# Uygulamalı Laboratuvar: Test Odaklı Geliştirme Uygulaması

**Gerekli tahmini süre:** 30 dakika

**Test Odaklı Geliştirme** laboratuvarına hoş geldiniz. Test odaklı geliştirme (TDD), önce sahip olmak istediğiniz kod için test senaryolarını yazdığınız ve ardından bu test senaryolarını geçirecek kodu yazdığınız bir yazılım geliştirme yaklaşımıdır.

Bu laboratuvar boyunca, size verilen gereksinimlere dayanarak test senaryoları yazacak ve ardından bu test senaryolarını geçirecek kodu yazacaksınız. Bunun ne kadar kolay olduğunu göreceksiniz.

## Öğrenme Hedefleri

Bu laboratuvarı tamamladıktan sonra şunları yapabileceksiniz:

- TDD iş akışını takip etmek
- Uygulama gereksinimlerine dayalı test senaryoları yazmak
- Test senaryolarını geçirecek kodu yazmak

# Theia Hakkında

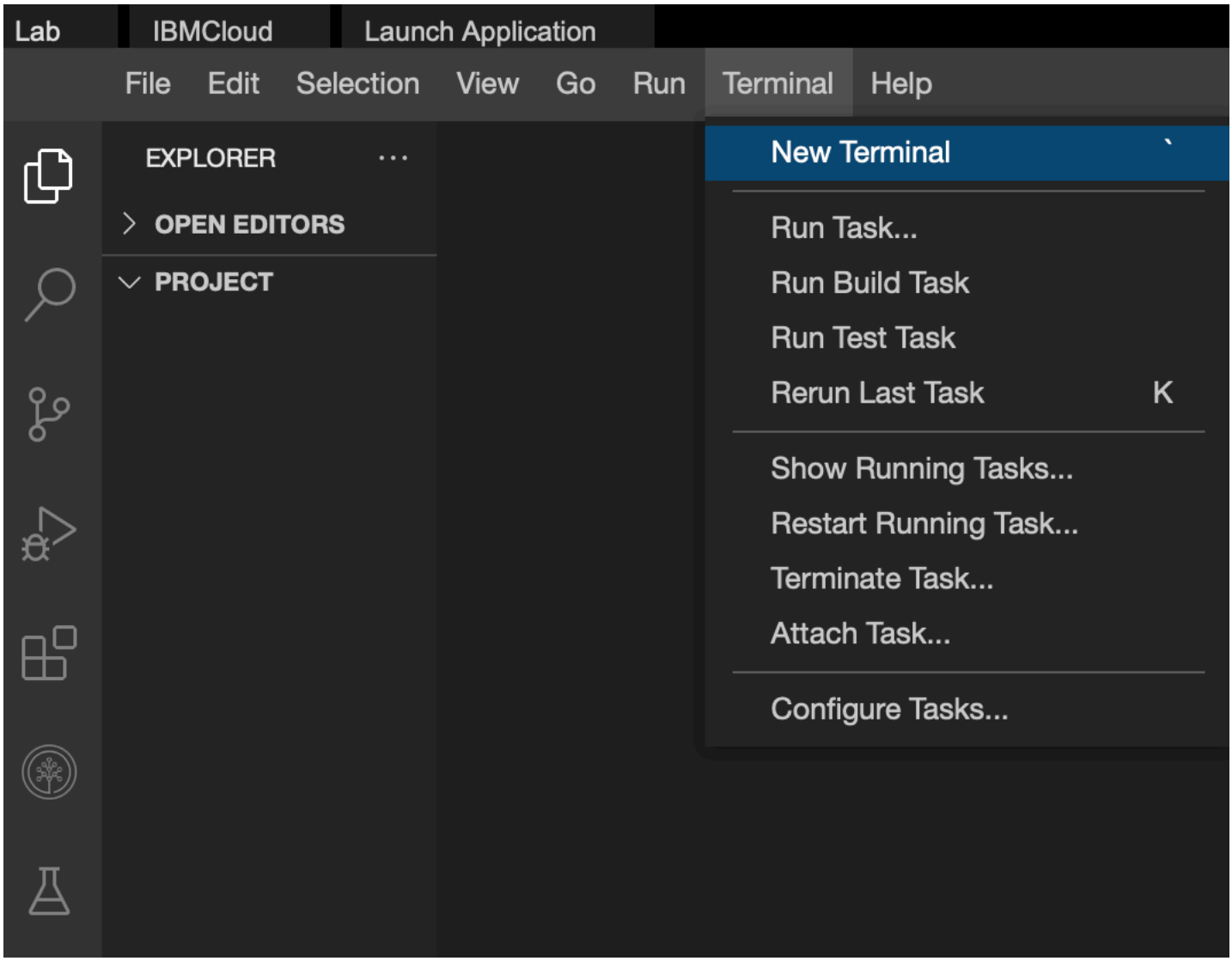
Theia, masaüstünde veya bulutta çalıştırılabilen açık kaynaklı bir IDE (Entegre Geliştirme Ortamı)dir. Bu laboratuvarı yapmak için Theia IDE'sini kullanacaksınız. Theia ortamına giriş yaptığınızda, sadece sizin için ayrılmış 'buluttaki özel bir bilgisayar' ile karşılaşırsınız. Laboratuvarlarla çalıştığınız sürece bu sizin için mevcuttur. Çıkış yaptığınızda, bu 'buluttaki özel 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 yapmanız gerekiyor.

## Terminali Açın

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 Klonla

Şimdi test etmeniz gereken kodu alın. Bunu yapmak için, git deposunu klonlamak üzere `git clone` komutunu kullanın:

```
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/07_practicing_tdd
```

## Python Bağımlılıklarını Kurun

Son hazırlık adımı, laboratuvar için gereken Python paketlerini kurmak üzere `pip` kullanmaktır:

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

Laboratuvara başlamak için hazırsınız.

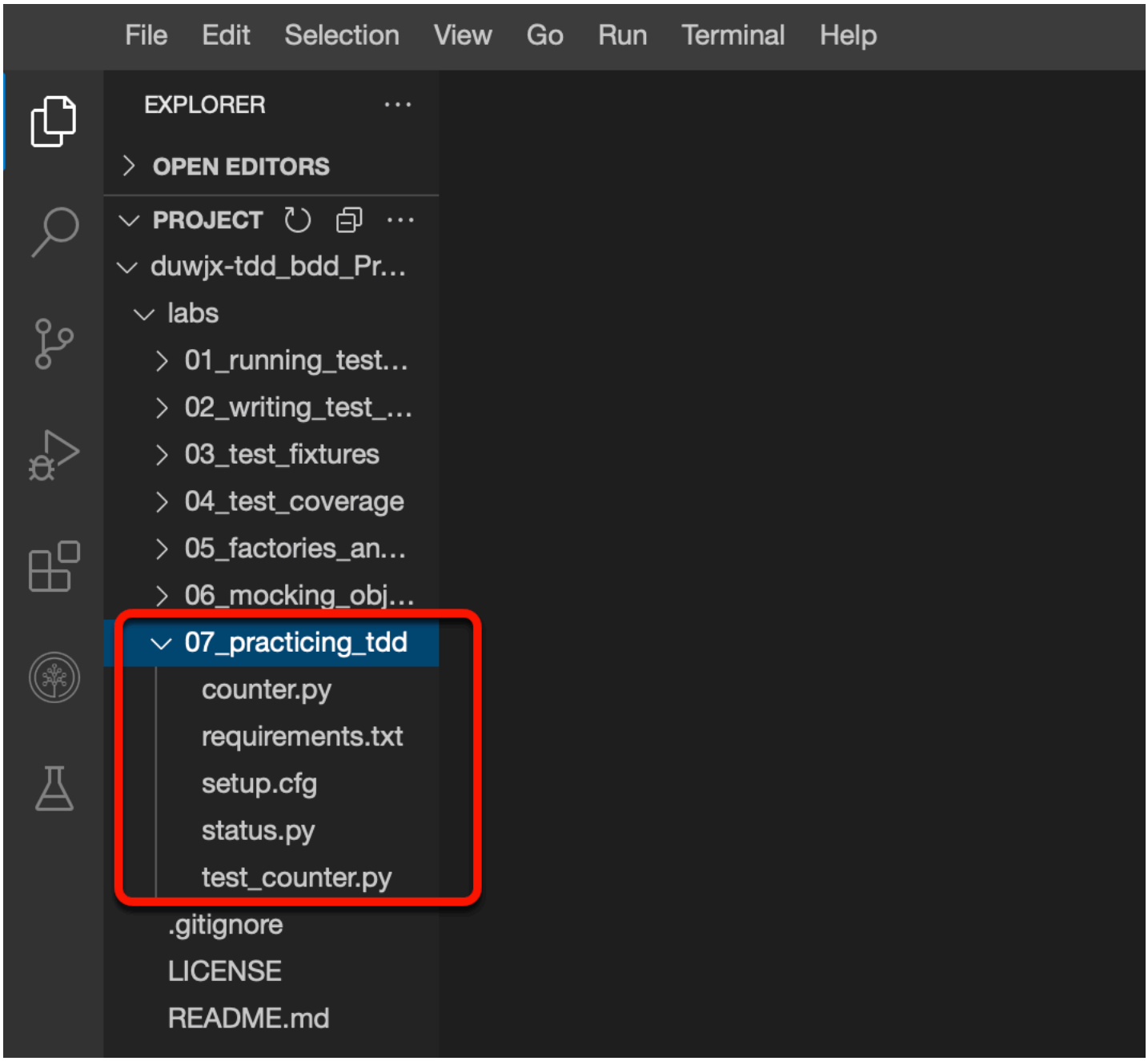
### Opsiyonel

Eğer terminalde çalışmak zorlaşırsa çünkü komut istemi çok uzunsa, aşağıdaki komutu kullanarak istemi kısaltabilirsiniz:

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

# Koda Git

Ekranınızın sol tarafındaki IDE'de, `duwjx-tdd_bdd_PracticeCode/labs/07_practicing_tdd` klasörüne gidin. Bu klasör, bu laboratuvar için kullanacağınız tüm kaynak kodunu içerir.




Ekleyeceğiniz test senaryoları `test_counter.py` dosyasında, ekleyeceğiniz kod ise `counter.py` dosyasında bulunmaktadır. Çalışacağınız tek iki dosya bunlardır.

# Gereksinimler

Birden fazla sayacı takip edebilen bir web servisi oluşturmanız istendiğini varsayın. Web servisinin aşağıdaki gereksinimleri vardır:

- API RESTful olmalıdır.
- Uç nokta `/counters` olarak adlandırılmalıdır.
- Bir sayaç oluştururken, ismin yolu belirtmeniz gerekmektedir.
- Tekrar eden isimler bir çakışma hata kodu döndürmelidir.
- Servis, bir sayacı isme göre güncelleyebilmelidir.
- Servis, bir sayacın mevcut değerini alabilmelidir.
- Servis, bir sayacı silebilmelidir.

Son üç gereksinim henüz uygulanmamıştır. Bu gereksinimleri TDD prensiplerini kullanarak, önce test durumlarını yazarak ve ardından test durumlarını geçirecek kodu yazarak uygulamanız istendi.

# REST API Kılavuzları İncelemesi

Bu laboratuvar için test vakalarını yazmanıza olanak tanıyan REST API'leri oluşturmak için kılavuzlar bulunmaktadır:

| Eylem | Yöntem | Dönüş kodu | URL |
|---|---|---|---|
| Oluştur | POST | 201_CREATED | POST /counters/{name} |
| Oku | GET | 200_OK | GET /counters/{name} |
| Güncelle | PUT | 200_OK | PUT /counters/{name} |

| Eylem | Yöntem | Dönüş kodu | URL |
|---|---|---|---|
| Sil | DELETE | 204_NO_CONTENT | DELETE /counters/{name} |

Bu kılavuzları takip ederek, web hizmetini nasıl çağıracağınız ve ne döndürmesi gerektiği hakkında varsayımlarda bulunabilirsiniz.

# Adım 1: Sayacı Güncellemek İçin Bir Test Yaz

Bir sayacı güncellemek için bir test durumu uygulamaya başlayacaksınız. REST API yönergelerine göre, bir güncelleme `PUT` isteği kullanır ve başarılıysa `200_OK` kodu döner. Önce bir sayaç oluşturun, ardından onu güncelleyin.

## Göreviniz

`test_counter.py` dosyasında `test_update_a_counter(self)` adında bir test oluşturun. Aşağıdaki adımları uygulamalıdır:

1. Bir sayaç oluşturma çağrısı yapın.
2. Başarılı bir dönüş kodu döndüğünden emin olun.
3. Sayaç değerini bir temel olarak kontrol edin.
4. Az önce oluşturduğunuz sayacı güncelleme çağrısı yapın.
5. Başarılı bir dönüş kodu döndüğünden emin olun.
6. Sayaç değerinin, 3. adımda ölçtüğünüz temel değerden bir fazla olduğunu kontrol edin.

   İpucu: Her test için farklı bir sayaç adı kullanın, böylece bir test diğerlerini etkilemez.

Open **test_counter.py** in IDE

## Çözüm

▼ Çözüm için buraya tıklayın.

Bu, bir sayacı güncelleme işlevselliğini test etmek için bir çözümdür.

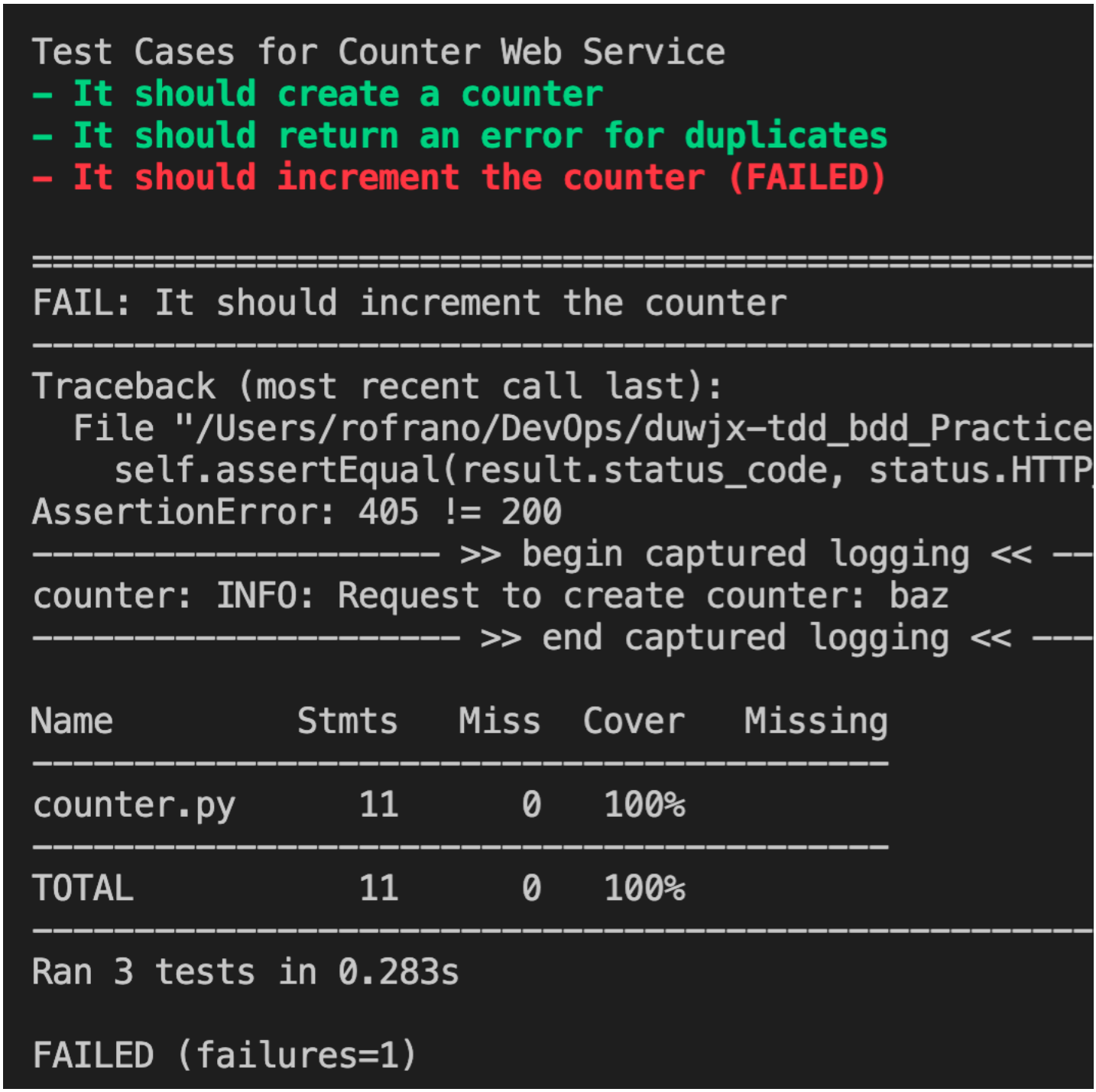
```
        def test_update_a_counter(self):
            """It should increment the counter"""
            result = self.client.post("/counters/baz")
            self.assertEqual(result.status_code, status.HTTP_201_CREATED)
            data = result.get_json()
            baseline = data["baz"]
            # Update the counter
            result = self.client.put("/counters/baz")
            self.assertEqual(result.status_code, status.HTTP_200_OK)
            data = result.get_json()
            self.assertEqual(data["baz"], baseline + 1)
```

## Testleri Çalıştır

Yeni testin başarısız olduğunu görmek için `nosetests` komutunu çalıştırın:

```
nosetests
```

Terminalde tüm sonuçları görmek için yukarı kaydırmanız gerekebilir. Sonuçlar şu şekilde görünmelidir:




Hata mesajının **AssertionError: 405 != 200** olduğunu unutmayın. 405, `405_METHOD_NOT_ALLOWED` anlamına gelir ve Flask'ın `/counters/<name>` adında bir rota bulduğunu, ancak `PUT` metoduna izin vermediğini gösterir. Bunun nedeni, o kodu henüz yazmamış olmanızdır. Bu böyle.

# Adım 2: Sayacı Güncelleme Uygulama

Şimdi testi geçmek için kodu yazın. Eğer **Flask** konusunda deneyiminiz varsa, sayaç hizmetinin tüm yollarının aynı olduğunu, sadece **yöntemin** değiştiğini unutmayın.

Başlamak için, sayacı güncelleyen bir fonksiyon uygulayacaksınız. REST API yönergelerine göre, bir güncelleme `PUT` isteği kullanır ve başarılıysa `200_OK` kodu döner. Belirtilen adı karşılayan sayacı güncelleyen bir fonksiyon oluşturun.

## Göreviniz

`counter.py` dosyasında `update_counter(name)` adında bir fonksiyon oluşturun. Aşağıdaki adımları uygulamalıdır:

1. `/counters/<name>` uç noktasında PUT yöntemi için bir rota oluşturun.
2. O rotayı uygulamak için bir fonksiyon oluşturun.
3. Sayacı 1 artırın.
4. Yeni sayacı ve `200_OK` dönüş kodunu döndürün.

Open **counter.py** in IDE

## Çözüm

▼ Çözüm için buraya tıklayın.

Bu, bir sayacı güncelleme işlevselliğini uygulamak için bir çözümdür.

```
@app.route("/counters/<name>", methods=["PUT"])
def update_counter(name):
    """Update a counter"""
    app.logger.info(f"Request to update counter: {name}")
    global COUNTERS
    COUNTERS[name] += 1
    app.logger.info(f"Counter: {name} is now {COUNTERS[name]}")
    return { name: COUNTERS[name] }, status.HTTP_200_OK
```

Bu kodun herhangi bir hata işleme sağlamadığını unutmayın. Ya sayaç yoksa? Sayaç mevcut olduğundan emin olmak için yeni bir test durumu ve ardından onu artırmadan önce kontrol etmek için daha fazla koda ihtiyacınız olacak.

## Testleri Çalıştır

`nosetests` komutunu çalıştırın ve test durumlarının geçtiğinden emin olun:

```
nosetests
```

Sonuçlar şöyle görünmelidir:

```
Test Cases for Counter Web Service
- It should create a counter
- It should return an error for duplicates
- It should increment the counter

Name         Stmts   Miss  Cover   Missing
------------------------------------------
counter.py      18      1    94%   28
------------------------------------------
TOTAL           18      1    94%
----------------------------------------------------------------------
Ran 3 tests in 0.170s

OK
```

# Adım 3: Bir sayaç okumak için test yazın

Sonraki adımda, bir sayacı okumak için bir test durumu yazacaksınız. REST API yönergelerine göre, bir okuma `GET` isteği kullanır ve başarılıysa `200_OK` kodu döner. Bir sayaç oluşturun ve ardından onu okuyun.

## Göreviniz

`test_counter.py` dosyasında `test_read_a_counter(self)` adında bir test oluşturun. Aşağıdaki adımları uygulamalıdır:

1. Bir sayaç oluşturmak için bir çağrı yapın.
2. Başarılı bir dönüş kodu döndüğünden emin olun.
3. Az önce oluşturduğunuz sayacı okumak için bir çağrı yapın.
4. Başarılı bir dönüş kodu döndüğünden emin olun.
5. Dönen sayaç değerinin 0 olduğunu kontrol edin.

> İpucu: Her test için farklı bir sayaç adı kullanın, böylece bir test diğerlerini etkilemez.

# Çözüm

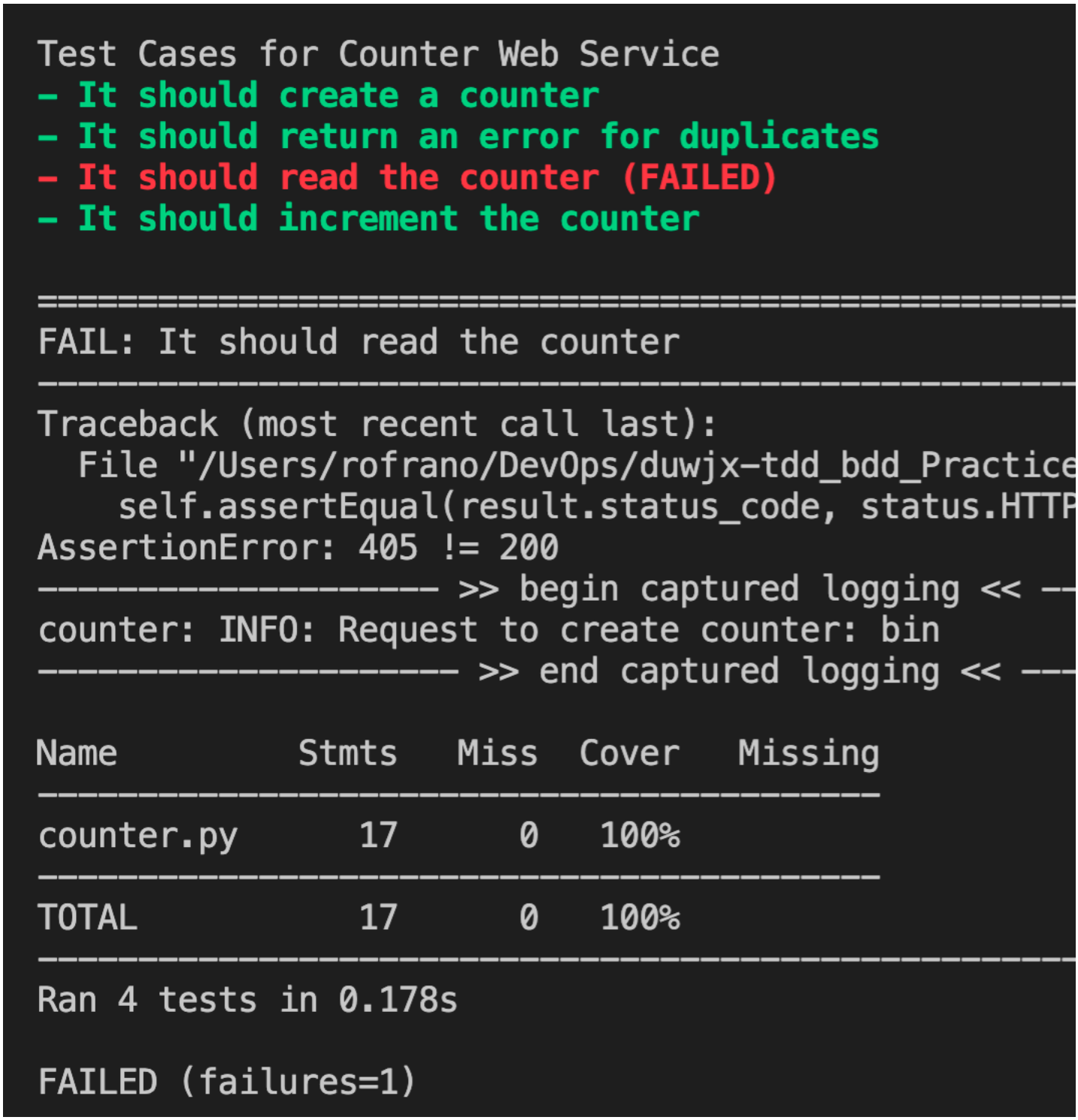
▶ Çözüm için buraya tıklayın.

# Testleri Çalıştır

Yeni testin başarısız olduğunu görmek için `nosetests` komutunu çalıştırın:

```
nosetests
```

Sonuçlar şu şekilde görünmelidir:




Tekrar dikkat edin, hata **AssertionError: 405 != 200**. Bir 405 `405_METHOD_NOT_ALLOWED`'dır, bu da Flask'ın `/counters/<name>` adında bir rota bulduğunu ancak `GET` yöntemine izin vermediğini gösterir. Bunun nedeni henüz o kodu yazmamış olmanızdır. O幾.

# Adım 4: Sayacı okuma işlemini gerçekleştirin

Bir kez daha, bir testin geçmesi için kod yazma zamanı geldi. Sayacı okuma işlemi için kodu uygulayacaksınız. REST API yönergelerine göre, bir okuma `GET` isteği kullanır ve başarılıysa `200_OK` kodu döner. Belirtilen adı karşılayan sayacı döndüren bir fonksiyon oluşturun.

## Göreviniz

`counter.py` dosyasında `read_counter(name)` adında bir fonksiyon oluşturun. Aşağıdaki adımları uygulamalıdır:

1. `/counters/<name>` uç noktasında `GET` yöntemi için bir yol oluşturun.
2. O yolu uygulamak için bir fonksiyon oluşturun.
3. Sayacın mevcut değerini alın.
4. Sayacı ve `200_OK` dönüş kodunu döndürün.

## Çözüm

▼ Çözüm için buraya tıklayın.

Bu, bir sayacı okuma işlevselliğini uygulamak için bir çözümdür.

```
@app.route("/counters/<name>", methods=["GET"])
def read_counter(name):
    """Read a counter"""
    app.logger.info(f"Request to read counter: {name}")
    counter = COUNTERS[name]
    app.logger.info(f"Counter: {name} is {counter}")
    return { name: counter }, status.HTTP_200_OK
```

> Kodun herhangi bir hata işleme sağlamadığını unutmayın. Tekrar, eğer sayaç mevcut değilse, yeni bir test durumu oluşturmanız ve ardından sayaç mevcut olup olmadığını kontrol etmek için daha fazla kod yazmanız gerekecek.

### Testleri Çalıştır

`nosetests` komutunu çalıştırın ve test durumlarının geçtiğinden emin olun:

```
nosetests
```

Sonuçlar şöyle görünmelidir:

```
Test Cases for Counter Web Service
- It should create a counter
- It should return an error for duplicates
- It should read the counter
- It should increment the counter

Name         Stmts   Miss  Cover   Missing
------------------------------------------
counter.py      23      0   100%
------------------------------------------
TOTAL           23      0   100%
----------------------------------------------------------------------
Ran 4 tests in 0.203s

OK
```

## Adım 5: Bir sayacı silmek için test yazın

Şimdi bir sayacı silmek için bir test durumu yazacaksınız. REST API yönergelerine göre, bir okuma `DELETE` isteği kullanır ve başarılıysa `204_NO_CONTENT` kodunu döner. Belirtilen adı eşleşen sayacı silen bir fonksiyon oluşturun.

### Göreviniz

`test_counter.py` dosyasında `test_delete_a_counter(self)` adında bir fonksiyon oluşturun. Aşağıdaki adımları uygulamalıdır:

1. Bir sayaç oluşturma çağrısı yapın.
2. Başarılı bir dönüş kodu döndüğünden emin olun.
3. Yeni oluşturduğunuz sayacı silme çağrısı yapın.
4. Başarılı bir dönüş kodu döndüğünden emin olun.

> İpucu: Her test için farklı bir sayaç adı kullanın, böylece bir test diğerlerini etkilemez.

# Çözüm

▼ Çözüm için buraya tıklayın.

Bu, bir sayacı silme işlevselliğini test etmek için bir çözümdür.

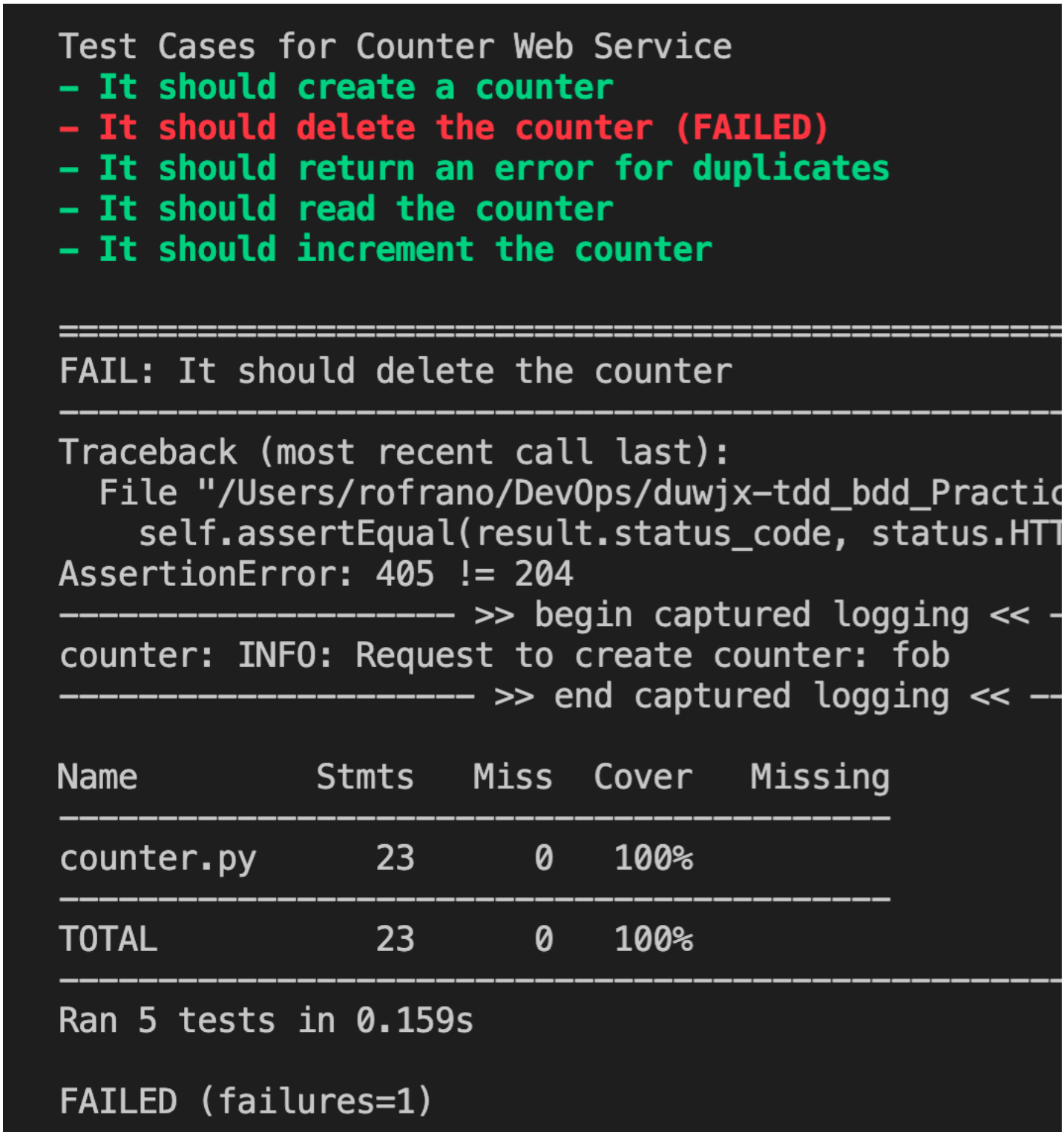
```
        def test_delete_a_counter(self):
            """It should delete the counter"""
            result = self.client.post("/counters/fob")
            self.assertEqual(result.status_code, status.HTTP_201_CREATED)
            # Delete the counter
            result = self.client.delete("/counters/fob")
            self.assertEqual(result.status_code, status.HTTP_204_NO_CONTENT)
```

# Testleri Çalıştır

Yeni testin başarısız olduğunu görmek için `nosetests` komutunu çalıştırın:

```
nosetests
```

Sonuçlar şöyle görünmelidir:




Tekrar dikkat edin, hata **AssertionError: 405 != 204**. Bir 405, `405_METHOD_NOT_ALLOWED` olup, Flask'ın `/counters/<name>` adlı bir rota bulduğunu ancak `DELETE` yöntemine izin vermediğini gösterir. Bunun nedeni henüz o kodu yazmamış olmanızdır. Bu.

# Adım 6: Bir sayacı silme işlemini gerçekleştir

Bu son adımda, bir testin geçmesi için tekrar kod yazacaksınız. Bu sefer, bir sayacı silmek için kodu uygulayacaksınız. REST API yönergelerine göre, bir silme işlemi `DELETE` isteği kullanır ve başarılıysa `204_NO_CONTENT` kodunu döner.

## Göreviniz

`counter.py` dosyasında `delete_counter(name)` adında bir fonksiyon oluşturun. Aşağıdaki adımları uygulamalıdır:

1. `/counters/<name>` uç noktasında `DELETE` yöntemi için bir rota oluşturun.
2. O rotayı uygulamak için bir fonksiyon oluşturun.
3. İsimle eşleşen sayacı silin.
4. Sayacı ve `204_NO_CONTENT` dönüş kodunu döndürün.

## Çözüm

▼ Çözüm için buraya tıklayın.

Bu, bir sayacı silme işlevselliğini uygulamak için bir çözümdür.

```
@app.route("/counters/<name>", methods=["DELETE"])
def delete_counter(name):
    """Delete a counter"""
    app.logger.info(f"Request to delete counter: {name}")
    del(COUNTERS[name])
    app.logger.info(f"Counter: {name} has been deleted")
    return '', status.HTTP_204_NO_CONTENT
```

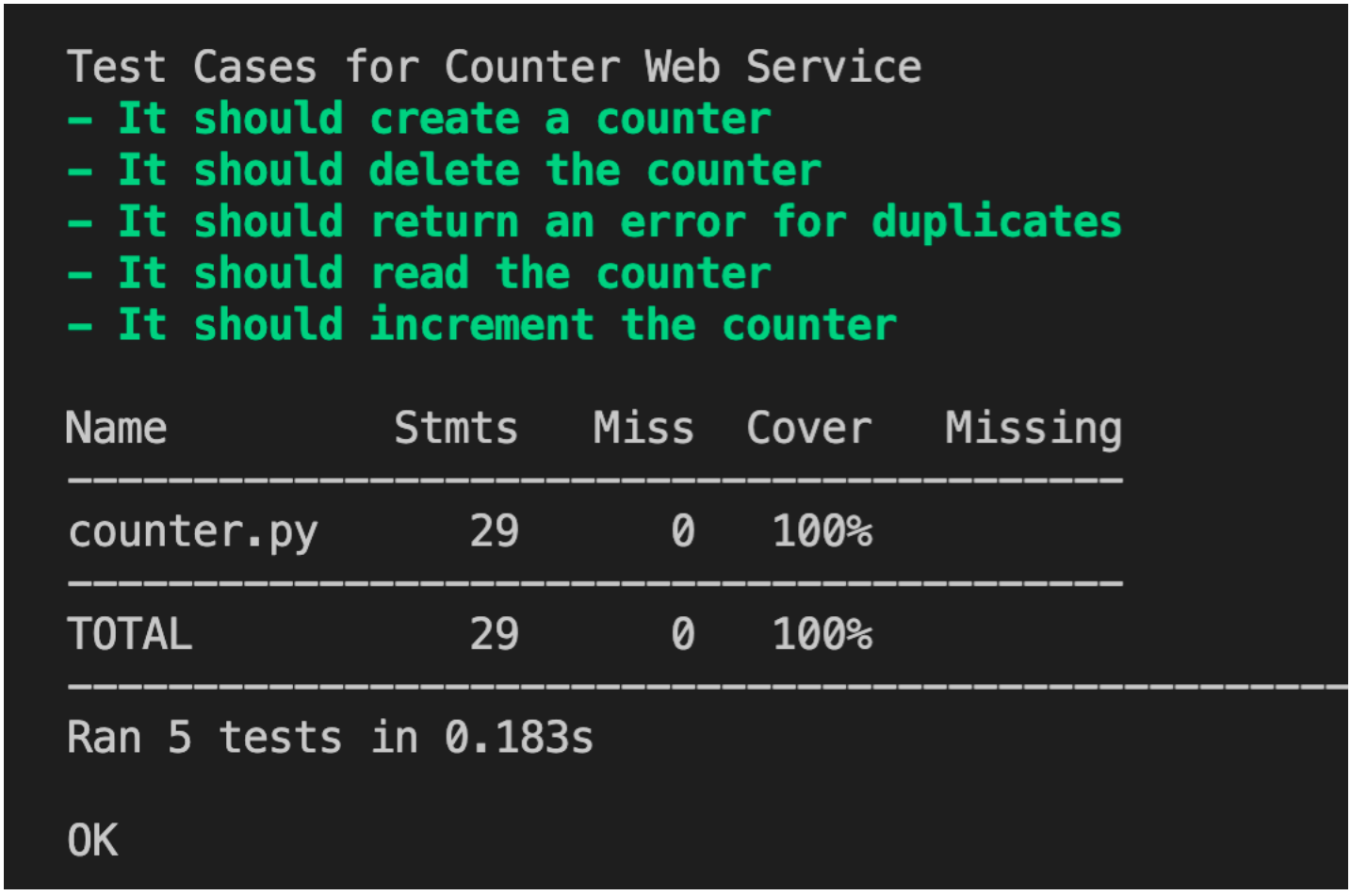
Hatanın ele alınmadığını unutmayın. Yine, eğer sayaç mevcut değilse, yeni bir test durumu oluşturmanız ve ardından sayacın silinmeden önce mevcut olup olmadığını kontrol etmek için daha fazla kod yazmanız gerekecektir.

## Testleri Çalıştır

`nosetests` komutunu çalıştırın ve test durumlarının geçtiğinden emin olun:

```
nosetests
```

Sonuçlar şöyle görünmelidir:




# Sonuç

Tebrikler! **Test Tabanlı Geliştirme** laboratuvarını tamamladınız.

Umarım, henüz mevcut olmayan kodlar için test senaryoları yazmanın ne kadar kolay olduğunu şimdi anlamışsınızdır. Test ettiğiniz davranışı tanımlamak için uygulama gereksinimlerini kullanıyorsunuz ve ardından bu şekilde davranan kodu yazıyorsunuz.

Sadece “mutlu yolları” test ettiğinizi fark etmiş olabilirsiniz; her şeyin yolunda gittiği yollar. Bu fonksiyonların her birinin “üzgün yolu” vardı; sayacın mevcut olmadığı durum. Bu fonksiyonların her biri, sayaç mevcut değilse, üretimde sorun çıkarır. Sayaç mevcut olmadığında ne olacağını test etmek için kendi başınıza daha fazla test senaryosu yazıp yazamayacağınıza bakın. Ardından, bu test senaryolarını geçirecek kodu yazın.

Artık gerçek test tabanlı geliştirmeyi uygulamak için araçlara sahipsiniz. Bir sonraki zorluğunuz, bu araçları projelerinizde kullanmaktır. Bir sonraki projenizde test senaryolarını önce yazmayı deneyin, ardından bunları geçirecek kodu yazın. Sonra kodu ve test senaryolarını daha sağlam hale getirene kadar çalışın, böylece kodun tasarlandığı gibi çalıştığından emin olabilirsiniz.

## Author(s)

John J. Rofrano

**© IBM Corporation. Tüm hakları saklıdır.**