**In reference to Linkedin Learning Course Test-Driven Development in Django**

https://www.linkedin.com/learning-login/share?forceAccount=false&redirect=https%3A%2F%2Fwww.linkedin.com%2Flearning%2Ftest-driven-development-in-django%3Ftrk%3Dshare_ent_url&account=56973065

# Test-Driven Development in Django

- Don't write any code until you have a failed test.
- Write code to pass that test.
- So, for this one first test is going to be if there is a home page.
- Then we write the code to pass this test.
- Then we will test if there are hashes in the home page.
- And then we write the different tests and then code to pass these tests.
- So, we are creating a website that gives the hash code for a particular text and then when we click hash it also creates a page where we can come back to see its hash.

**Functional test vs Unit test**

- Functional test deals with the way a user interacts with your project, also known as functionality.
- Example:
    - If the user goes to your homepage, they will see the restaurant's logo.
    - If the user goes to the order page, they will be able to order a pizza.
    - If the user wants to change the order, they can.
- These tests are to make sure that the users can do things with your website.
- On the other hand, the unit test is to ensure that small pieces of your project are working as they should be.
- Example: The ability to create a Pizza object from the Pizza class.
- The function pizza_discription() will appropriately tell the size and topping of a pizza.
- A specific URL loads a specific view file.

So, the things that the user know and care about is a functional test. (Ordering a pizza) While things the user would never know about (Ensure pizza_description() is working properly) is a unit test.

**Writing a test in Selenium**

- First, we need to install selenium, for that we use the command pip install selenium.
- The we need something called as driver that is different for different browser that you are using.
- So, in my case I want with chromedriver for Google chrome.
- Chrome Drier is a separate executable that WebDriver uses to control Chrome. It is maintained with help from WebDriver contributors.
- So, the point is that by using this we can launch selenium test scripts in chrome.
- WebDriver driver = new ChromeDriver();
- But it didn't work then I tried it with Edge and it worked but I was not getting even the error messages that I was supposed to get.
- Then I shifted on the linux machine and it had the mozila installed so I tried installing the driver for that and it worked.
- So, then I installed Mozila on my pc and pasted the driver from the link:

    - Seleniumhq.org
    - And there you can find the driver for almost any browser.
    - Then extract and paste the driver exe file into the Scripts folder inside the Virtual Environment folder.
- This is to run test cases on google chrome. It is to make navigation on website automatic.
- To get it we download it extract it and then set the properties of the chrome driver by providing the path.
- Then we just go to the python file and create a test case:
- And we write code something like this:
    - from selenium import webdriver
    -
    - browser = webdriver.Chrome()
    - browser.get('http://localhost:8000')
    -
    - assert browser.page_source.find('install')
- So, this is our first test file.
- Then when you run the test it will open the browser and a 404 page will be there and you will get an error message say selenium was not able to find the page that you where looking for.
- Then in the next step we create a project name hashthat
    - Django-admin startproject hashthat
    - Then create an application in it but first cd into the project folder
        - Cd hashthat
        - Django-admin startapp hashing
    - Then you can run the server by
        - Python manage.py runserver
- If we look in the project folder in the hashing folder (that is the folder which has our application related files)
- There is a file name test.py
- So, we need to write the following code into it.
    - Just some required imports
    - from django.test import TestCase
    - from selenium import webdriver
    -
    - Class for functional Testing with inherits form TestCase.
    - class FunctionalTestCase(TestCase):
    - This is to start the browser
    - def setUp(self):
    - self.browser = webdriver.Firefox()
    -
    - This is the main test case all the test case should go in between the close and open browser method.
    - def test_there_is_homepage(self):
    - Check if there exists page mentione.
    - self.browser.get('http://127.0.0.1:8000')
    - Checks for the string in the webpage.
    - self.assertIn('install',self.browser.page_source)

- 
- Closes the browser.
- def tearDown(self):
- self.browser.quit()

**Now to create the test for hash function (Functional Test)**

- So, in the test file we add the following code:
  - def test_hash_of_hello(self):
  - self.browser.get('http://127.0.0.1:8000')
  - text = self.browser.find_element_by_id('id_text')
  - text.send_keys('hello')
  - self.browser.find_element_by_name('submit').click()
  - self.assertIn('2cf24dba5fb0a30e26e83b2ac5b9e29e1b161e5c1fa7425e73043362938b9824',self.browser.page_source)
  - def tearDown(self):
- So, this is creating another test in the test file.
- We define tests as methods.
- Then we are going to the home page and looking for a element by its id value.
- And the id value that it is searching for is id_text.
- Then it keys in the text hello in the text field with the id value 'id_text'.
- Then click the submit button.
- Then it checks if the text that is out hash value in the page by using the assertIn() function.

**Unit Testing**

- So, what we do is first we write the functional test that defines what the user wants to do then we write the unit tests that help us write the code that can fulfil the functional test.
- So, we write the following code for the first unit test case:
  - class UnitTestCase(TestCase):
  - def test_home_homepage_template(self):
  - Now this line is to get the home page response.
  - responce = self.client.get('/')
  - Then we check that that ok we have the response but is it using this template.
  - self.assertTemplateUsed(responce,'hashing/home.html')
  - Then run the test and it would fail.
- Now to pass this test.
  - So, for that first we need to add our application to the installed app list in the setting.py file.
  - Then we need to go to the urls.py file and add the url pattern for the home page.
  - So, whenever a request for the home page comes it goes to the views.home and we name it home too.
  - Also we need to import the views from our application first.
  - from hashing import views
    - urlpatterns = [
    - path('admin/', admin.site.urls),
    - path('',views.home, name = 'home'),

        - ]
    - Then we go to the views.py file.
        - from django.shortcuts import render
        -
        - def home(request):
        - return render(request,'hashing/home.html')
        - So, we create a method and its take request as the parameter and
        - Return a call to render function that takes the request and string pointing to the template for that page.
    - So, now we create templates\hashing\home.html.
    - And done forget the s in template.
    - Where the html code would go but for now, I put 'hello!'.
- So, now the next step is that we need a form where the information or the text can be entered and then it could be processed to get the hash value.
- So, the next unit test is going to check whether there is a form.
- For that first import the form:
    - From .forms import HashForm.
- Then we create a function name test_hash_form(self):. So, as we know that for creating a testcase in Django we create a function that must start with test.
    - def test_hash_form(self):
    - form = HashForm(data={'text':'hello'})
    - self.assertTrue(form.is_valid())
- So, what we did is we created a form and then passed dome data which is a dictionary and we passed hello as text.
- Then we assert that this is valid.
- Then we run the test and it will fail.
- Then we create a new file called forms.py in the hashing folder.
- Then code like this:
    - from django import forms
    - class HashForm(forms.Form):
    - text = forms.CharField(label='Enter hash here:', widget=forms.Textarea)
- First import the code to create a form.
- Then create a class HashForm and pass the forms.Form as parameter.
- Then we add a text field and that is a CharField and we pass label value as what we want to show as for that form.
- So, this label must match the test in the functional tests where we are looking for 'Enter text here:'. So, you can copy paste from there.
- Then we set the widget to forms.Textarea so that user have enough space to write the text.
- Now we have our form so let's go any display that by creating a view for it.
- So, first we import the form:
    - from .form import HashForm
- Then we create a form by:
    - Form = HashForm()
- Then we pass the form to the home.html template as a dictionary.
    - Return render( request, 'hadhing/home.html', {'form': form} )
- So, now we need to update our template to show the form.

  - So, action is something that happens after submission so we need to come back to home page to display the hash. So, action goes to home page. And the method is post. So, this is a reason that we need a csrf_token because we are using the post.
  - Csrf_token is for Cross Site Request Forgery protection. More on this on this website:
    - https://docs.djangoproject.com/en/3.0/ref/csrf/
  - <form action ="{% url 'home' %}" method="POST">
  - {% csrf_token %}
  - So, now we display the from and that as_p is to say that display this as the paragraph tag.
  - {{ form.as_p }}
  - Then we need a submit button. The value is the text that shows up in the button box.
  - <input name ="submit" type="submit" value="Hash">
  - </form>
- Now we move on to the hashing function.
- So, now we are just checking if the library is working fine.
- We do that by importing the hashlib which is for hashing.
  - Import hashlib
- And then create a test case like this:
  -
  - def test_hash_function_works(self):
  - So, in the hashlib we call the sha256() function then we pass the hello string and we need to encode the string as utf-8.
  - Then we do a hexdigest() this gives us a string.
  - text_hash = hashlib.sha256('hello'.encode('utf-8')).hexdigest()
  - self.assertEqual('2cf24dba5fb0a30e26e83b2ac5b9e29e1b161e5c1fa7425e73043362938b9824',text_hash)
  - This is to compare the hash value.
- Then we run the test and we will pass it.
- So, now we move on to the part where we want to save this hash values.
- To do that we create a model and this is the code that we write in the models.py file:
  - from django.db import models
  - Create a class that is our table name also it is inheriting the models.Model.
  - class Hash(models.Model):
  - Then we create variables that are going to be our columns.
  - text = models.TextField()
  - hash = models.CharField(max_length=64)
  - To reflect these changes into our database we will need to migrate these changes to our database.
  - To do that run:
    - Python manage.py makemigrations
    - Python manage.py migrate
- Now, we need to test the hash object that we just created in our model.
- So, to do that we first import the Hash model into our test.py
  - from .models import Hash

- Then we create a test case for the model test like this:
    - def test_hash_object(self):
    - Create an instance of Hash model
    - hash = Hash()
    - Set the values of parameters of hash object.
    - hash.text = 'hello'
    - hash.hash = '2cf24dba5fb0a30e26e83b2ac5b9e29e1b161e5c1fa7425e73043362938b9824'
    - Save it in the database.
    -
    - hash.save()
    - Pull the information out from the database and compare it to the values we previously set.
    - pulled_hash = Hash.objects.get(hash='2cf24dba5fb0a30e26e83b2ac5b9e29e1b161e5c1fa7425e73043362938b9824')
    - self.assertEqual(hash.text,pulled_hash.text)
- Then we create a test database.
- Go to the settings.py file and
    - DATABASES = {
    - 'default': {
    - 'ENGINE': 'django.db.backends.sqlite3',
    - 'NAME': os.path.join(BASE_DIR, 'db.sqlite3'),
    - We added this line so that whenever we do any testing the changes are made in the test database and will be deleted after the testing is done.
    - 'TEST_NAME': os.path.join(BASE_DIR, 'test_db.sqlite3'),
    - }
    - }
- So, now we would actually would like to save some data into the database.
- To do that go to views and import hash model.
- So, we modify our views file like this:
    - from django.shortcuts import render
    - from .form import HashForm
    - We imported the Hash model.
    - from .models import Hash
    - Also imported the library which we are going to use for hashing.
    - import hadhlib
    -
    -
    - def home(request):
    - First check if the request is of type post
    - if request.method == 'POST':
    - Then then set the variable filledform to the HashForm.
    - filled_form = HashForm(request.POST)
    - If filled form is valid.
    - if filled_form.is_valid():
    - Set the values of text and text_hash.

    - text = filled_form.cleaned_data['text']
    - Use the hashlib to get the hash value for that text.
    - text_hash = hashlib.sha256(text.encode('utf-8')).hexdigest()
    - Try if the entry for that hash already exist in the database.
    - try :
    - Hash.objects.get(hash=text_hash)
    - If not then instantiate the Hash model and save it in the database.
    - except Hash.DoesNotExist:
    - hash = Hash()
    - hash.text = text
    - hash.hash = text_hash
    - hash.save()
    -
    - form = HashForm()
    - return render(request,'hashing/home.html', {'form':form})
- Then we run our test and we are going to pass all the tests.
- Then we can check our website and see if everything is working.
- Then we need the ability to view our hash. So, first we will create a test for that.
- We then add the following line of code for the test creation.
    -
    - def test_viewing_hash(self):
    - So, we save the data into the database.
    - hash = self.saveHash()
    - Now we try to get a response from our client. This is to retrieve the page where the hash for that particular text is going to show up.
    - So, this will be the response that we get for the url that we are providing as link.
    - response = self.client.get('/hash/2cf24dba5fb0a30e26e83b2ac5b9e29e1b161e5c1fa7425e73043362938b9824')
    - Then we use the assert to check that the page contains the required or expected text string.
    - self.assertContains(response,'hello')
- Also, this code was redundant in the above function so we created a function for saving the text and the hash value for it into that database.
    - def saveHash(self):
    - hash = Hash()
    - hash.text = 'hello'
    - hash.hash = '2cf24dba5fb0a30e26e83b2ac5b9e29e1b161e5c1fa7425e73043362938b9824'
    - hash.save()
    - return hash
- Now to pass this test.
- So, first we add the url path in the urls.py file in url patterns.
- And this is the code that we add.
    - path('hash/<str:hash>', views.hash, name='hash'),
    - So, as we can see that the <str:hash> is the variable type string name hash.
    - And we send this request to views.hash.

    - Also, name is useful if we want to refer this page anywhere.
- Then we go into views.py.
- As we need to create a view for the above mentioned url we write a view like mentioned below:
    - So, we create a function called hash which takes request and the hash value from the url string.
    - def hash(request, hash):
    - Then we get the hash object from the database for which the hash value is the string that is provided in the url.
    - hash = Hash.objects.get(hash=hash)
    - Then we return a call to the render function which takes the request and the path to the template. Also, we pass the required data in a directory format.
    - return render (request,'hashing/hash.html',{'hash':hash})

**Advanced Testing**

- Using unit with functional tests.
- So just uncomment the functional tests and run the test and it should probably run fine and all the test will pass.
- Now, what if we gave some bad information like the hash we gave is too long.
- So, for that we import ValidationError from Django.code.exceptions
- And then we add the following code.
    - So, we are creating a function or a test for bad data and we pass it self.
    - def test_bad_data(self):
    - Then in that we define a function called badhash.
    - def badHash():
    - Then we create an instance of Hash model
    - hash = Hash()
    - And we pass a wrong hash value that is longer then what it should be.
    - hash.hash = '2cf24dba5fb0a30e26e83b2ac5b9e29e1b161e5c1fa7425e73043362938b9824hsaf'
    - So, the full clean is to rase a validation error for any error there occurs.
    - hash.full_clean()
    - The then we check that there is a validation error will occur.
    - self.assertRaises(ValidationError, badHash)

So, the Ajax part didn't work and probably needs some improvement.

- Waiting
- Its is sometime appropriate to wait for something to happen. Like when we are using ajax. And we expect some data to be returned. So, for that we need to test waiting.
    - def test_hash_ajax(self):
    - self.browser.get('http://127.0.0.1:8000')
    - text = self.browser.find_element_by_id('id_text')
    - text.send_keys('hello')
    - self.assertIn('2cf24dba5fb0a30e26e83b2ac5b9e29e1b161e5c1fa7425e73043362938b9824',self.browser.page_source)

- AJAX
- Add this in the urls
    - path('quickhash',views.quickhash, name='quickhash'),
- Then add this in views.
    - def quickhash(request):
    - text = request.GET['text']
    - return JsonResponse({'hash':hashlib.sha256(text.encode('utf-8')).hexdigest()})
- And lastly edit the home page for ajax script like this:
    - <form action ="{% url 'home' %}" method="POST">
    - {% csrf_token %}
    - {{ form.as_p }}
    - <input name ="submit" type="submit" value="Hash">
    - </form>
    -
    - <h2 id="quickhash"></h2>
    -
    - <script src="https://ajax.googleapis.com/ajax/libs/jquery/3.4.1/jquery.min.js"></script>
    -
    - <script>
    - $('#id_text').keyup(function () {
    - var text = $(this).val();
    -
    - $.ajax({
    - url: '/quickhash',
    - data: {
    - 'text': text
    - },
    - datatype:'json',
    - success: function (data) {
    - $('#quickhash').text(data['hash']);
    - }
    - })
    - });
    - </script>

- Testing and Deployment
- Deploy on the staging server and then run the functional test locally on the staging server and we run the unit test on the staging server.

- TDD vs test after
- Tests can take away joy, motivation, and speed.
- When a project is young, it changes too much for testing.
- You'll often delete functions and the test associated with them.
- So, for testing afterwards.

- Wait until you have the version 1.0.
- Create testing for those things you find crucial.
- Provide freedom and security that your code is working.