PYDANTİC :

Linter : Programlama dillerinde yazılan kodun hatalarını ve potansiyel sorunlarını tespit eden, kodun stil kurallarına uyumluluğunu kontrol eden bir araçtır. Linter'lar, kodun belirli bir stil rehberine uygun yazılmasını sağlamak, kod kalitesini artırmak ve hataları erken aşamada tespit etmek amacıyla kullanılır.

Bir linter'ın gerçekleştirebileceği bazı kontroller şunlardır:

- **Sözdizimi hataları**: Dilin sözdizimine uygun olmayan yapıları tespit eder.
- **Stil kuralları**: Kodun belirli bir stil rehberine (örneğin, Google'ın JavaScript stil rehberi) uygun olup olmadığını kontrol eder.
- **Değişken kullanımı**: Tanımlanmamış değişkenlerin kullanımı, kullanılmayan değişkenler gibi sorunları tespit eder.
- **Yapısal sorunlar**: Potansiyel hatalara yol açabilecek yapısal problemleri bulur (örneğin, sonsuz döngüler, yanlış yerleştirilmiş parantezler).

Bazı popüler linter araçları şunlardır:

- **ESLint**: JavaScript ve diğer ilgili diller için yaygın olarak kullanılan bir linter.
- **Pylint**: Python için yaygın bir linter.
- **Rubocop**: Ruby için kullanılan bir linter.
- **Checkstyle**: Java için kullanılan bir linter.

Linter kullanımı kodun okunabilirliğini artırır, ekip içi kod standartlarını korur ve yazılım geliştirme sürecinde hataları erken tespit ederek daha kaliteli bir kod tabanı oluşturulmasına yardımcı olur.

Django projelerinde Pylint kurulumu:

Pylint-django pluginide Django özelindeki iyileştirmeler, uyarılar için kullanılabilir.

1.Gerekli paketleri kurunuz.

```
3\Desktop\django-pylint> pip install django pylint
```

Pydantic extantion(Linter mantığı):

PyCharm ve VSCode için Pydantic Extension Kullanımı

Pydantic: Python'da veri doğrulama ve serileştirme için kullanılan bir kütüphanedir.

PyCharm için Pydantic Eklentisi:

Pydantic Eklentisi : Pydantic modelleri için otomatik tamamlama, doğrulama ve hata ayıklama gibi özellikler sağlar. PyCharm içinde Pydantic modelleri ile çalışmayı daha verimli hale getirmektedir.

PyCharm ve VSCode için Pydantic eklentileri, Pydantic modelleri üzerinde linter işlevselliği sağlayarak geliştiricilere yardımcı olur. Bu linter işlevleri arasında şunlar bulunur:

- **Otomatik Tamamlama:** Model alanlarının ve yöntemlerinin otomatik tamamlanması.
- **Tip Denetimi:** Model alanlarının ve girdilerinin doğru tipte olup olmadığını kontrol etme.

- **Doğrulama:** Model tanımlamalarının ve doğrulama kurallarının doğru şekilde yapıldığını kontrol etme.
- **Hata Ayıklama:** Kod yazımı sırasında hataları ve stil ihlallerini anında bildirme ve hızlı düzeltme önerileri sunar.

Gereksinimler

- Python 3.7+
- PyCharm veya VSCode
- İnternet bağlantısı (eklenti yüklemek için)

Kurulum

1. **PyCharm'ı açın:** PyCharm'ı başlatın ve ana ekrana gelin.
2. **Eklenti Yöneticisini Açın:** Üst menüden File > Settings (Windows/Linux) veya PyCharm > Preferences (macOS) menüsüne gidin.
3. **Eklentileri Yükleyin:** Sol menüde Plugins sekmesini seçin ve Marketplace sekmesine geçin. Arama çubuğuna pydantic yazın.
4. **Pydantic Eklentisini Yükleyin:** Arama sonuçlarından Pydantic eklentisini bulun ve Install butonuna tıklayın.
5. **PyCharm'ı Yeniden Başlatın:** Eklentiyi yükledikten sonra, PyCharm'ı yeniden başlatın.

VSCode için Pydantic Eklentisi

Kurulum

1. **VSCode'u açın:** VSCode'u başlatın ve ana ekrana gelin.
2. **Eklenti Yöneticisini Açın:** Sol kenar çubuğundan Extensions (Kısayol: Ctrl+Shift+X veya Cmd+Shift+X (macOS)) simgesine tıklayın.
3. **Eklentileri Yükleyin:** Arama çubuğuna pydantic yazın.
4. **Pydantic Eklentisini Yükleyin:** Arama sonuçlarından Pydantic eklentisini bulun ve Install butonuna tıklayın.
5. **VSCode'u Yeniden Başlatın:** Eklentiyi yükledikten sonra, VSCode'u yeniden başlatın.

Özellikler

- **Otomatik Tamamlama:** Pydantic modelleri için otomatik tamamlama sağlar.
- **Tip Denetimi:** Model alanları için tip denetimi ve doğrulama yapar.
- **Hata Ayıklama:** Kod yazarken hataları ve stil ihlallerini anında bildirir ve hızlı düzeltme önerileri sunar.
- **Json Şeması Oluşturma**:Pydantic modellerinden JSON şeması oluşturma

Django Pydantic Best Pratics :

Django ile Pydantic'i entegre etmek, uygulamalarınızda veri doğrulama süreçlerini standartlaştırmanızı ve güçlendirilmesini sağlar. Pydantic, güçlü tip denetimi, otomatik veri dönüşümü ve karmaşık yapıların kolay yönetimi gibi özellikleri sayesinde özellikle API geliştirme sürecinde fayda sağlamaktadır.. İşte Django ile Pydantic kullanırken uygulayabileceğiniz bazı en iyi uygulamalar ve örnekler:

1. Pydantic Modelinin Doğrulanması

- Django model verileriniz için Pydantic `BaseModel` sınıfları oluşturun.
- Pydantic'in veri dönüşüm özelliklerinden yararlanarak, JSON gibi karmaşık veri yapılarını kolayca yönetin.

```
from pydantic import BaseModel
class UserModel(BaseModel):
    username: str
    email: str
    is_active: bool
```

2. API Doğrulama ve Serileştirme

API geliştirirken, Django Rest Framework'ün `serializer`lar yerine Pydantic modellerini kullanın. Bu, daha temiz ve modüler bir API kod tabanı sağlar.

- API view'larında doğrudan Pydantic modellerini kullanarak verileri doğrulayın ve serileştirin.
- Hata yakalama yapısını kullanarak, doğrulama hatalarını uygun HTTP hata kodları ile işleyin. `from django.http import JsonResponse`

```
from rest_framework.decorators import api_view
from pydantic import ValidationError

@api_view(['POST'])
def create_user(request):
    try:
        user = UserSchema(**request.data)
    except ValidationError as e:
        return JsonResponse({'errors': e.errors()}, status=400)
    # Veritabanı kaydı vs.
```

```
    return JsonResponse({'message': 'User created successfully'}, status=201)

```

3. Asenkron İşlemler

Django'nun asenkron yeteneklerini kullanarak, Pydantic model doğrulama işlemlerini asenkron fonksiyonlar içinde yapın. Bu, özellikle ağır iş yükleri için performansı artırabilir.

```
from asgiref.sync import sync_to_async

@api_view(['POST'])
async def create_user(request):
    user_data = await sync_to_async(UserSchema.parse_obj)(request.data)
    # Asenkron veritabanı kaydı vs.
    return JsonResponse({'message': 'User created successfully'}, status=201)
```

- Django 3.1 ve sonrası için `async def` ile asenkron view'lar oluşturun.
- Pydantic modelinizle asenkron olarak etkileşime geçin.

4. Performans Optimizasyonu

Pydantic kullanırken, modelinizdeki bazı doğrulayıcıların performansı etkileyebileceğini göz önünde bulundurun.

- Doğrulama sürecini ihtiyaca göre ayarlayın; her durumda tüm alanları doğrulamak yerine gerektiğinde spesifik alanları doğrulayın.
- Gereksiz yere her seferinde modeli yeniden oluşturmak yerine, doğrulanmış verileri önbelleğe almayı düşünün.

5. Test Süreçleri

Pydantic modellerini kullanarak, model doğrulama işlemlerinin doğru çalıştığından emin olacak şekilde testler yazın.

- `pytest` kullanarak Pydantic modeliniz için kapsamlı birim testleri yazın.
- API endpoint'lerinizin doğru hataları döndürdüğünden emin olacak şekilde entegrasyon testleri yapın.

```
import pytest
```

```
from pydantic import ValidationError

def test_user_validation():
    with pytest.raises(ValidationError):
        UserSchema(username='johndoe', email='not-an-email', age=-1)

```

6. Dokümantasyon ve Sürdürülebilirlik

Kodunuzun açıklamalarını güncel tutun ve Pydantic modellerinizin neyi doğruladığını açıklayın. Bu, kodunuzu daha sürdürülebilir kılar.

- Pydantic model alanlarına, doğrulama kriterlerini açıklayan docstringler ekleyin.
- API endpoint'lerinizin davranışını ve beklenen girdi/çıktılarını dokümante edin.

Bu best practices, Django ve Pydantic'i birleştirirken veri doğrulama sürecinizi güçlendirecek ve API'lerinizin daha sağlam ve sürdürülebilir olmasını sağlayacaktır.

Pydantic Custom Hook

Ruff için type hint hook araştırması sonucunda doğrudan bu ihtiyacı karşılayan mevcut bir çözüm bulunmamaktadır. Ruff'un kendi dokümantasyonunda ve GitHub sayfasında da Pydantic tabanlı bir çözüm bulunmamaktadır. Özel bir hook yazarak bu işlevselliği ekleyebiliriz.Ruff'un tanımlı eklentileri doğrudan destek sunmaması sebebiyle doğrudan bir plugin olarak yapılamaz.Ancak Python'un AST(Abstract Syntax Tree) kitaplığını kullanarak,dış bir script oluşturarak ve bu scripti bir 'pre commit'hook olarak entegre edip Pydantic modellerinde type hint doğrulaması yapabiliriz. Bu süreç kodumuzda Pydantic modellerinin düzgün şekilde type hint ile tanımlandığını doğrulamak için kullanılabilir.

Adımlar

1. Proje Dizini ve Sanal Ortam Oluşturma

Yeni bir proje dizini oluşturun ve bir Python sanal ortamı kurun:

```
mkdir pydantic_typehint_project
cd pydantic_typehint_project
python -m venv myenv
source venv/bin/activate  # Windows için: venv\Scripts\activate
```

Bu adımda, projenin bağımlılıklarını izole etmek için bir sanal ortam kuruyoruz.

2. Gerekli Paketleri Kurma

Pydantic ve pre-commit paketlerini yükleyin:

```
pip install pydantic pre-commit
```

Bu adımda, projede kullanacağımız pydantic ve pre-commit paketlerini kuruyoruz.

3. Proje Yapısını Oluşturma

Proje dizininde gerekli dosyaları ve klasörleri oluşturun:

```
mkdir hooks
touch hooks/type_hint_checker.py
touch .pre-commit-config.yaml
touch example.py
```

Bu adımda, proje yapısını oluşturuyoruz. pre-commit-config.yaml dosyasında pre-commit konfigürasyonunu ve example.py dosyasında örnek Pydantic modellerini oluşturacağız.

4. Type Hint Doğrulama Script'i Yazma

Pydantic BaseModel sınıflarındaki type hint eksikliklerini kontrol eden bir AST (Abstract Syntax Tree) tabanlı doğrulama aracıdır.
Aşağıdaki kod bloğunda, Pydantic BaseModel sınıflarında type hint doğrulaması yapar ve eksik type hint'leri raporlar. pre-commit hook'u olarak kullanıldığında, kodu commit etmeden önce bu doğrulamayı otomatik olarak yapar. Eğer bir hata bulunursa, commit işlemi iptal edilir ve hatalar raporlanır.

hooks/type_hint_checker.py

```
import ast
import sys

class TypeHintChecker(ast.NodeVisitor):
    def __init__(self):
        self.errors = []

    def visit_ClassDef(self, node):
        if any(self.is_base_model(base) for base in node.bases):
            for field in node.body:
                if isinstance(field, ast.AnnAssign) and not field.annotation:
                    self.errors.append((field.lineno, f"Field '{field.target.id}' in class '{node.name}'
lacks a type hint"))
        self.generic_visit(node)

    def is_base_model(self, base):
        """BaseModel sınıfı olup olmadığını belirleyen yardımcı fonksiyon"""
        while isinstance(base, ast.Attribute):
            base = base.value
        return isinstance(base, ast.Name) and base.id == 'BaseModel'

    def check_file(self, filename):
        with open(filename, 'r', encoding='utf-8') as file:
            node = ast.parse(file.read(), filename=filename)
            self.visit(node)
        if self.errors:
            for error in self.errors:
                print(f"{filename}:{error[0]}: {error[1]}")
            sys.exit(1)

def run_check(filenames):
    all_errors = []
    for filename in filenames:
        checker = TypeHintChecker()
        checker.check_file(filename)
    if all_errors:
        sys.exit(1)

if __name__ == "__main__":
    run_check(sys.argv[1:])
```

6. pre-commit Konfigürasyon Dosyasını Oluşturma

.pre-commit-config.yaml

Proje dizininde .pre-commit-config.yaml dosyasını oluşturun ve aşağıdaki içeriği ekleyin:

```yaml
repos:
  - repo: local
    hooks:
      - id: pydantic-typehint
        name: Pydantic TypeHint Check
        entry: python hooks/type_hint_checker.py
        language: system
        files: \.py$
```

Bu konfigürasyon dosyası, pre-commit hook'unu tanımlar ve hangi script'in çalıştırılacağını belirtir.

7. pre-commit Ayarlarını Yapılandırma

pre-commit konfigürasyonunu yükleyin ve kurun:

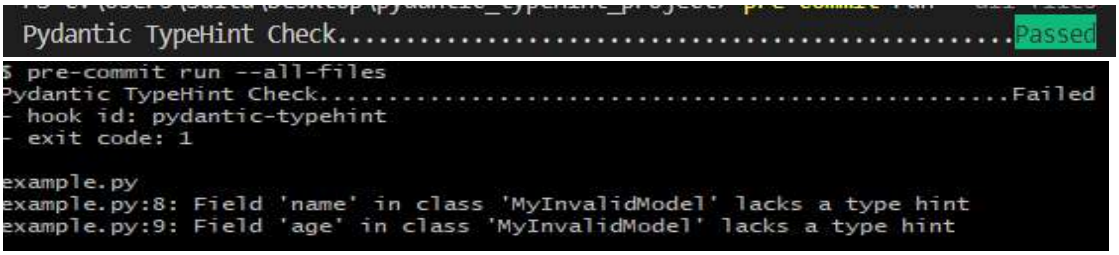
```
pre-commit install
```

8. Pre-commit Hook'unu Test Etme

pre-commit hook'unun doğru çalışıp çalışmadığını test etmek için tüm dosyaları kontrol edin:

```
pre-commit run --all-files
```

Bu komut, tüm Python dosyalarını kontrol eder ve type hint eksikliklerini rapor eder.



Kaynaklar

https://github.com/pydantic/pydantic

https://www.obytes.com/blog/integrate-pydantic-with-django-and-django-rest-framework

https://www.kevsrobots.com/learn/pydantic/11_performance_optimization_tips.html

https://stackoverflow.com/questions/77661139/how-do-i-stop-the-ruff-linter-from-moving-imports-into-a-type-checking-if-block

https://docs.pydantic.dev/latest/#why-use-pydantic