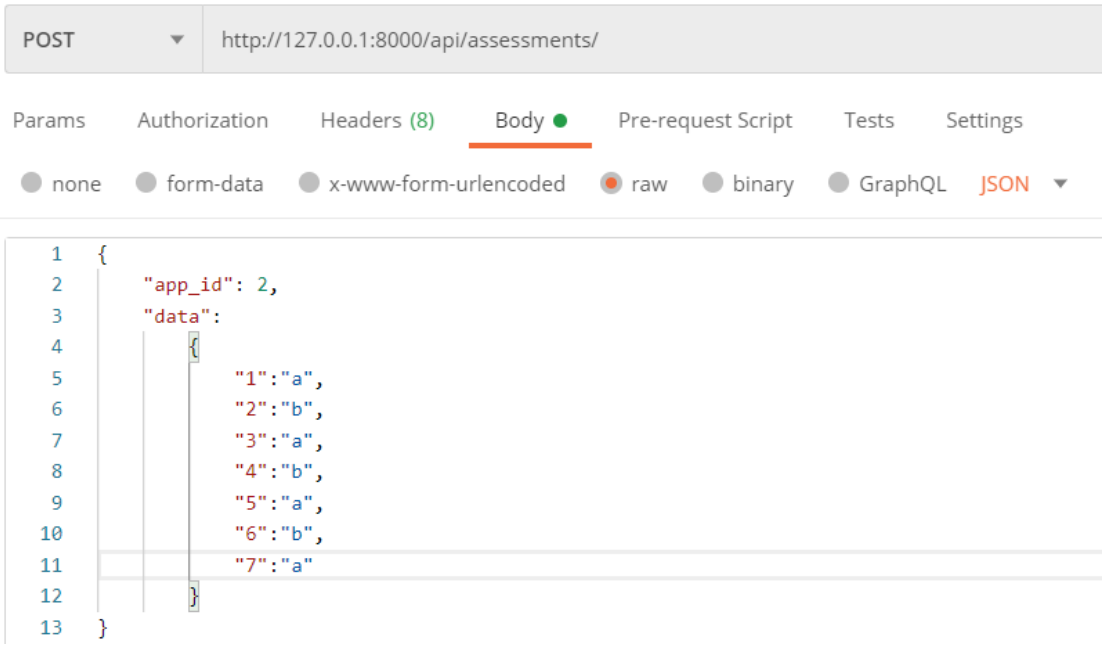
De POST-request voor een Assessment aan te maken zal data nodig hebben om een vraag aan te maken, er zal dus een hoop data meegegeven moeten worden via de body van deze request.

De code die deze POST-request mogelijk maakt:

```python
    def post(self, request, *args, **kwargs):
        body = json.loads(request.body)
        questions= Question.objects.all()
        a = Assessment(
                kts_type="",
                application=Application.objects.get(pk=body.get("app_id")),
                data= body.get("data")
            )
        answers = {}
        E = 0
        I = 0
        S = 0
        N = 0
        T = 0
        F = 0
        J = 0
        P = 0
        kts_string = ""
        for x in body.get("data"):
            answers[x]=body.get("data")[x]
            test = questions.filter(pk=int(x))
            if str(test[0].type)[0] == "E":
                if body.get("data")[x] == "a":
                    E = E + 1
                else:
                    I = I + 1
            elif str(test[0].type)[0] == "S":
                if body.get("data")[x] == "a":
                    S = S + 1
                else:
                    N = N + 1
            elif str(test[0].type)[0] == "T":
                if body.get("data")[x] == "a":
                    T = T + 1
                else:
                    F = F + 1
            elif str(test[0].type)[0] == "J":
                if body.get("data")[x] == "a":
                    J = J + 1
                else:
                    P = P + 1
```

```python
        if E >= I:
            kts_string += "E"
        else:
            kts_string += "I"

        if S >= N:
            kts_string += "S"
        else:
            kts_string += "N"

        if T >= F:
            kts_string += "T"
        else:
            kts_string += "F"

        if J >= P:
            kts_string += "J"
        else:
            kts_string += "P"
        a.kts_type = kts_string
        a.save()
        ap = Application.objects.get(pk=body.get("app_id"))
        ap.kts = kts_string
        ap.save()
        res = {
            'success': True,
            'messsge': 'Class based view: api to create, created assesment with id: '+str(a.id),
            'type': a.kts_type,
        }
        return JsonResponse(res)
```

Voorbeeld POST-request (Postman):



Resultaat:

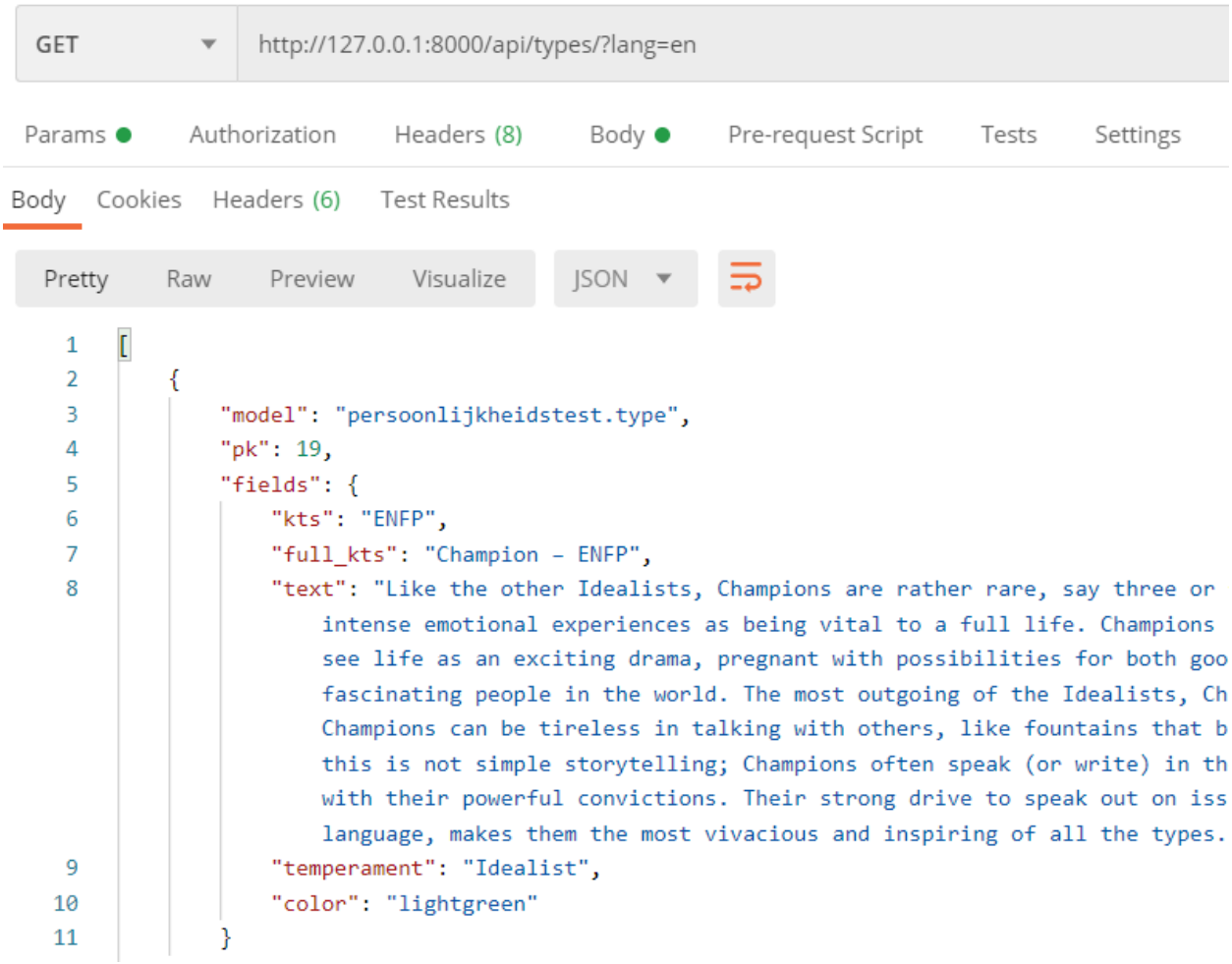
{"success": true, "messsge": "Class based view: api to create, created assesment with id: 30", "type": "ESTJ"}


```
{
    "model": "persoonlijkheidstest.assessment",
    "pk": 30,
    "fields": {
        "application": 2,
        "kts_type": "ESTJ",
        "data": {…
        },
        "calculation": {…
        },
        "created": "2021-04-30T11:37:41.164Z",
        "last_updated": "2021-04-30T11:37:41.164Z"
    }
}
```

Types

De GET-request voor al onze Types vereist een parameter genoemd “lang”, deze parameter zorgt er voor dat je alle Questions krijgt in één van de 4 geïmplementeerde talen (NL,EN,DE,FR).

Voorbeeld GET-request:



Integratie frontend

Onze applicatie bestaat uit 2 delen, de front -en backend. Om deployment van onze applicatie simpeler te maken hebben we gekozen om onze backend en frontend samen te smelten. Dit wil zeggen dat Django alle requests van zowel back als frontend zal behandelen. We zullen dit bereiken door onze frontend op voorhand te laten compileren en als “template” te gebruiken in onze backend. Meer informatie over de frontend kan u vinden in de documentatie van de frontend

Frontend Compileren

Als we wijzigingen willen aanbrengen aan onze frontend dan zullen we nadat we deze wijzigingen hebben gedaan genoodzaakt zijn om onze code opnieuw te laten compileren zodat deze gebruikt zal kunnen worden als template door onze backend.

Het compileren van onze frontend gebeurt door het invoeren van “npm run build” in de folder persoonlijkheidstest-Vue. Hierdoor zal onze frontend automatisch updaten.