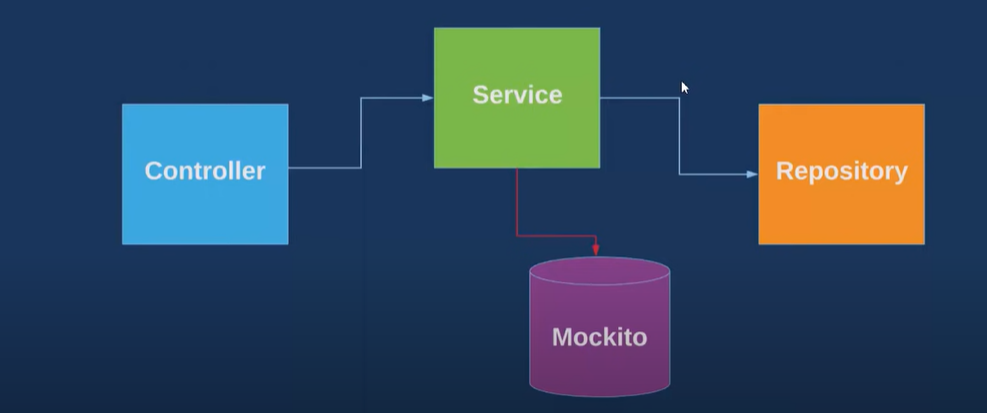
**Spring Boot Testing | Writing JUnit Tests using JUnit and Mockito(Service Layer Testing)**



In this tutorial we are going to discuss about Mockito framework like how to write unit test for my application using Mockito and in real time how people are using it. Okay? So before move to Mockito, first let's understand why we need unit test case.

So, first of all, we are writing test case to check whether my all functionalities working properly or not. So, let's go through an example. Assume one of my colleagues wrote an api and he didn't test it and luckily it went for the deployment.

So later on we found some functionality is not working as expected. Something like the screen alignment is not proper or else when I click on the find or search button, it populating the list of data and the particular field is getting null.

So like this, some issue may become if you are deploying your application without test. So to avoid such kind of issue, we are writing test case and it is also easy for other developers who will go through my test case.

He can easily identify okay, the exact api need, these are the request field and these are the response. Okay? So if somewhere he found any issue, he can easily do the debug and he can easily fix it, right?

So now let's discuss about the market. So why we need Mockito here and what are the advantages? So if you observe every application, every web application having three tire architecture, right? So one is the controller, another one is the service and the repository.

Whenever I am writing test case using our traditional approach without using Mockito, suppose I write a test case for controller. So first it will hit the controller method, then it will go to the surveys and then it will go to the repository.

So think to test our application we are hitting the. DB. So multiple time hitting the dv is not a good practice. So to avert such kind of issue Mockito it came into the picture. So what makito is doing here?

So, suppose I want to mock the service data or repository data. So, whatever the request is, come to the controller first it will forward to the service and instead of forward it to the repository and repository goes to the DB, it will mock the data here and it will behave like a database, and it will mock the data from here to service.

So that the request flow will not go to the DB for our testing purpose, that's the moto of the Mockito framework. So, same we'll do in action with our code. Okay? So, let's go to the eclipse already.

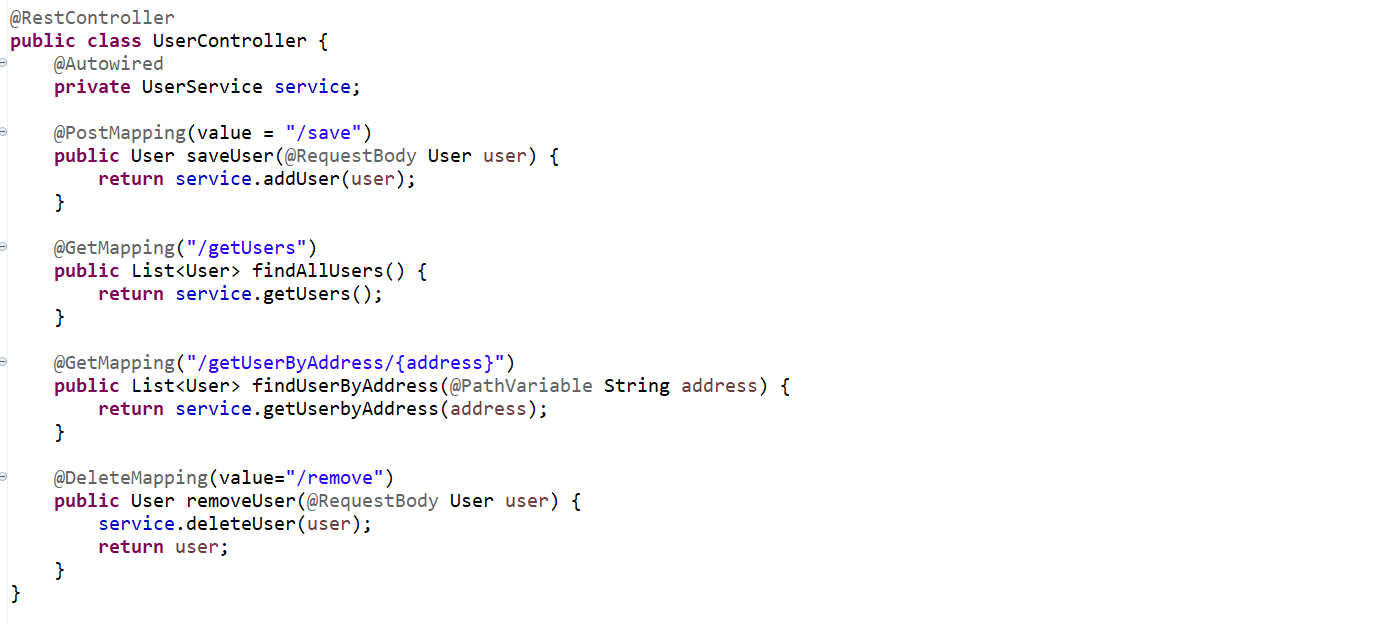
I created a dummy application with **CRUD** features.

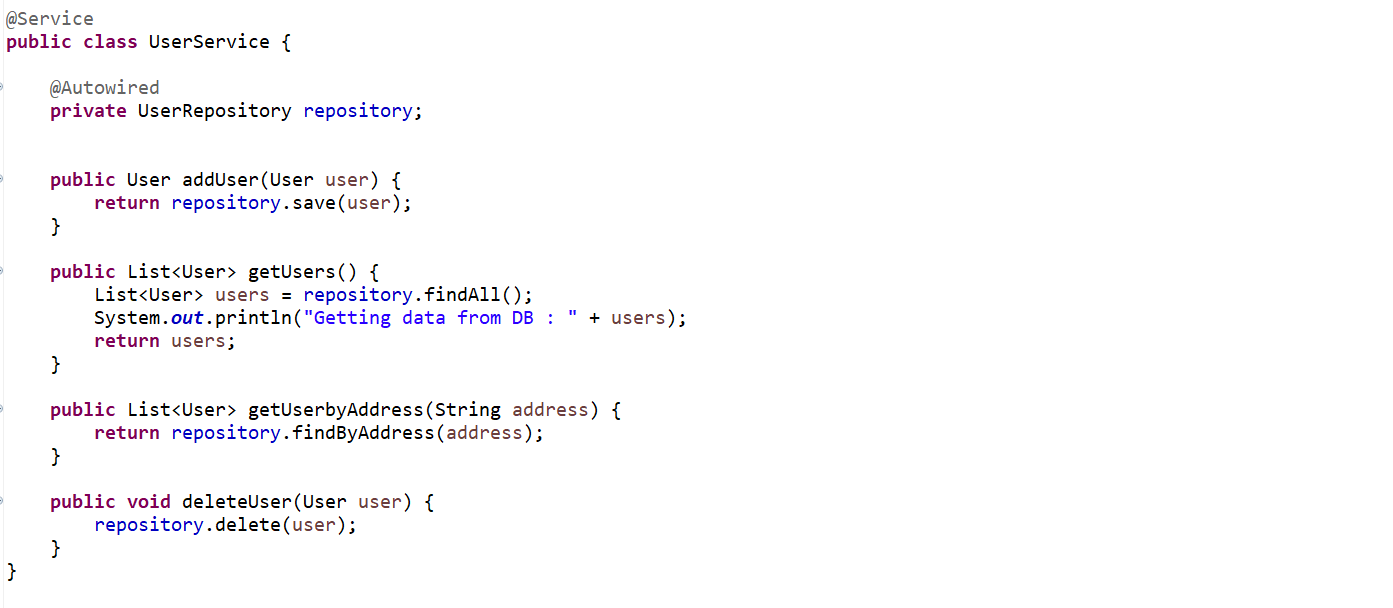
I have the three-layer controller and service and repository. I am using MongoDB here. So that's why I extend from mongo repository. And in this example, we are going to write the test case for our user service, and we are going to mock the data of our user repository.

App - **spring-boot-mockito**

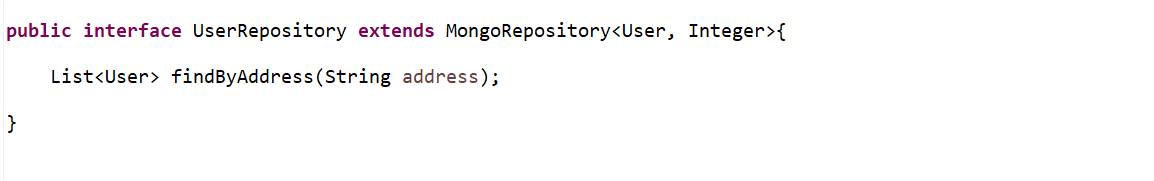
Dependencies – **Spring data MongoDB, Spring Dev Tool, Spring Web**

We have created a controller, model, repository, and service.



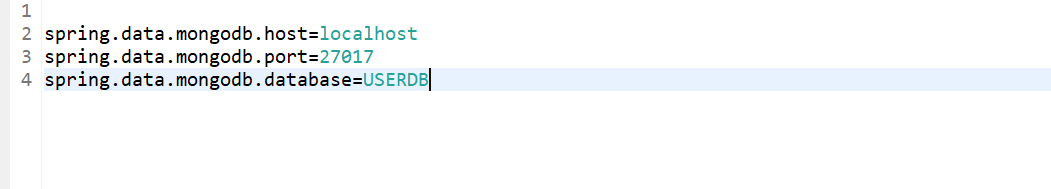






Now we are going to write a test case for our **UserService,** and we are going to mock the data of our **UserRepository.** Which means no need to hit the database so, instead of getting the actual data from Database we will just bypass the data mean we will just mock the data and we will just return it back to the service.

Already we know how to write a test case for your controller using mockito, but the approach is bit different there we have used MockMVC, MockMVC is also part of a Mockito Framework.

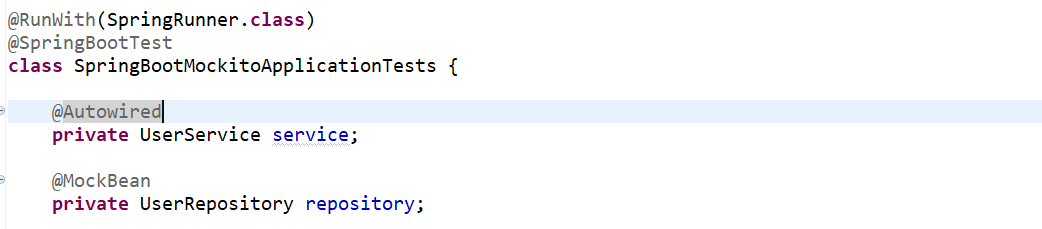


<https://www.youtube.com/watch?v=fN8VAdYfJB8>

Let’s write a Test-Case now…

Let’s go to test package and let’s annotate class with

* @**RunWith**(SpringRunner.**class**)



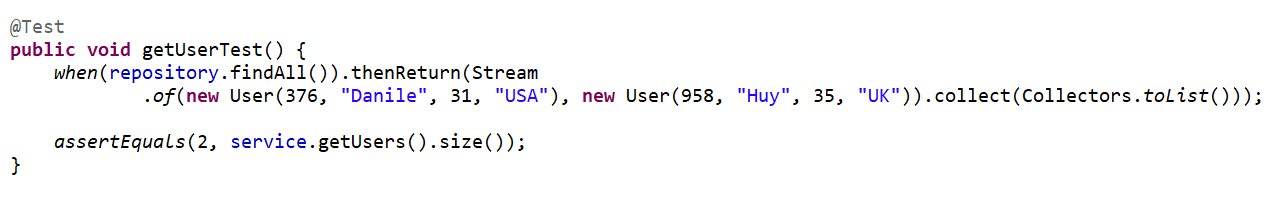
We want to mock the repository so let’s annotate with @**MockBean.** we want to inject the service here bcz we want to write the test case for Service.

Now let’s write the test case for UserService class one by one.so there are 4 methods in Service class

* **addUser()**
* **getUsers()**
* **getUserbyAddress()**
* **deleteUser()**

so, let me write the test case for **getUsers**() so you will get to know exactly what mockito is doing.

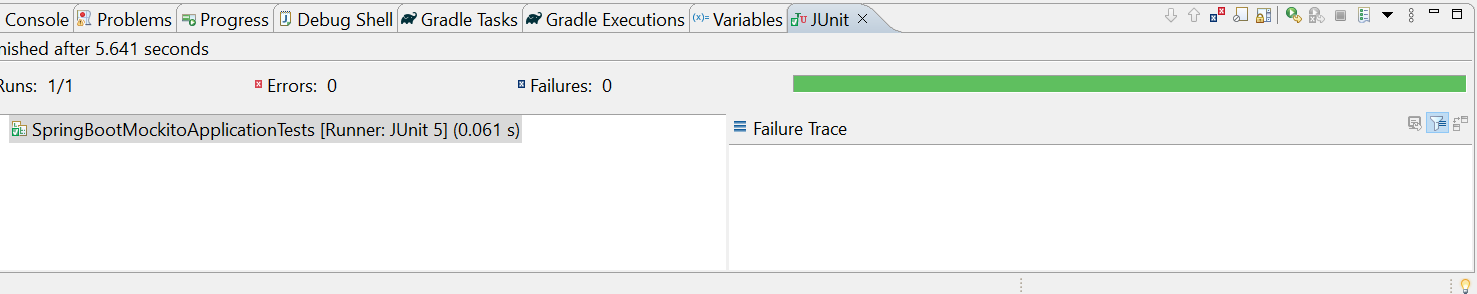
So, there is a pre-defined static method in mockito **when** I am going to call repository.findAll() **then** return the value which I am going to mock here. so, it will return the list of records.



What does this line mean when I am going to hit this method then return the list of objects instead of hitting the database. that’s why we mock here the repository. now we are comparing with the **assertEqual.**

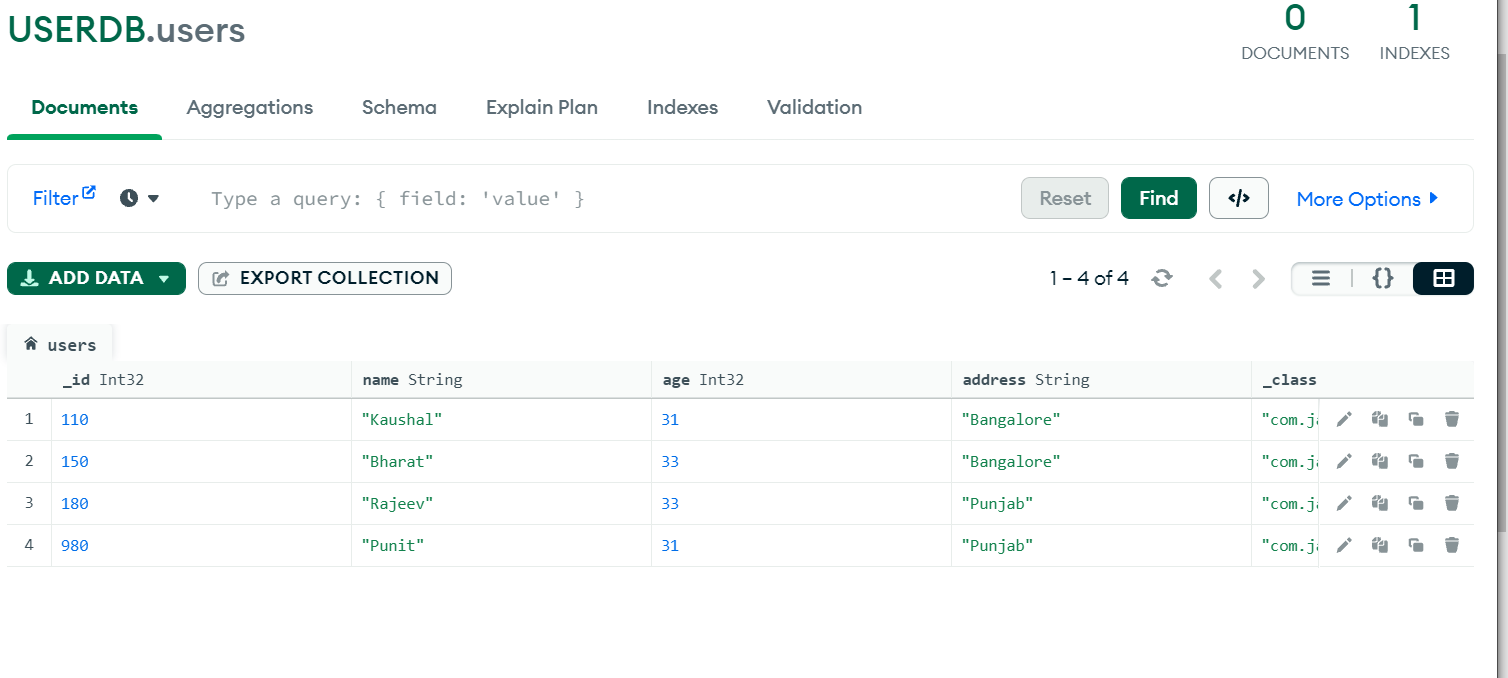
So, here **findAll**() always give the List of records. So, what we can compare here is just compare with the size. Actual is our **service.getUsers()** so, it will return the list of users and we just want the size of it.

So this method is going to call the repository and then what it will return we just mock here and then we are comparing this is my expected and this is actually giving by invoking the service.

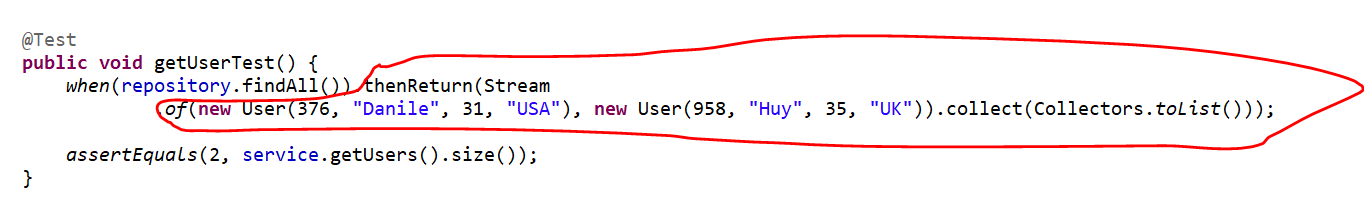


So, our test case getting successfully executed.

As we can see in our actual Database, we have total of 4 records.

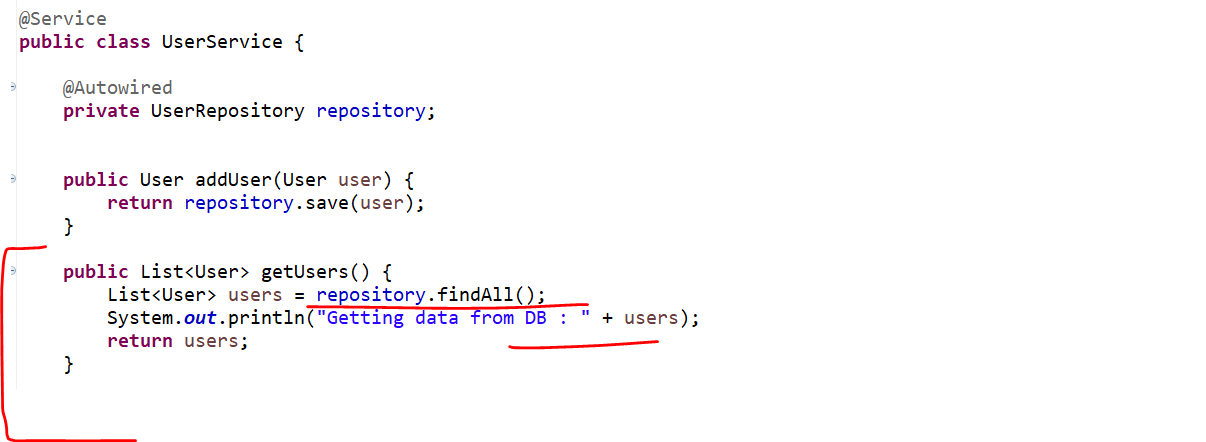


And here I just hard-code the dummy data or mock the data..

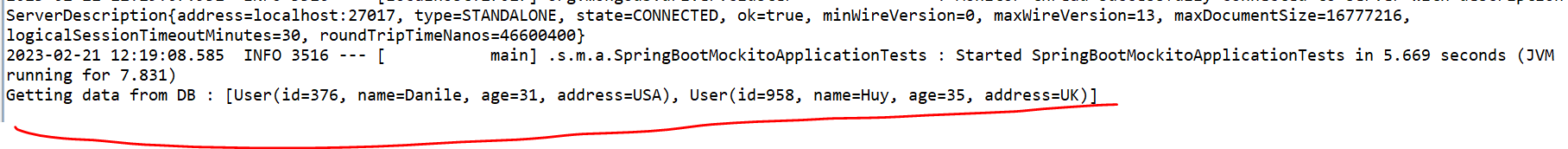


So, here I just want to confirm that actually it is hitting our database or not.

So, now go to the Service class…in service it will call the repository.findAll(), which means it should give 4 records which is available in my DB. So now let’s printing using sysout and see either its giving actually data from DB or its giving mock data which we mentioned in our test..

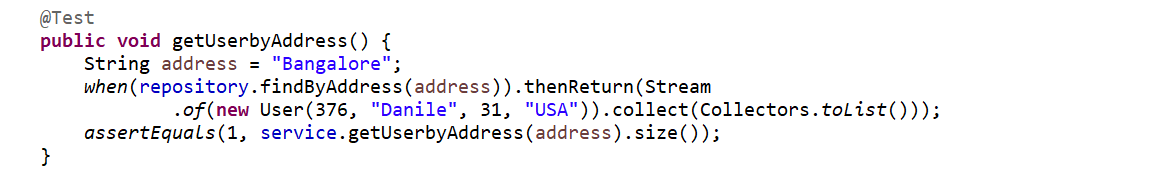


Now let’s run our test class.

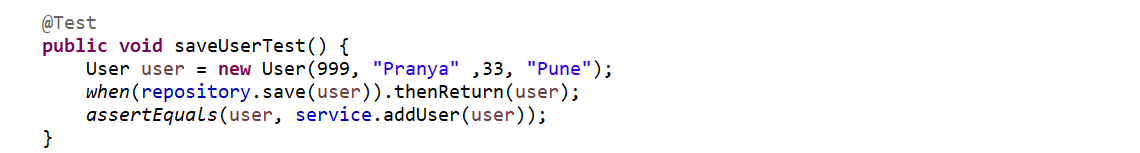


So, we can see the data which we mock here it is giving us back. Which means its not hitting our database that’s what we mock our repository.

Now let’s quickly write the test cases for rest of the methods which is in Service class…



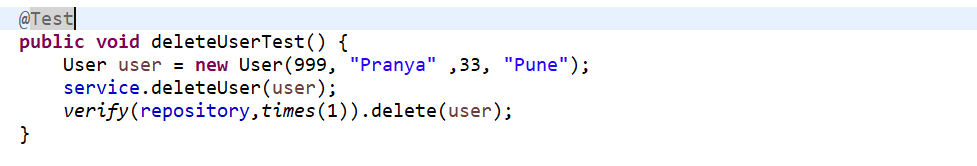
Similarly, write test case for the save method… now let’s create a user object when we call the **repository.save()** method we can see in our Service class the return type is entity.



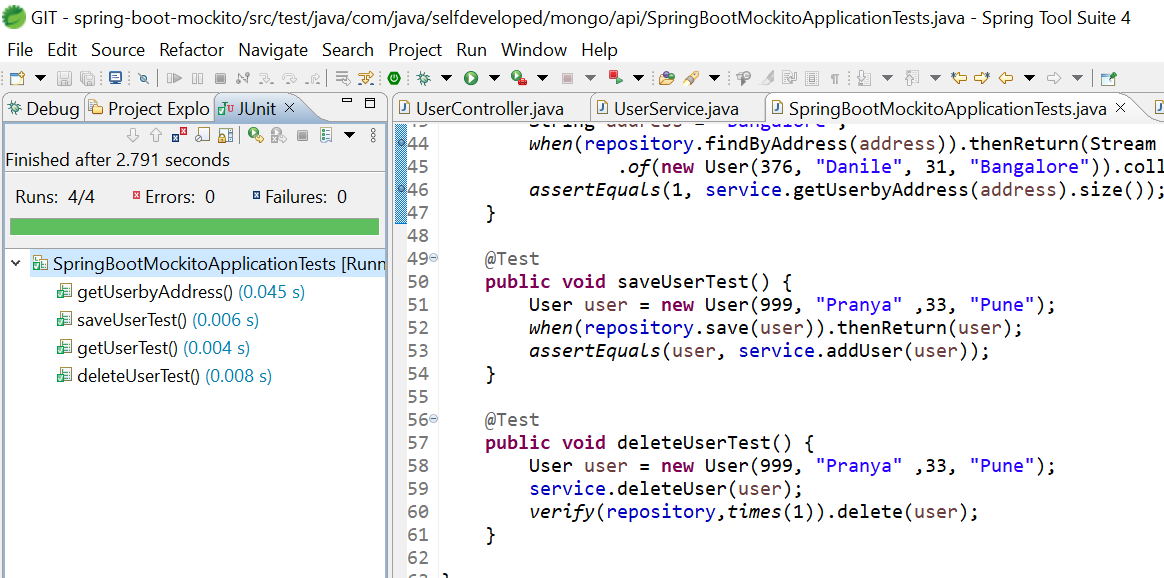
Now let’s write it for delete method. So, we can see In our service class its not returning anything mean the return type is void. So, let’s see how we can write a test case for void method using a mockito. So, let’s create a user object which we want to delete. So, the same user which we just add same we are going to delete for testing purpose. So here just call the **service.deleteUser(**user**)**.

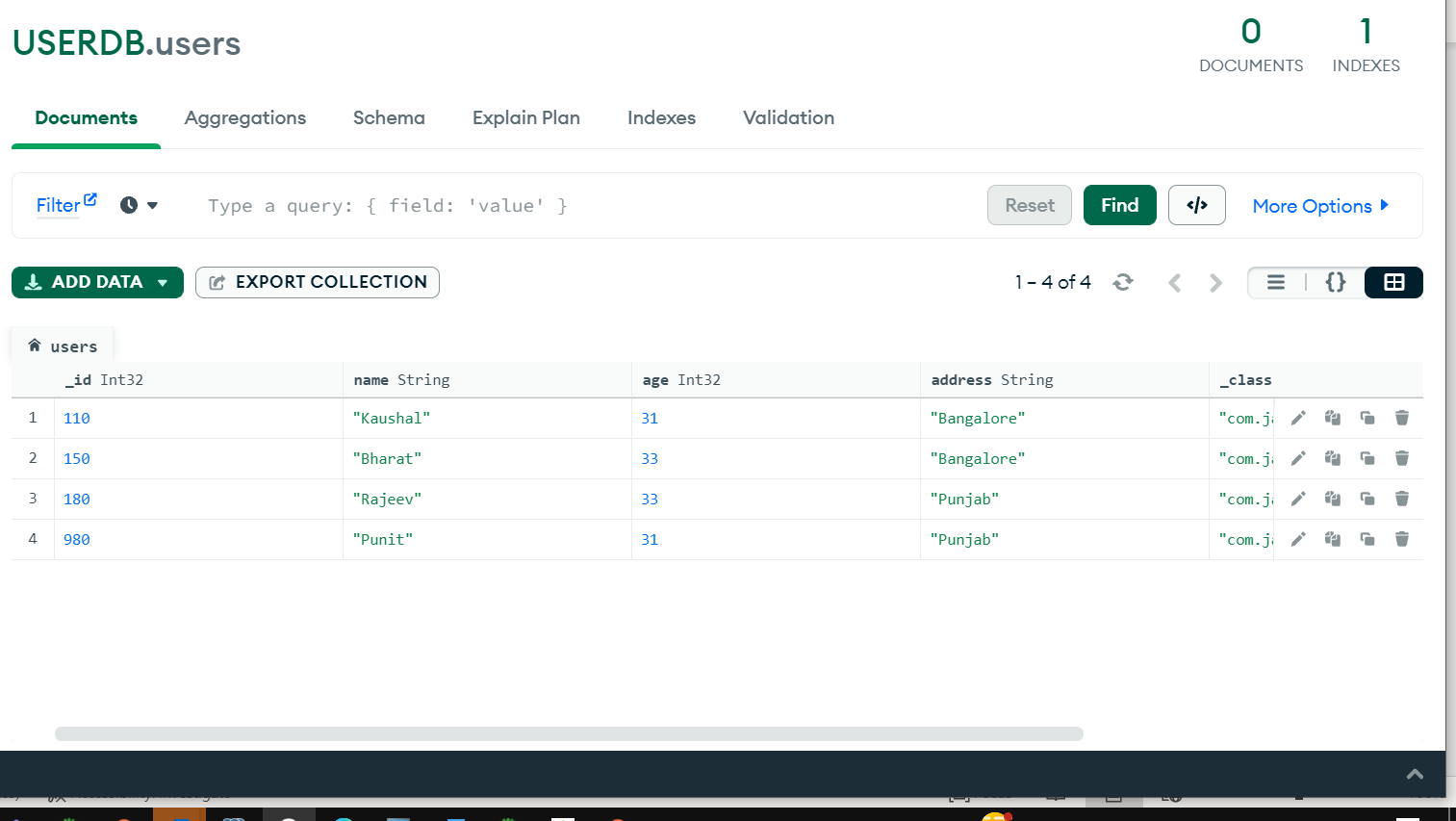
If you want to verify Either our method is called or not for Return type void.

Now as the method return type is void, we can’t expect any return type and we can’t compare with the **assert** so what we can do we can verify our method is called or not.so, verify the mocking data so our mock is our repository and let’s verify either it called for 1 time or not there is one method called **times()** you can specify your count here for testing I am giving 1 then the method which you want to check whether it called or not that is our delete method from repository.



Now let’s test for rest 3 methods…





So, we can see the data is not inserted that’s why the mockito is very useful. So that our Test application will not interact with our database instead of going to the db getting the actual data we are just mocking the data here. bcz we are inserting the mock data.

Now in next we will see how we can write a test case for private, static and final method using power mock.