Table of Contents

Testing a Simple Home Page with Unit Tests

We finished the last chapter with a functional test (FT) failing, telling us that it wanted the home page for our site to have “To-Do” in its title. Time to start working on our application. In this chapter, we’ll build our first HTML page, find out about URL handling, and create responses to HTTP requests with Django’s view functions.

Warning: Things Are About to Get Real

The first two chapters were intentionally nice and light. From now on, we get into some more meaty coding. Here’s a prediction: at some point, things are going to go wrong. You’re going to see different results from what I say you should see. This is a Good Thing, because it will be a genuine character-building Learning Experience™.

One possibility is that I've given some ambiguous explanations, and you've done something different from what I intended. Step back and have a think about what we're trying to achieve at this point in the book. Which file are we editing, what do we want the user to be able to do, what are we testing and why? It may be that you've edited the wrong file or function, or are running the wrong tests. I reckon you'll learn more about TDD from these "stop and think" moments than you do from all the times when following instructions and copy-pasting goes smoothly.

Or it may be a real bug. Be tenacious, read the error message carefully (see Reading Tracebacks), and you'll get to the bottom of it. It's probably just a missing comma, or trailing slash, or a missing *s* in one of the Selenium find methods. But, as Zed Shaw memorably insisted in *Learn Python The Hard Way* (https://learnpythonthehardway.org), debugging is also an absolutely vital part of learning, so do stick it out! You can always drop me an email if you get really stuck. Happy debugging!

Our First Django App and Our First Unit Test

Django encourages you to structure your code into *apps*. The theory is that one project can have many apps; you can use third-party apps developed by other people, and you might even reuse one of your own apps in a different project…although I have to say, I've never actually managed the latter, myself! Still, apps are a good way to keep your code organised.

Let's start an app for our to-do lists:

```
$ python manage.py startapp lists
```

That will create a folder called *lists*, next to *manage.py* and the existing *superlists* folder, and within it a number of placeholder files for things like models, views, and, of immediate interest to us, tests:

```
.
├── db.sqlite3
├── functional_tests.py
├── lists
│   ├── __init__.py
│   ├── admin.py
│   ├── apps.py
│   ├── migrations
│   │   └── __init__.py
│   ├── models.py
│   ├── tests.py
│   └── views.py
├── manage.py
└── superlists
    ├── __init__.py
    ├── asgi.py
    ├── settings.py
    ├── urls.py
    └── wsgi.py
```

Unit Tests, and How They Differ from Functional Tests

As with so many of the labels we put on things, the line between unit tests and FTs can become a little blurry at times. The basic distinction, though, is that FTs test the application from the outside, from the user's point of view. Unit tests on the other hand test the application from the inside, from the programmer's point of view.

The TDD approach I'm demonstrating uses both types of test to drive the development of our application, and ensure its correctness. Our workflow will look a bit like this:

1. We start by writing a *functional test*, describing a typical example of our new functionality from the user's point of view.
2. Once we have an FT that fails, we start to think about how to write code that can get it to pass (or at least to get past its current failure). We now use one or more *unit tests* to define how we want our code to behave—the idea is that each line of production code we write should be tested by (at least) one of our unit tests.
3. Once we have a failing unit test, we write the smallest amount of *application code* we can—just enough to get the unit test to pass. We may iterate between steps 2 and 3 a few times, until we think the FT will get a little further.
4. Now we can rerun our FTs and see if they pass, or get a little further. That may prompt us to write some new unit tests, and some new code, and so on.

5. Once we're comfortable that the core functionality works end-to-end, we can extend out to cover more permutations and edge cases, using just unit tests now.

You can see that, all the way through, the FTs are driving what development we do from a high level, while the unit tests drive what we do at a low level.

The FTs don't aim to cover every single tiny detail of our app's behaviour; they are there to reassure us that everything is wired up correctly. The unit tests are there to exhaustively check all the lower-level details and corner cases. See Functional tests versus unit tests.

Table 1. Functional tests versus unit tests

Functional tests	**Unit tests**
One test per feature/user story	Many tests per feature
Tests from the user's point of view	Tests the code (i.e., the programmer's point of view)
Can test that the UI "really" works	Tests the internals—individual functions or classes
Provides confidence that everything is wired together correctly and works end-to-end	Can exhaustively check permutations, details, and edge cases
Can warn about problems without telling you exactly what's wrong	Can point at exactly where the problem is
Slow	Fast

> Functional tests should help you build an application that actually works, and guarantee you never accidentally break it. Unit tests should help you to write code that's clean and bug free.

Enough theory for now—let's see how it looks in practice.

Unit Testing in Django

Let's see how to write a unit test for our home page view. Open up the new file at *lists/tests.py*, and you'll see something like this:

lists/tests.py

```
from django.test import TestCase

# Create your tests here.
```

PYTHON

Django has helpfully suggested we use a special version of `TestCase`, which it provides. It's an augmented version of the standard `unittest.TestCase`, with some additional Django-specific features, which we'll discover over the next few chapters.

You've already seen that the TDD cycle involves starting with a test that fails, then writing code to get it to pass. Well, before we can even get that far, we want to know that the unit test we're writing will definitely be run by our automated test runner, whatever it is. In the case of *functional_tests.py* we're running it directly, but this file made by Django is a bit more like magic. So, just to make sure, let's make a deliberately silly failing test:

lists/tests.py (ch03l002)

```
from django.test import TestCase


class SmokeTest(TestCase):
    def test_bad_maths(self):
        self.assertEqual(1 + 1, 3)
```

PYTHON

Now, let's invoke this mysterious Django test runner. As usual, it's a *manage.py* command:

```
$ python manage.py test
Creating test database for alias 'default'...
Found 1 test(s).
System check identified no issues (0 silenced).
F
======================================================================
FAIL: test_bad_maths (lists.tests.SmokeTest.test_bad_maths)
 ----------------------------------------------------------------------
Traceback (most recent call last):
  File "...goat-book/lists/tests.py", line 6, in test_bad_maths
    self.assertEqual(1 + 1, 3)
    ~~~~~~~~~~~~~~~~^^^^^^^^^^
AssertionError: 2 != 3

 ----------------------------------------------------------------------
Ran 1 test in 0.001s

FAILED (failures=1)
Destroying test database for alias 'default'...
```

Excellent. The machinery seems to be working. This is a good point for a commit:

```
$ git status  # should show you lists/ is untracked
$ git add lists
$ git diff --staged  # will show you the diff that you're about to commit
$ git commit -m "Add app for lists, with deliberately failing unit test"
```

As you've no doubt guessed, the `-m` flag lets you pass in a commit message at the command line, so you don't need to use an editor. It's up to you to pick the way you like to use the Git command line; I'll just show you the main ones I've seen used. For me, the key rule of VCS hygiene is: *make sure you always review what you're about to commit before you do it*.

Django's MVC, URLs, and View Functions

Django is structured along a classic *model-view-controller* (MVC) pattern—well, *broadly*. It definitely does have models, but what Django calls "views" are really controllers, and the view part is actually provided by the templates, but you can see the general idea is there!

If you're interested, you can look up the finer points of the discussion in the Django FAQs (https://oreil.ly/fz-ne).

Irrespective of any of that, as with any web server, Django's main job is to decide what to do when a user asks for a particular URL on our site. Django's workflow goes something like this:

1. An HTTP *request* comes in for a particular URL.
2. Django uses some rules to decide which *view* function should deal with the request (this is referred to as *resolving* the URL).
3. The view function processes the request and returns an HTTP *response*.

So, we want to test two things:

1. Can we make this view function return the HTML we need?
2. Can we tell Django to use this view function when we make a request for the root of the site ("/")?

Let's start with the first.

Unit Testing a View

Open up *lists/tests.py*, and change our silly test to something like this:

lists/tests.py (ch03l003)

```
PYTHON
from django.test import TestCase
from django.http import HttpRequest   1
from lists.views import home_page


class HomePageTest(TestCase):
    def test_home_page_returns_correct_html(self):
        request = HttpRequest()   1
        response = home_page(request)   2
        html = response.content.decode("utf8")   3
        self.assertIn("<title>To-Do lists</title>", html)   4
        self.assertTrue(html.startswith("<html>"))   5
        self.assertTrue(html.endswith("</html>"))   5
```

What's going on in this new test? Well, remember, a view function takes an HTTP request as input, and produces an HTTP response. So, to test that:

1 We import the `HttpRequest` class so that we can then create a request object within our test. This is the kind of object that Django will create when a user's browser asks for a page.

2 We pass the `HttpRequest` object to our `home_page` view, which gives us a response. You won't be surprised to hear that the response is an instance of a class called `HttpResponse`.

3 Then, we extract the `.content` of the response. These are the raw bytes, the ones and zeros that would be sent down the wire to the user's browser. We call `.decode()` to convert them into the string of HTML that's being sent to the user.

4 Now we can make some assertions: we know we want an HTML `<title>` tag somewhere in there, with the words "To-Do lists" in it—because that's what we specified in our FT.

5 And we can do a vague sense-check that it's valid HTML by checking that it starts with an `<html>` tag, which gets closed at the end.

So, what do you think will happen when we run the tests?

```
$ python manage.py test
Found 1 test(s).
System check identified no issues (0 silenced).
E
======================================================================
ERROR: lists.tests (unittest.loader._FailedTest.lists.tests)
 ----------------------------------------------------------------------
ImportError: Failed to import test module: lists.tests
Traceback (most recent call last):
[...]
  File "...goat-book/lists/tests.py", line 3, in <module>
    from lists.views import home_page
ImportError: cannot import name 'home_page' from 'lists.views'
```

It's a very predictable and uninteresting error: we tried to import something we haven't even written yet. But it's still good news—for the purposes of TDD, an exception that was predicted counts as an expected failure. Because we have both a failing FT and a failing unit test, we have the Testing Goat's full blessing to code away.

At Last! We Actually Write Some Application Code!

It is exciting, isn't it? Be warned, TDD means that long periods of anticipation are only defused very gradually, and by tiny increments. Especially as we're learning and only just starting out, we only allow ourselves to change (or add) one line of code at a time—and each time, we make just the minimal change required to address the current test failure.

I'm being deliberately extreme here, but what's our current test failure? We can't import `home_page` from `lists.views`? OK, let's fix that—and only that. In *lists/views.py*:

lists/views.py (ch03l004)

```
from django.shortcuts import render

# Create your views here.
home_page = None
```

PYTHON

"You must be joking!" I can hear you say.

I can hear you because it's what I used to say (with feeling) when my colleagues first demonstrated TDD to me. Well, bear with me, and we'll talk about whether or not this is all taking it too far in a little while. But for now, let yourself follow along, even if it's with some exasperation, and see if our tests can help us write the correct code, one tiny step at a time.

Let's run the tests again:

```
[...]
  File "...goat-book/lists/tests.py", line 9, in
test_home_page_returns_correct_html
    response = home_page(request)
TypeError: 'NoneType' object is not callable
```

We still get an error, but it's moved on a bit. Instead of an import error, our tests are telling us that our `home_page` "function" is not callable. That gives us a justification for changing it from being `None` to being an actual function. At the very smallest level of detail, every single code change can be driven by the tests!

Back in *lists/views.py*:

lists/views.py (ch03l005)

```
from django.shortcuts import render


def home_page():
    pass
```

PYTHON

Again, we're making the smallest, simplest change we can possibly make, that addresses precisely the current test failure. Our tests wanted something callable, so we gave them the simplest possible callable thing: a function that takes no arguments and returns nothing.

Let's run the tests again and see what they think:

```
    response = home_page(request)
TypeError: home_page() takes 0 positional arguments but 1 was given
```

Once more, our error message has changed slightly, and is guiding us towards fixing the next thing that's wrong.

The Unit-Test/Code Cycle

We can start to settle into the TDD *unit-test/code cycle* now:

1. In the terminal, run the unit tests and see how they fail.
2. In the editor, make a minimal code change to address the current test failure.

And repeat!

The more nervous we are about getting our code right, the smaller and more minimal we make each code change—the idea is to be absolutely sure that each bit of code is justified by a test.

This may seem laborious—and at first, it will be. But once you get into the swing of things, you'll find yourself coding quickly even if you take microscopic steps—this is how we write all of our production code at work.

Let's see how fast we can get this cycle going:

- Minimal code change:

  lists/views.py (ch03l006)

  ```
  def home_page(request):
      pass
  ```
  PYTHON

- Tests:

```
    html = response.content.decode("utf8")
           ^^^^^^^^^^^^^^^^
AttributeError: 'NoneType' object has no attribute 'content'
```

- Code—we use `django.http.HttpResponse`, as predicted:

lists/views.py (ch03l007)

```
from django.http import HttpResponse


def home_page(request):
    return HttpResponse()
```

- Tests again:

```
AssertionError: '<title>To-Do lists</title>' not found in ''
```

- Code again:

lists/views.py (ch03l008)

```
def home_page(request):
    return HttpResponse("<title>To-Do lists</title>")
```

- Tests yet again:

```
    self.assertTrue(html.startswith("<html>"))
AssertionError: False is not true
```

- Code yet again:

lists/views.py (ch03l009)

```
def home_page(request):
    return HttpResponse("<html><title>To-Do lists</title>")
```

- Tests—almost there?

```
        self.assertTrue(html.endswith("</html>"))
    AssertionError: False is not true
```

- Come on, one last effort:

  lists/views.py (ch03l010)

  ```
  def home_page(request):
      return HttpResponse("<html><title>To-Do lists</title></html>")
  ```
  PYTHON

- Surely?

```
$ python manage.py test
Creating test database for alias 'default'...
Found 1 test(s).
System check identified no issues (0 silenced).
.
 ----------------------------------------------------------------------
Ran 1 test in 0.001s

OK
Destroying test database for alias 'default'...
```

Hooray! Our first ever unit test pass! That's so momentous that I think it's worthy of a commit:

```
$ git diff  # should show changes to tests.py, and views.py
$ git commit -am "First unit test and view function"
```

That was the last variation on `git commit` I'll show, the `a` and `m` flags together, which adds all changes to tracked files and uses the commit message from the command line.[1]

`git commit -am` is the quickest formulation, but also gives you the least feedback about what's being committed, so make sure you've done a `git status` and a `git diff` beforehand, and are clear on what changes are about to go in.

Our Functional Tests Tell Us We're Not Quite Done Yet

We've got our unit test passing, so let's go back to running our FTs to see if we've made progress. Don't forget to spin up the dev server again, if it's not still running.

```
$ python functional_tests.py
F
======================================================================
FAIL: test_can_start_a_todo_list
(__main__.NewVisitorTest.test_can_start_a_todo_list)
 ----------------------------------------------------------------------
Traceback (most recent call last):
  File "...goat-book/functional_tests.py", line 18, in
test_can_start_a_todo_list
    self.assertIn("To-Do", self.browser.title)
    ~~~~~~~~~~~~~^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
AssertionError: 'To-Do' not found in 'The install worked successfully!
Congratulations!'

 ----------------------------------------------------------------------
Ran 1 test in 1.609s

FAILED (failures=1)
```

Looks like something isn't quite right. This is the reason we have functional tests!

Do you remember at the beginning of the chapter, we said we needed to do two things: firstly, create a view function to produce responses for requests, and secondly, tell the server which functions should respond to which URLs? Thanks to our FT, we have been reminded that we still need to do the second thing.

How can we write a test for URL resolution? At the moment, we just test the view function directly by importing it and calling it. But we want to test more layers of the Django stack. Django, like most web frameworks, supplies a tool for doing just that, called the Django test client (https://docs.djangoproject.com/en/5.2/topics/testing/tools/#the-test-client).

Let's see how to use it by adding a second, alternative test to our unit tests:

lists/tests.py (ch03l011)

```python
class HomePageTest(TestCase):
    def test_home_page_returns_correct_html(self):  # 1
        request = HttpRequest()
        response = home_page(request)
        html = response.content.decode("utf8")
        self.assertIn("<title>To-Do lists</title>", html)
        self.assertTrue(html.startswith("<html>"))
        self.assertTrue(html.endswith("</html>"))

    def test_home_page_returns_correct_html_2(self):
        response = self.client.get("/")  # 2
        self.assertContains(response, "<title>To-Do lists</title>")  # 3
```

1 This is our existing test.

2 In our new test, we access the test client via `self.client`, which is available on any test that uses `django.test.TestCase`. It provides methods like `.get()`, which simulates a browser making HTTP requests, and takes a URL as its first parameter. We use this instead of manually creating a request object and calling the view function directly.

3 Django also provides some assertion helpers like `assertContains`, which save us from having to manually extract and decode response content, and have some other nice properties besides, as we'll see.

Let's see how that works:

```
$ python manage.py test
Found 2 test(s).
Creating test database for alias 'default'...
System check identified no issues (0 silenced).
.F
======================================================================
FAIL: test_home_page_returns_correct_html_2
(lists.tests.HomePageTest.test_home_page_returns_correct_html_2)
 ----------------------------------------------------------------------
Traceback (most recent call last):
  File "...goat-book/lists/tests.py", line 17, in
test_home_page_returns_correct_html_2
    self.assertContains(response, "<title>To-Do lists</title>")
[...]
AssertionError: 404 != 200 : Couldn't retrieve content: Response code was 404
(expected 200)

 ----------------------------------------------------------------------
Ran 2 tests in 0.004s

FAILED (failures=1)
Destroying test database for alias 'default'...
```

Hmm, something about 404s? Let's dig into it.

Reading Tracebacks

Let's spend a moment talking about how to read tracebacks, as it's something we have to do a lot in TDD. You soon learn to scan through them and pick up relevant clues:

```
======================================================================
FAIL: test_home_page_returns_correct_html_2   2
(lists.tests.HomePageTest.test_home_page_returns_correct_html_2)   2
 ----------------------------------------------------------------------
Traceback (most recent call last):
  File "...goat-book/lists/tests.py", line 17, in
test_home_page_returns_correct_html_2
    self.assertContains(response, "<title>To-Do lists</title>")   3
    ~~~~~~~~~~~~~~~~~~^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^   4
AssertionError: 404 != 200 : Couldn't retrieve content: Response code was 404   1
(expected 200)

 ----------------------------------------------------------------------
[...]
```

1. The first place you look is usually *the error itself*. Sometimes that's all you need to see, and it will let you identify the problem immediately. But sometimes, like in this case, it's not quite self-evident.
2. The next thing to double-check is: *which test is failing?* Is it definitely the one we expected —that is, the one we just wrote? In this case, the answer is yes.
3. Then we look for the place in *our test code* that kicked off the failure. We work our way down from the top of the traceback, looking for the filename of the tests file to check which test function, and what line of code, the failure is coming from. In this case, it's the line where we call the `assertContains` method.
4. In Python 3.11 and later, you can also look out for the string of carets, which try to tell you exactly where the exception came from. This is more useful for unexpected exceptions than for assertion failures like we have now.

There is ordinarily a fifth step, where we look further down for any of *our own application code* that was involved with the problem. In this case, it's all Django code, but we'll see plenty of examples of this fifth step later in the book.

Pulling it all together, we interpret the traceback as telling us that:

- When we tried to do our assertion on the content of the response.
- Django's test helpers failed, saying that they could not do that.
- Because the response is an HTML 404 Not Found error, instead of a normal 200 OK response.

In other words, Django isn't yet configured to respond to requests for the root URL ("/") of our site. Let's make that happen now.

urls.py

Django uses a file called *urls.py* to map URLs to view functions. This mapping is also called *routing*. There's a main *urls.py* for the whole site in the *superlists* folder. Let's go take a look:

superlists/urls.py

```
"""
URL configuration for superlists project.

The `urlpatterns` list routes URLs to views. For more information please see:
    https://docs.djangoproject.com/en/5.2/topics/http/urls/
Examples:
Function views
    1. Add an import:  from my_app import views
    2. Add a URL to urlpatterns:  path('', views.home, name='home')
Class-based views
    1. Add an import:  from other_app.views import Home
    2. Add a URL to urlpatterns:  path('', Home.as_view(), name='home')
Including another URLconf
    1. Import the include() function: from django.urls import include, path
    2. Add a URL to urlpatterns:  path('blog/', include('blog.urls'))
"""

from django.contrib import admin
from django.urls import path

urlpatterns = [
    path("admin/", admin.site.urls),
]
```

PYTHON

As usual, lots of helpful comments and default suggestions from Django. In fact, that very first example is pretty much exactly what we want! Let's use that, with some minor changes:

superlists/urls.py (ch03l012)

```
from django.urls import path   1
from lists.views import home_page   2

urlpatterns = [
    path("", home_page, name="home"),   3
]
```

PYTHON

1. No need to import `admin` from `django.contrib`. Django's admin site is amazing, but it's a topic for another book.
2. But we will import our home page view function.

3 And we wire it up here, as a `path()` entry in the `urlpatterns` global. Django strips the leading slash from all URLs, so `"/url/path/to"` becomes `"url/path/to"` and the base URL is just the empty string, `""`. So this config says, the "base URL should point to our home page view".

Now we can run our unit tests again, with `python manage.py test`:

```
[...]
..
 ----------------------------------------------------------------------
Ran 2 tests in 0.003s

OK
```

Hooray!

Time for a little tidy-up. We don't need two separate tests, so let's move everything out of our low-level test that calls the view function directly, into the test that uses the Django test client:

lists/tests.py (ch03l013)

```
class HomePageTest(TestCase):
    def test_home_page_returns_correct_html(self):
        response = self.client.get("/")
        self.assertContains(response, "<title>To-Do lists</title>")
        self.assertContains(response, "<html>")
        self.assertContains(response, "</html>")
```

PYTHON

Why Didn't We Just Use the Django Test Client All Along?

You may be asking yourself, "Why didn't we just use the Django test client from the very beginning?" In real life, that's what I would do. But I wanted to show you the "manual" way of doing it first, for a couple of reasons. Firstly, because it enabled me to introduce concepts one by one, and keep the learning curve as shallow as possible. Secondly, because you may not always be using Django to build your apps, and testing tools may not always be available—but calling functions directly and examining their responses is always possible!

> The Django test client does also have disadvantages; later in the book (in [chapter_27_hot_lava]) we'll discuss the difference between fully isolated unit tests and the types of test that the test client pushes us towards (people often say these are technically "integration tests"). But for now, it's very much the pragmatic choice.

But now the moment of truth: will our functional tests pass?

```
$ python functional_tests.py
[...]
======================================================================
FAIL: test_can_start_a_todo_list
(__main__.NewVisitorTest.test_can_start_a_todo_list)
 ----------------------------------------------------------------------
Traceback (most recent call last):
  File "...goat-book/functional_tests.py", line 21, in
test_can_start_a_todo_list
    self.fail("Finish the test!")
AssertionError: Finish the test!
```

Failed? What? Oh, it's just our little reminder? Yes? Yes! We have a web page!

Ahem. Well, *I* thought it was a thrilling end to the chapter. You may still be a little baffled, perhaps keen to hear a justification for all these tests (and don't worry; all that will come), but I hope you felt just a tinge of excitement near the end there.

Just a little commit to calm down, and reflect on what we've covered:

```
$ git diff  # should show our modified test in tests.py, and the new config in urls.py
$ git commit -am "url config, map / to home_page view"
```

That was quite a chapter! Why not try typing `git log`, possibly using the `--oneline` flag, for a reminder of what we got up to:

```
$ git log --oneline
a6e6cc9 url config, map / to home_page view
450c0f3 First unit test and view function
ea2b037 Add app for lists, with deliberately failing unit test
[...]
```

Not bad—we covered the following:

- Starting a Django app
- The Django unit test runner
- The difference between FTs and unit tests
- Django view functions, and request and response objects
- Django URL resolving and *urls.py*
- The Django test client
- Returning basic HTML from a view

Useful Commands and Concepts

Running the Django dev server

```
python manage.py runserver
```

Running the functional tests

```
python functional_tests.py
```

Running the unit tests

```
python manage.py test
```

The unit-test/code cycle

1. Run the unit tests in the terminal.
2. Make a minimal code change in the editor.
3. Repeat!

1. I'm quite casual about my commit messages in this book, but in professional organisations or open source projects, people often want to be a bit more formal. Check out https://cbea.ms/git-commit and https://www.conventionalcommits.org.

Comments

ALSO ON **OBEY THE TESTING GOAT!**

Working Incrementally

2 years ago • 3 comments

Enough housekeeping with our URLs. It's time to bite the bullet and change our …

Saving User Input: Testing the Database

2 years ago • 4 comments

We've already had a hint of it, and now it's time to start to get to know the real power of …

Deployment Part 1: Containerization aka …

2 years ago • 10 comments

Is all fun and game until you are need of put it in production. Devops Borat It's time to …

Obey the T

8 years ago •

I'm still not g
Here you ge
of one tho. T

Emoji?

16 Responses

Upvote Funny Love Surprised Angry Sad

1 Comment Kgotso Koete ▾

Join the discussion…

♡ **Share** **Best** **Newest** **Oldest**

Said Maax

2 years ago

What a great start! Definitely got me excited for the rest of the book.

0 0 Reply