Uygulamalı Laboratuvar: Test Fixture'ları Kullanarak Başlangıç Durumu Oluşturma

Gerekli tahmini süre: 30 dakika

Test Fixture'ları Kullanarak Başlangıç Durumu Oluşturma laboratuvarına hoş geldiniz. Test fixture'ları, testleri çalıştırmadan önce ve sonra bilinen bir başlangıç durumunu oluşturmak için kullanılır. Test fixture'ları ile bir testin veya test gruplarının çalıştırılmasından önce test ortamının nasıl göründüğünü ve test çalıştıktan sonra tekrar tanımlayabilirsiniz.

Öğrenme Hedefleri

Bu laboratuvarı tamamladıktan sonra şunları yapabileceksiniz:

- Başlangıç durumunu ayarlamak ve kaldırmak için test fixture'ları oluşturmak
- Kodunuzu test etmek için harici dosyalardan test verisi yüklemek
- Yüklenen test verilerini test durumlarınızda ve doğrulamalarınızda kullanmak

Theia Hakkında

Theia, masaüstünde veya bulutta çalıştırılabilen açık kaynaklı bir IDE (Entegre Geliştirme Ortamı)dir. Bu laboratuvarı yapmak için Theia IDE'sini kullanacaksınız. Theia ortamına giriş yaptığınızda, yalnızca sizin için ayrılmış bir 'bulut üzerindeki bilgisayar' ile karşılaşırsınız. Bu, laboratuvarlar üzerinde çalıştığınız sürece sizin için mevcuttur. Çıkış yaptığınızda, bu 'bulut üzerindeki bilgisayar' ve oluşturduğunuz dosyalar silinir. Bu nedenle, laboratuvarlarınızı tek bir oturumda tamamlamak iyi bir fikirdir. Laboratuvarın bir kısmını tamamlayıp daha sonra Theia laboratuvarına dönerken, başlangıçtan başlamak zorunda kalabilirsiniz. Tüm Theia laboratuvarlarınızı tamamlamak için zamanınız olduğunda çalışmayı planlayın.

Laboratuvar Ortamını Kurma

Laboratuvara başlamadan önce biraz hazırlık yapmamız gerekiyor.

Terminali Aç

Editördeki menüyü kullanarak bir terminal penceresi açın: Terminal > Yeni Terminal.

Terminalde, eğer zaten `/home/projects` klasöründe değilseniz, şimdi proje klasörünüze geçin.

```
cd /home/project
```

Kodu Repo'sunu Klonlayın

Şimdi test etmemiz gereken kodu alalım. Bunu yapmak için, git deposunu klonlamak için `git clone` komutunu kullanacaksınız:

```
git clone https://github.com/ibm-developer-skills-network/duwjx-tdd_bdd_PracticeCode.git
```

Laboratuvar Klasörüne Geçin

Depoyu klonladıktan sonra, laboratuvar dizinine geçin,

```
cd duwjx-tdd_bdd_PracticeCode/labs/03_test_fixtures
```

Python Bağımlılıklarını Yükleyin

Son hazırlık adımı, laboratuvar için gerekli Python paketlerini yüklemek üzere `pip` kullanmaktır:

```
python3.8 -m pip install -r requirements.txt
```

Laboratuvara başlamak için hazırsınız.

Opsiyonel

Eğer komut istemi çok uzun olursa, bunu daha yönetilebilir bir hale getirmek için şöyle kısaltabilirsiniz:

```
export PS1="[\[\033[01;32m\]\u\[\033[00m\]: \[\033[01;34m\]\W\[\033[00m\]]\$ "
```

Test Donanımlarının Gözden Geçirilmesi

Bu laboratuvar çalışmasında, PyUnit paketinde bulunan çeşitli test donanımlarını kullanacaksınız. Aşağıdaki kodu gözden geçirmeniz gerekiyor; bu kod, hangi test donanımlarının mevcut olduğunu ve ne zaman çağrılacağını gösteriyor.

Laboratuvar sırasında, kod eklemek için uygun yeri seçeceksiniz. Örneğin, bir sınıftaki tüm testlerden önce bir kez çalışacak kod eklemeniz gerekebilir. `setUpClass()` metodunun, test durumundaki tüm testlerden önce bir kez çalışacağını bilmelisiniz.

```
def setUpModule():             # runs once before any tests
def tearDownModule():          # runs once after all tests
class MyTestCases(TestCase):   # the start of a test case
    @classmethod
    def setUpClass(cls):       # runs once before test case
    @classmethod
    def tearDownClass(cls):    # runs once after test case
    def setUp(self):           # runs before each test
    def tearDown(self):        # runs after each test
```

Kod Editörünü Aç

Bu laboratuvar, test fixture'larının test öncesi ve sonrası başlangıç durumunu ayarlamak ve temizlemek için farklı şekillerde nasıl kullanılabileceğini gösterecektir.

IDE'de, `duwjx-tdd_bdd_PracticeCode/labs/03_test_fixtures` klasörüne gidin. Bu, bu laboratuvar için kullanacağınız tüm kaynak kodunun bulunduğu yerdir.

```
duwjx-tdd_bdd_PracticeCode/labs/03_test_fixtures
```

You will do all of your editing work in the file `tests/test_account.py`. Open that up in the editor to get started.

Open **test_account.py** in IDE

Adım 1: Veritabanını Başlat

Bu adımda, veritabanına bağlanmak ve bağlantıyı kesmek için bir test düzeni kuracaksınız. Bunu yalnızca tüm testlerden önce ve sonra bir kez yapmanız yeterlidir.

Göreviniz

Tüm testlerden önce veritabanına bağlanmak ve tüm testlerden sonra veritabanından bağlantıyı kesmek için hangi metin düzenlerinin en iyi kullanılacağını düşünün.

Aşağıdaki **SQLAlchemy** komutları size bu konuda yardımcı olacaktır:

Komut	**Tanım**
db.create_all()	SQLAlchemy tablolarını oluşturur
db.session.close()	Veritabanı bağlantısını kapatır

Tüm testlerden önce `db.create_all()` çağırmak ve tüm testlerden sonra `db.session.close()` çağırmak için sınıf düzeyindeki düzenleri kullanın.

Çözüm

▼ Çözüm için buraya tıklayın.

Bu kodu `test_account.py` dosyasına yapıştırın. Doğru girintilemeye dikkat edin.

```
        @classmethod
        def setUpClass(cls):
            """ Connect and load data needed by tests """
            db.create_all()  # make our SQLAlchemy tables
        @classmethod
        def tearDownClass(cls):
            """Disconnect from database"""
            db.session.close() # close the database session
```

Testleri Çalıştır

Hata olmadan test durumunuzun çalıştığından emin olmak için `nosetests` komutunu çalıştırın.

```
nosetests
```

Adım 2: Test Verilerini Yükle

Bu adımda, test sırasında kullanılmak üzere bazı test verilerini yükleyeceksiniz. Bu, tüm testlerden önce sadece bir kez yapılması gereken bir işlemdir, bu nedenle bunu bir sınıf yönteminde gerçekleştireceksiniz.

Göreviniz

`tests/fixtures` klasöründe `account_data.json` adında bir dosyada test verileri bulunmaktadır.

`tests/fixtures/account_data.json` dosyasından verileri, önceden tanımlanmış olan `ACCOUNT_DATA` adındaki küresel bir değişkene yükleyin.

Verileri yüklemek için Python kodu:

```
with open('tests/fixtures/account_data.json') as json_data:
    ACCOUNT_DATA = json.load(json_data)
```

Çözüm

▼ Çözüm için buraya tıklayın.

Bu kodu `test_account.py` dosyasına yapıştırın. Doğru şekilde girintilediğinizden emin olun.

```
    @classmethod
    def setUpClass(cls):
        """ Connect and Load data needed by tests """
        db.create_all()  # make our SQLAlchemy tables
        global ACCOUNT_DATA
        with open('tests/fixtures/account_data.json') as json_data:
            ACCOUNT_DATA = json.load(json_data)
```

Testleri Çalıştır

Test durumunuzun hatasız bir şekilde çalıştığından emin olmak için `nosetests` komutunu çalıştırın.

```
nosetests
```

Adım 3: Bir Hesap Oluşturmak için Bir Test Durumu Yazın

Artık ilk testinizi yazmaya hazırsınız. Beş hesap için test verileri içeren `ACCOUNT_DATA` sözlüğünü kullanarak tek bir hesap oluşturacaksınız.

Göreviniz

`Account` sınıfının veritabanına bir hesap eklemek için kullanılabilecek bir `create()` metodu vardır. Ayrıca, tüm hesapları döndüren bir sorgu gerçekleştiren `all()` metodu da bulunmaktadır.

Test durumlarınız bir hesap oluşturmalı ve ardından `Account.all()` metodunu çağırarak bir hesabın döndüğünü doğrulamalıdır.

Çözüm

▼ Çözüm için buraya tıklayın.

Bu kodu `test_account.py` dosyasına yapıştırın. Doğru bir şekilde girintilediğinizden emin olun.

```
    def test_create_an_account(self):
        """ Test create a single Account """
        data = ACCOUNT_DATA[0] # get the first account
```

```
        account = Account(**data)
        account.create()
        self.assertEqual(len(Account.all()), 1)
```

Testleri Çalıştır

Test durumunuzun geçtiğinden emin olmak için `nosetests` komutunu çalıştırın.

```
nosetests
```

Görmelisiniz:

Adım 4: Tüm Hesapları Oluşturmak için Bir Test Senaryosu Yazın

Artık bir hesabın başarıyla oluşturulabileceğini bildiğinize göre, `ACCOUNT_DATA` sözlüğündeki beş hesabı oluşturan bir test senaryosu yazalım.

Göreviniz

`ACCOUNT_DATA` sözlüğündeki tüm verileri yüklemek için bir `for` döngüsü kullanın ve ardından `Account.all()` yöntemini kullanarak bunları alın ve dönen hesap sayısının test verileri sözlüğündeki hesap sayısına eşit olduğunu doğrulayın.

Çözüm

▼ Çözüm için buraya tıklayın.

Bu kodu `test_account.py` dosyasına yapıştırın. Doğru şekilde girinti yaptığınızdan emin olun.

```
    def test_create_all_accounts(self):
        """ Test creating multiple Accounts """
        for data in ACCOUNT_DATA:
            account = Account(**data)
            account.create()
        self.assertEqual(len(Account.all()), len(ACCOUNT_DATA))
```

Testleri Çalıştır

Test durumunuzun geçtiğinden emin olmak için `nosetests` komutunu çalıştırın.

```
nosetests
```

HATA: Bu sefer testler geçmedi! **AssertionError: 6 != 5** ve **AssertionError: 7 != 1** hakkında iki hata almış olmalısınız:

```
======================================================================
FAIL: Test Account creation using known data
----------------------------------------------------------------------
Traceback (most recent call last):
  File "/Users/rofrano/Code/duwjx-tdd_bdd_PracticeCode/labs/03_test_fixtures/tests/test_account.py", line 52, in test_create_an_account
    self.assertEqual(len(Account.all()), 1)
AssertionError: 7 != 1
```

Neden sadece bir tane beklerken yedi hesap döndürüldü?

Şimdi bunu bir sonraki adımda nasıl düzeltebileceğimize bakalım.

Adım 5: Her testten önce tabloları temizle

Test durumunuzun başarısız olmasının nedeni, önceki bir testten gelen verilerin bir sonraki testin sonucunu etkilemesidir. Bunu önlemek için, her testten önce ve sonra çalışacak daha fazla test verisi eklemeniz gerekir.

Göreviniz

Bir tablodan verileri kaldırmanın bir yolu, `db.session.query(<Table>).delete()` komutunu kullanmaktır; burada `<Table>`, tabloların sınıf adıdır. Bu, tablodaki tüm kayıtları silecektir. Bu değişikliği onaylamak için `db.session.commit()` komutunu da kullanmalısınız. Bunu her testten önce çalışan veriye ekleyin.

`Account` sınıfınız için doğru sözdizimi:

```
db.session.query(Account).delete()
```

Aynı zamanda her testten sonra `db.session.remove()` komutunu kullanmak da iyi bir fikirdir. Bunu her testten sonra çalışan fixture'a ekleyin.

Çözüm

▼ Çözüm için buraya tıklayın.

Bu kodu `test_account.py` dosyasına yapıştırın. Doğru bir şekilde girintilediğinizden emin olun.

```
        def setUp(self):
            """Truncate the tables"""
            db.session.query(Account).delete()
            db.session.commit()
        def tearDown(self):
            """Remove the session"""
            db.session.remove()
```

Testleri Çalıştır

Test durumunuzun geçtiğinden emin olmak için `nosetests` komutunu çalıştırın.

```
nosetests
```

Aşağıdaki raporu görmelisiniz:

Tebrikler! Bu sefer tüm test durumlarınız başarılı oldu.

Sonuç

Test Fixtures Laboratuvarını Tamamladığınız İçin Tebrikler

Umarım test fixtures'ları testlerinizde nasıl kullanacağınız hakkında iyi bir anlayışa sahip oldunuz. Test fixtures kullanmanın, her testten önce ve sonra sistemin durumunu kontrol etmenizi sağladığını gördünüz; böylece testler izole bir şekilde çalışır ve her seferinde tekrarlanabilir sonuçlar alırsınız. Şimdi bazı kişisel projelerinizdeki kodlar için test senaryoları yazmayı deneyin.

Author(s)

John J. Rofrano