

Testing in DJANGO

Unit testing

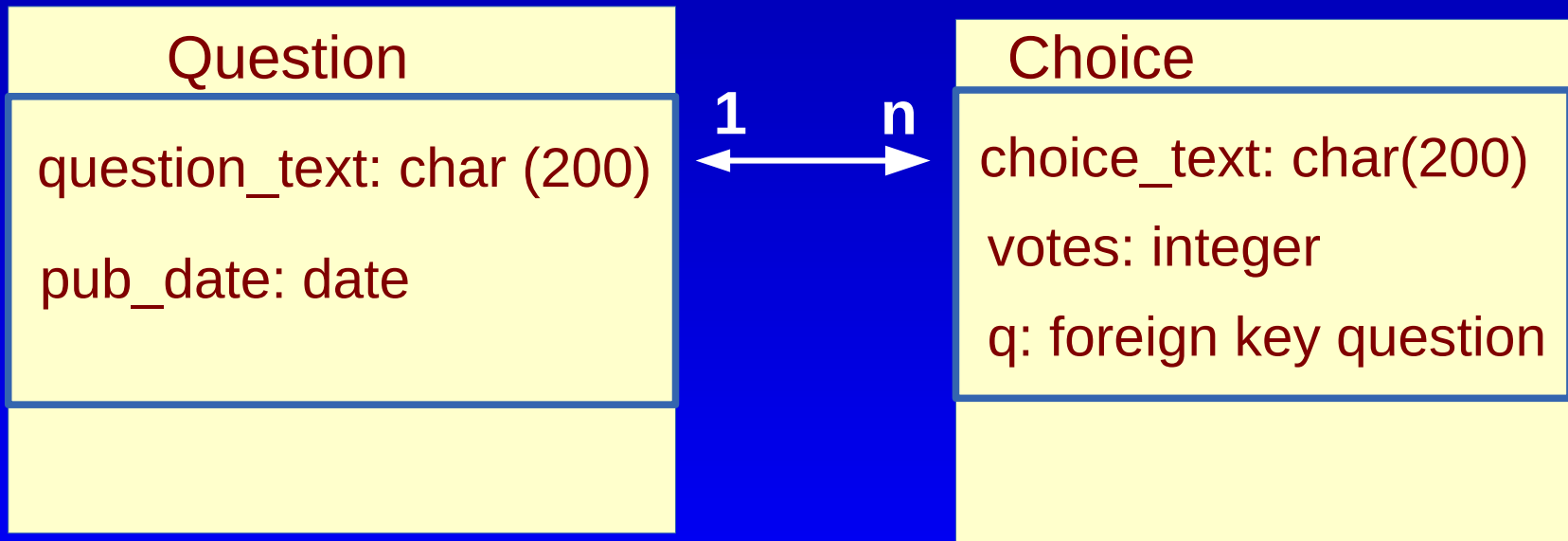
Necessità di unit testing automatizzati per applicazioni Web complesse

- Potenzialmente **decine di interazioni complesse** tra i componenti del sistema: cambiamenti su un componente possono avere **conseguenze inaspettate** sulle interazioni con le altre parti
- **Django supporta:**
 - Meccanismo per **definire** i test
 - Meccanismo per **lanciare** i test
- Cosa iniziare a testare?

Semplice applicazione polls

- Semplice **applicazione che gestisce sondaggi (poll)**
- Un sito pubblico che permette agli utenti di vedere i poll esistenti e di votare

<https://git.hipert.unimore.it/ncapodiecitechweb/-/tree/main/django/mysite>



Note

- Il database può essere popolato da una funzione che trovate in `mysite/initcmds.py` in cui si costruisce un database di Question & Choice usando <https://opentdb.com/> (sono domande e risposte multiple a quesiti di informatica)
- Question ha quindi 50 domande, indicizzate con pk a partire da 151
- Le choices sono (50*4), indicizzate con pk a partire da 601
- Sempre in `mysite/initcmds.py` ci sono due funzioni: una resetta l'intero DB, l'altra la popola (`erase_` ed `init_db`)



Ritorna un booleano

File models.py

- File polls/models.py:

```
from django.db import models
```

```
# Create your models here.
```

```
class Question(models.Model):
```

```
    question_text = models.CharField(max_length=200)
```

```
    pub_date = models.DateTimeField('date published')
```

```
    def was_published_recently(self):  
        return self.pub_date >= timezone.now() - datetime.timedelta(days=1)
```

Ritorna un booleano

```
class Choice(models.Model):
```

```
    question = models.ForeignKey(Question)
```

```
    choice_text = models.CharField(max_length=200)
```

```
    votes = models.IntegerField(default=0)
```

View di polls

1) **Indice dei sondaggi** → `def index(request):`

`http://127.0.0.1:8000/polls/`

Mostra gli ultimi 20 poll pubblicati in ordine di data decrescente

2) **Dettagli di un sondaggio** → `def detail(request, question_id):`

`http://127.0.0.1:8000/polls/100/`

Mostra il dettaglio del poll indicato (100)

3) **Sondaggi recenti** → `def recent(request, npolls):`

`http://127.0.0.1:8000/polls/recent/10/`

Mostra gli ultimi poll (10) pubblicati di recente

4) **Risultato di un sondaggio** → `def results(request, question_id):`

`http://127.0.0.1:8000/polls/10/results/`

La view index ■

View index: ritorna una lista degli ultimi 20 sondaggi inseriti in ordine cronologico inverso e separati da ', '

```
# File polls/views.py
from django.http import HttpResponse
from .models import Question
```

```
def index(request):
    latest_question_list = Question.objects.order_by('-pub_date')[:20]
    template = loader.get_template('polls/index.html')
    context = {'latest_question_list': latest_question_list}
    return HttpResponse(template.render(context,request))
```

Accesso ai dati del DB:
ultimi 20 poll in ordine di
data decrescente



La view recent■

```
def recent(request, n_polls):  
    question_list_ord = Question.objects.order_by('-pub_date')  
    latest_question_list = []  
    for q in question_list_ord:  
        if q.was_published_recently():  
            latest_question_list.append(q)  
    context =  
    {'latest_question_list':latest_question_list[:int(n_polls)]}  
    return render(request, 'polls/index.html', context)
```

Qual'è il bug?

Bug in app polls

- Metodo `Question.was_published_recently()`
`self.pub_date >= timezone.now() -
datetime.timedelta(days=1)`
- Ritorna **True** se il sondaggio è pubblicato entro le ultime 24 ore (che è corretto) ma **ANCHE** se il sondaggio ha **data di pubblicazione nel futuro** (non corretto)
 - Scenario mai considerato in fase di progetto
 - Esempio dell'importanza del testing a **design-time**
- Creiamo un **test** che verifica il **comportamento corretto** di `was_published_recently()`
 - controlla il caso di post con `pub_date` nel futuro

Creazione di un test ■

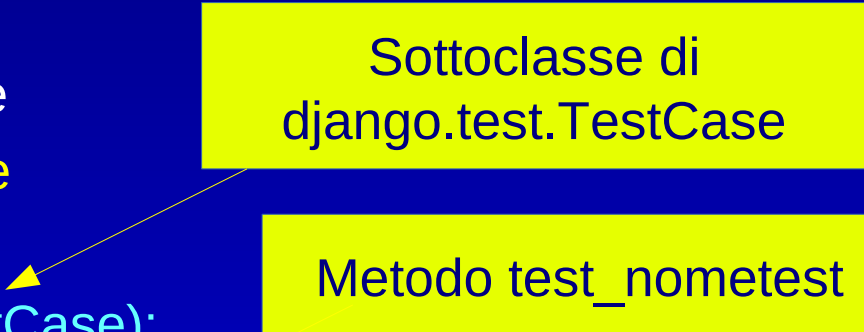
- Dove inserire i test?
 - File apposito per ogni applicazione `tests.py` (preimpostato di default ma vuoto)
- Che strumenti usare?
 - In Django **non** usiamo le funzioni offerte dal package `unittesting`
 - Package Django specifico
`from django.test import TestCase`
- Comportamento analogo
 - Creazione di una **classe di test** che eredita da `TestCase`
 - **Metodi:** nome `test_nometest` con asserzioni

Creazione del test

- File `polls/tests.py`

```
import datetime
from django.utils import timezone
from django.test import TestCase
from .models import Question
class QuestionMethodTests(TestCase):
```

Sottoclasse di
`django.test.TestCase`



Metodo `test_nometest`



```
    def test_was_published_recently_with_future_question(self):
```

```
        """
```

```
        was_published_recently() should return False for questions whose
        pub_date is in the future
```

```
        """
```

Creazione post con data nel futuro



```
        time = timezone.now() + datetime.timedelta(days=30)
```

```
        future_question = Question("Question?", pub_date=time)
```

```
        self.assertEqual(future_question.was_published_recently(), False)
```

Assertion



Esecuzione di un test

- Uso di manage.py
`python manage.py test polls`
- Il sistema di testing va a cercare i test contenuti nel file `tests.py` della app polls
- Nota: il testing lavora su un **database di prova** creato ad hoc di nome `test_nome-db-django`
- Nel nostro caso : “test_dbdjango”
Creating test database for alias 'default'...
- **Non si sporca lo scenario di lavoro dell'applicazione**

Possibili problemi

- **Permessi!!**
- Il sistema tenta di creare un DB di test col nome `test_nomedbdjango` (nomedbdjango = nome db usato nel progetto) con i permessi dello user impostato per il progetto mysite sul DB (es. user MySQL `django`user)
- **Possibile errore (es. in MySQL):**
Got an error creating the test database: (1044, "Access denied for user 'userdjango'@'localhost' to database 'test_dbdjango'")
- Dobbiamo dare a `django`user diritti di creazione sul DB `test_djangodb`
- Entrare in `mysql` con utente `root`, poi:
`mysql> grant all on test_dbdjango.* to 'userdjango'@'localhost';`

Output del test

Creating test database for alias 'default'...

F

=====

FAIL: test_was_published_recently_with_future_question
(polls.tests.QuestionMethodTests)

Traceback (most recent call last):

File "/path/to/mysite/polls/tests.py", line 16, in
test_was_published_recently_with_future_question
self.assertEqual(future_question.was_published_recently(), False)

AssertionError: True != False

Ran 1 test in 0.001s

FAILED (failures=1)

Destroying test database for alias 'default'...



DB di test distrutto
alla fine del test


Cosa è successo a basso livello

- `python manage.py test polls` → cerca i test nel file `polls/tests.py`
- Cerca nel file una **sottoclasse** di `django.test.TestCase`
- **Crea un DB** per il testing: `test_dbdjango`
- Cerca **metodi di test** (nome `“test_***”`)
- Esegue i metodi
`test_was_published_recently_with_future_question`: crea un sondaggio con data fra 30 giorni
- **Statement assert fallisce**: si aspetta `False`, invece il metodo ritorna `True`

Correzione del bug

- Nel file `model.py` ridefiniamo il metodo `was_published_recently`

```
def was_published_recently(self):  
    now = timezone.now()  
    return now - datetime.timedelta(days=1) <= \
```



```
self.pub_date <= now
```

- Ora possiamo tornare a invocare il test:

Creating test database for alias 'default'...

Ran 1 test in 0.001s

OK

Destroying test database for alias 'default'...

Esercizio: definire nuovi test

- Assicuriamoci di **non aver introdotto nuovi errori** con la modifica effettuata – **testiamo completamente** il metodo `was_published_recently`
- Definiamo **tre scenari per testare il metodo che abbiamo modificato**:
 - Lo scenario del poll futuro è già stato definito e incluso nei test
 - Creiamo un poll vecchio di 30 giorni → non deve risultare pubblicato recentemente
 - Creiamo un poll vecchio di 1 ora → deve risultare pubblicato recentemente

Approccio Black box testing - Equivalence partitioning

Test sondaggio nel passato

```
def test_was_published_recently_with_old_question(self):  
    """  
    was_published_recently() should return False for  
    questions whose pub_date is older than 1 day  
    """  
    time = timezone.now() - datetime.timedelta(days=30)  
    old_question = Question("Old", pub_date=time)  
    self.assertEqual(old_question.was_published_recently(), \n                      False)
```

Test sondaggio recente

```
def test_was_published_recently_with_recent_question(self):  
    """  
    was_published_recently() should return True for  
    questions whose pub_date is within the last day  
    """  
  
    time = timezone.now() - datetime.timedelta(hours=1)  
    recent_question = Question("Recent", pub_date=time)  
    self.assertEqual(recent_question.was_published_recently(), True)
```

Testare una view

- Il **modello di testing** descritto va bene per verificare i **modelli** → **codice Python interno**
- Il comportamento delle **view** risulta più **complesso da testare**
 - Verificare il **comportamento dell'applicazione** così come viene **percepito dall'utente finale**
- Servono **strumenti appositi**
- Cosa mette a disposizione Django per questo tipo di test?
 - Focus su package **django.test.utils**

Strumenti per testare view

- **Django Test Client**
- Strumento per simulare l'azione di un utente che interagisce con il codice a livello delle view
 - Utilizzabile da tests.py o da shell

- Nel caso della **shell**, sono necessarie un paio di operazioni aggiuntive rispetto a tests.py

```
>>> from django.test.utils import setup_test_environment  
>>> setup_test_environment()
```

- Funzione per il settaggio dell'ambiente: sostituisce il renderer di default per i template con uno strumento più adatto al testing che ci dà **accesso ad attributi della risposta**, come **response.context**, altrimenti non visibili

Testare una view (shell)

- Strumento per mandare richieste al server: classe di test **Client** (compresa anche in `django.test.TestCase`)

```
>>> from django.test.client import Client
```

```
# create an instance of the client for our use
```

```
>>> client = Client()
```

Possiamo fare richieste e interrogare l'oggetto response

```
# HTTP GET request to not existing URL '/mysite/'
```

```
>>> response = client.get('/mysite/')
```

```
# HTTP response = status code 404 from that address
```

```
>>> response.status_code
```

```
404
```



Oggetto response: dà accesso ai parametri della risposta

Testare una view (shell)

- Il client può usare **funzioni** per accedere al **controller**
- Es: funzione **reverse**

```
>>> from django.core.urlresolvers import reverse
```

```
>>> response = client.get(reverse('polls:index'))
```

```
>>> response.status_code
```

```
200
```

Accesso al controller
per ottenere gli URL

- Altro uso dell'oggetto **response**: **vedere il contenuto**

```
>>> response.content
```

```
"\n\n<link rel=\"stylesheet\" type=\"text/css\"
```

```
href=\"/static/polls/style.css\" />\n\n\n  <ul>\n    \n  <li><a
```

```
href=\"/polls/1/\">Qual'è il tuo colore preferito?</a></li>\n    \n
```

```
</ul>\n\n\n\n"
```

Contenuto della risposta

Oggetto response

- L'**oggetto response** consente di accedere a diversi tipi di dati:
- Dati della **risposta**
 - Status code
 - Content
- Informazioni sulla **richiesta generatrice**
 - Client
 - Request
- Strutture dati del **renderer**
 - Templates
 - Context

Oggetto response

```
>>> response.context['latest_question_list']  
[<Question: Qual'è il tuo colore preferito?>]
```

Accesso a variabile di template latest_question_list

Possibilità di seguire eventuali **redirezioni HTTP** (parametro **follow=True**)

```
>>> response = client.get('/redirect_me/', follow=True)
```

Il client segue tutte le redirezioni e nell'attributo **redirect_chain** di response restituisce una lista di tuple binarie con gli URL intermedi e i relativi status code

```
>>> response.redirect_chain  
[('http://testserver/next/', 302), ('http://testserver/final/', 200)]
```

Client di test

Il client permette l'invio di **richieste** con i parametri che ci interessano

- Metodo (GET o POST)
- URL (parametri hard-coded o impostati accedendo al controller)
 - Parametri passabili a reverse attraverso il keyword argument `args=(,)`
- Gestione delle redirezioni
- Autenticazione (metodo login)
- Cookies
- ...

Bug nella view IndexView ('/polls/')

- Nota: la view fino ad ora creata **non gestisce correttamente i poll con data nel futuro** (non vogliamo che siano visualizzati all'utente fino alla data di pubblicazione)
- Codice incriminato – view IndexView:

```
class IndexView(generic.ListView):
```

```
    template_name = 'polls/index.html'
```

```
    context_object_name = 'latest_question_list'
```

```
    def get_queryset(self):
```

```
        """Return the last five published questions."""
```

```
        return Question.objects.order_by('-pub_date')[:20]
```

Versione class-based della view

Modifica della view IndexView

- Il problema è a carico della **funzione get_queryset()**
- Possiamo **modificare** la funzione come segue:

```
from django.utils import timezone
```

```
....
```

```
def get_queryset(self):
```

```
    """Return the last 20 published questions not including those set to  
    be published in the future
```

```
    """
```

```
    return Question.objects.filter(  
        pub_date__lte=timezone.now()  
    ).order_by('-pub_date')[:20]
```

Filter torna un queryset
a cui posso applicare
altre API django

__lte: less than or equal

Testare la nuova view

- Creazione di un **nuovo test** per testare il comportamento della IndexView nei seguenti casi:
 - Assenza di sondaggi (controllo presenza di messaggio 'No polls are available' da template)
 - Presenza di soli sondaggi con pub_date passata
 - Presenza di soli sondaggi con pub_date futura
 - Presenza di entrambi i tipi di sondaggi
- Creiamo una funzione **create_question** da usare nei test per creare comodamente sondaggi con una specifica data di pubblicazione

Testare la nuova view

#file polls/tests.py

```
from django.core.urlresolvers import reverse
```

```
import datetime
```

```
from django.utils import timezone
```

```
from django.test import TestCase
```

```
from .models import Question
```

```
def create_question(question_text, days):
```

```
    """
```

Creates a question: days represent the offset to now: negative for questions published in the past, positive for those in the future

```
    """
```

```
    time = timezone.now() + datetime.timedelta(days=days)
```

```
    return Question.objects.create(question_text=question_text,  
                                   pub_date=time)
```

Metodo per creare
un nuovo sondaggio

days è un offset
rispetto ad oggi

Testare la nuova view

- Gestione caso in cui non ci sono poll nel DB

```
class QuestionViewTests(TestCase):
```

```
    def test_index_view_with_no_questions(self):
```

```
        """
```

No questions --> "No polls are available"

message should be displayed.

```
        """
```

`self.client` (ereditato da classe `TestCase`)

```
        response = self.client.get(reverse('polls:index'))
```

```
        self.assertEqual(response.status_code, 200)
```

```
        self.assertContains(response, "No polls are available.")
```

```
        self.assertQuerysetEqual(
```

```
            response.context['latest_question_list'], [ ]
```

Metodo `test_***`

Non crea polls, ma controlla codice, messaggio e che la var `latest_questions_list` sia vuota

Accetta Queryset → liste

Testare la nuova view

- I poll con pub_date passata devono essere mostrati

```
def test_index_view_with_a_past_question(self):
```

```
    """
```

Questions with a pub_date in the past should be displayed

```
    """
```

Controllo la var di template latest_question_list

```
    create_question(question="Past question.", days=-30)
```

```
    response = self.client.get(reverse('polls:index'))
```

```
    self.assertQuerysetEqual(
```

self.client (TestCase)

```
        response.context['latest_question_list'],
```

```
        ['<Question: Past question.>']
```

```
    )
```

Offset negativo



Testare la nuova view

- I sondaggi futuri non devono essere mostrati

```
def test_index_view_with_a_future_question(self):
```

```
.....
```

Questions with a pub_date in the future --> not shown

```
.....
```

```
create_question(question_text="Future question.", days=30)
```

```
response = self.client.get(reverse('polls:index'))
```


```
self.assertEqual(response.status_code, 200)
```

```
self.assertContains(response, "No polls are available")
```

```
self.assertQuerysetEqual(
```

```
response.context['latest_question_list'], [ ])
```

Effettuo gli
stessi controlli
del caso no polls



Nota: i poll vengono creati
solo nel DB di test e **il DB**
viene resettato ad ogni test!

Testare la nuova view

- **Controllo il caso misto (poll futuri e passati)**

```
def test_index_view_with_future_question_and_past_question(self):
```

```
    """
```

Even if both past and future polls exist, only past polls should be displayed.

```
    """
```

```
    create_question(question_text="Past poll.", days=-30)
    create_question(question_text="Future poll.", days=30)
    response = self.client.get(reverse('polls:index'))
    self.assertQuerysetEqual(
        response.context['latest_question_list'],
        ['<Question: Past poll.>']
    )
```

Considerazioni sul testing

- I test tendono ad aumentare rapidamente di numero e complessità
 - Si può arrivare ad avere più codice per i test che non per il software
 - Molto del codice dei test è simile: spesso si usa ed abusa di copia-e-incolla
- Questo non rappresenta un problema
 - I test non necessitano di grande manutenzione → la ridondanza del codice non è un problema
 - I test si scrivono una volta e restano per garantire che non ci siano regressioni → il loro numero non è un problema

Considerazioni sul testing

- Alcuni suggerimenti per tenere il **codice di testing gestibile**:
 - Creare una **classe di test separata** per ogni **classe di modello** e per ogni **view** da testare
 - Creare un **metodo di test** per ogni **condizione** da verificare
 - Usare **nomi di test esplicativi**, anche se lunghi
- Il testing visto fino ad ora funziona bene per la parte **server-side del sito**
- Ulteriori strumenti (**testing in-browser**) possono essere necessari per **codice client-side**