

22장. 테스트, 정말 거추장스럽고 낭비일까?

Index

- 어떻게 테스트를 구축할 것인가
- 단위 테스트 (Unit Test)
- 통합 테스트 (Integration Test)
- 지속적 통합 (Continuous Integration)
- 테스트할 시간이 어디 있다고!
- 테스트 범위 게임
- unittest의 대안

어떻게 테스트를 구축할 것인가

```
app
└── __init__.py
└── admin.py
└── apps.py
└── models.py
└── test_models.py
└── test_views.py
└── urls.py
└── views.py
```

다른 종류의 파일들이 있다면 똑같은 방식으로 `test_` 프리픽스를 붙여서 `forms.py`에 대한 테스트 파일은 `test_forms.py`처럼 생성한다. 이것은 파이썬의 도의 수평적인 것이 중첩된 것보다 낫다는 철학에 기반을 두었다.

단위 테스트

단위 테스트 방법론

1. 각 테스트 메서드는 테스트를 한 가지씩만 수행해야 한다.
 - 테스트 메서드는 그 범위가 좁아야 한다. 하나의 기능에 대해서만 테스트 가 이루어져야 한다.

2. 뷰에 대해서는 RequestFactory를 이용하자.

- django.test.client.RequestFactory는 모든 뷰에 대해 인자로 넘길 수 있는 요청 인스턴스를 제공한다.
- 하지만 생성된 이렇게 요청은 세션과 인증을 포함한 미들웨어를 지원하지 않는다(추가적인 작업 필요함).

3. 테스트가 필요한 테스트 코드를 작성하지 말자.

- 테스트는 가능한 한 단순하게 작성해야 한다.

4. 같은 일을 반복하지 않는다 는 테스트 케이스를 쓰는 데는 적용되지 않는다.

- 테스트 유ти리티를 굳이 공들여 만들어놓을 필요는 없다.
- 같은 작업이 필요하다면 복사, 붙여넣기를 사용해도 괜찮다.

5. 픽스처를 너무 신뢰하지 말자.

- 픽스처를 사용하면 유지하기가 매우 어렵다.
- 차라리 ORM 의존적인 코드를 제작하는 편이 쉽다.
- 테스트 데이터를 생성해주는 도구인([factory boy](#), [model mommy](#), [mock](#)) 같은 것들을 사용해보는 것도 좋다.

6. 테스트해야 할 대상들

- 전부 다해라
- 뷰: 데이터의 뷰, 데이터 변경 그리고 커스텀 클래스에 기반을 둔 뷰 메서드
- 모델: 모델의 생성, 수정, 삭제, 모델의 메서드, 모델 관리 메서드
- 폼: 폼 메서드, clean() 메서드, 커스텀 필드
- 유효성 검사기: 커스텀 유효성 검사기에 대해 다양하고 심도있는 테스트 케이스가 필요하다.
- 시그널: 시그널은 원격에서 작동하기에 테스트를 하지 않을 경우 문제를 야기하기 쉽다.
- 필터: 기본적으로 한 개 또는 두 개의 인자를 넘겨받는 메서드이므로 크게 어렵지 않다.
- 템플릿 태그: 기능이 막강하기 때문에 더더욱 테스트 케이스를 잘 작성해야 한다. 하지만 때때로 까다로울 수도 있다.
- 기타: 이메일, 콘텍스트 프로세서, 미들웨어, 기타 등등

7. 테스트의 목적은 테스트의 실패를 찾는 데 있다.

- 미리 실패를 인지해서 문제점이 무엇인지 정확하게 인지하고, 사용자가 보다 나은 품질로 서비스를 즐길 수 있게 한다.

8. 목(Mock)을 이용해서 실제 데이터에 문제를 일으키지 않고 단위 테스트 하기

- 단위 테스트는 단위 테스트 자체가 호출하는 함수나 메서드 이외의 것은 테스트하지 않도록 구성되어 있다.
- 이 말인즉슨, 외부 API에 대한 접속이나 이메일 수신 등 외부 환경의 영향을 받아선 안된다는 의미이다.
- 이런 경우 두가지 해법이 있다.
- 선택 1: 단위 테스트 자체를 통합 테스트로 변경한다.
- 선택 2: 목(Mock) 라이브러리를 이용해서 외부 API에 대한 가짜 응답을 만들어낸다.
- 목(Mock) 라이브러리는 테스트를 위한 값들을 매우 빠르게 이용할 수 있는 몽키 패치 라이브러리를 제공한다.
- 이렇게 되면 외부 API에 대한 테스트가 아닌 우리가 작성한 코드에 대한 테스트가 이루어지는 것이다.

9. 단언 메서드 사용하기

- 단언(assertion) 메서드를 잘 이용하자(Python3 Unittest 공식문서 [링크](#))
- 유용하게 쓰이는 단언 메서드 리스트
- assertRaises()
- assertCountEqual() (in Python3)
- assertDictEqual()
- assertFormError()
- assertContains(): response status 200인지 체크
- assertHTMLEqual(): 빈칸을 무시하고 비교
- assertJSONEqual()

10. 테스트의 문서화

- 각 클래스, 메서드에 대해 독스트링('' '' ~ '' '')을 이용해서 문서화하는 것처럼 테스트도 똑같은 방식으로 목적과 방식을 문서화해준다.
- 문서화되지 않은 테스트 코드는 테스트를 불가능하게 할 수도 있으므로 문서화를 해라.
- 문서화해라. 두 번 해라.

통합 테스트

통합 테스트란? 개별적인 소프트웨어 모듈이 하나의 그룹으로 조합되어 테스트되는 것을 의미한다.

단위 테스트가 끝난 뒤 실행하는 것이 가장 이상적인 방법이다.

통합 테스트의 예시

- 브라우저에서 잘 작동하는지 확인하는 Selenium을 이용한 기능 테스트 하는 경우
- 서드 파티 API에 대한 가상의 목 응답을 대신하는 실제 테스팅하는 경우
- 외부로 나가는 요청에 대한 유효성을 검사하기 위한 [requestb.in](#) 같은 것과 연동하는 경우
- API가 잘 작동하는지 확인하기 위한 [runscope.com](#)을 이용하는 경우

통합 테스트는 모든 부분이 잘 작동하는지를 확인하는 훌륭한 방법이다.

하지만 이러한 통합 테스트에도 문제가 있다.

- 통합 테스트 세팅에 시간이 많이 잡아먹힐 수 있다.
- 단위 테스트와 비교하면 시간이 느리다.
- 여러의 원인을 찾기가 힘들다.
- 좀 더 주의깊게 코드를 작성해야 한다.

이러한 문제가 있지만 그 리스크만큼 훌륭한 테스트 방법이라고 한다.

지속적 통합

프로젝트 저장소에 새로운 코드가 커밋될 때마다 테스트를 실행하는 지속적 통합 서버를 두기를 추천한다.(e.g. Travis CI, Circle CI, Jenkins)

테스트할 시간이 어디 있다고!

테스트 작성을 미루면 그 당시에는 편할지 몰라도 나중에 리팩토링할 때나 기능 추가할 때 힘들다.

테스트 작성은 해두면 버전 업그레이드나 기능 추가, 리팩토링이 쉬워진다는 얘기를 예시를 들어 설명하고 있다.

테스트 범위 게임

coverage 패키지를 이용한 간단한 튜토리얼을 설명한다.

unittest의 대안

현재 two scoops of django 에서는 모든 예제가 unittest로만 이루어져 있다.

하지만 unittest는 코드를 많이 작성해야 한다라는 단점 때문에 `pytest-django` , `django-nose` 가 unittest의 대안으로 쓸만하다고 추천하고 있다.
더해서 함수 기반의 단순화된 테스트는 상속이 불가능하다는 단점을 가지고 있다.

만약 빈번하게 프로젝트 여러 곳에서 쓰이는 테스트라면 함수 기반의 테스트를 작성하는 방식은 지양해야 한다.

여기까지가 책의 내용이다.

테스트는 정말 중요한 부분이지만 실행에 옮기기 어려운 부분이기도 한 것 같다.

항상 Test Driven Development를 지향하지만 프로젝트를 하다보면 기한에 쫓겨 Test는 뒷전이고 기능부터 빠르게 구현할 때가 빈번하다.

다음 프로젝트는 꼭 Test를 작성하면서 해야겠다(~~라고는 하지만 안할것같다~~)

23장. 문서화에 집착하자

Index

- 파이썬 문서에 rst 이용하기
- rst로부터 스팽크스를 이용하여 문서 생성하기
- 어떤 문서들을 작성해야 하는가
- 문서화에 대한 자료
- Markdown
- 위키와 다른 문서화 방법들

파이썬 문서에 rst 이용하기

최근 경향을 보면 reStructuredText를 이용해서 문서 작성은 많이 하는 것 같다고 말하고 있다(~~나는 이 글을 작성할 때 Markdown을 사용했...~~)
핵심 명령을 간추려보겠다.

각 섹션의 헤더

강조(Bold**)**

*이탤릭(*italic*)*

기본 링크: <https://djangoproject.com>

구문에 링크 달기: '*Django DOCUMENTATION*' _

.. _*Django DOCUMENTATION*: <https://djangoproject.com>

서브 섹션의 헤더

- #) 번호를 가진 리스트 아이템
 - #) 두 번째 아이템
- * 첫 번째 목록 기호
 - * 두 번째 목록 기호
 - * 들여 쓴 목록 기호
 - * 들여 쓴 상태에서 줄 바꾸기

코드 블록::

```
def like():
    print('I like it!')

def i in range(10):
    like()
```

파이썬 코드 블록(highlighting을 위해서는 pygments가 필요):

```
code-block:: python
    # pip install pygments

    for i in range(10):
        print(i)
```

자바스크립트 코드 블록(highlighting):

```
code-block:: javascript
    console.log("Don't use alert().");
```

rst로부터 스팍크스를 이용하여 문서 생성하기

Sphinx는 .rst 파일을 보기 좋게 꾸며진 문서로 변환해 주는 도구다.
주기적으로 스팍크스 문서를 빌드하는 것이 좋다.

어떤 문서들을 작성해야 하는가

개발 문서: 프로젝트를 셋업하고 관리하는 데 필요한 설명과 가이드라인

개발 문서에 들어가는 것들

- 설치
- 개발
- 아키텍쳐 노트
- 테스트 케이스를 실행하는 방법
- 코드의 PR 요청 방법 등등

꼭 필요하다고 생각되는 문서들

- README.rst
- docs/
- docs/deployment.rst
- docs/installation.rst
- docs/architecture.rst

문서화에 대한 자료

- [독스트링에 대한 공식 스펙 문서](#)
- [스핑크스 문서를 무료로 호스팅해 주는 서비스](#)
- [문서를 호스팅해 주는 또 다른 무료 서비스](#)

Markdown

마크다운(Markdown) 은 reStructuredText 와 크게 다르지 않은 텍스트 포맷 문법이다.

reStructuredText와 같은 내장 기능은 없지만 배우기 쉽다는 장점이 있다.

Python Package Index는 rst를 제외한 다른 포맷의 문서에선

`long_description` 을 지원하지 않는다.

Pandoc(팬독)은 한 마크업 언어를 다른 마크업 언어로 변환해주는 도구이다.

위키와 다른 문서화 방법들

어떤 이유에서 문서를 프로젝트 안에 위치시킬 수 없을 경우에는 다른 방법을 써서라도 문서를 제공해야 한다.

위키나 온라인에 문서를 저장하는 방식 또는 VCS를 이용할 수 없는 워드 프로세스 문서 형식이라도 문서가 없는 것 보다는 낫다.

개인적인 소감

여기까지가 책의 내용인데 개인적으로도 이 책의 내용과 생각이 비슷하다.
한 가지 다른 점을 꼽자면 나는 rst를 사용하지 않고 md를 사용하는 것 정도를
꼽을 수 있다.

이제부터는 rst도 써봐야겠다.(물론 저자가 md를 까는건 별로 좋지 않다고 생
각한다.)

감사합니다 :D