Agenda.

- → What to test?
- → Best practices for Unit Testing
- → MOCKING.
  - ↳ Types of Double.

⇒ Unit Testing

↳ Testing the code in isolation.

⇒ What to Test?

- → Edge cases. ⇒ Corner cases.
  (Test cases which are not very easy to get).
- → Negative TC
  - ↳ Test cases for which our code will generate wrong out.
- → Positive TC.

Rain water Trapping

- Positive TC => Normal Array
- Negative TC
- Edge Cases
  - Empty Array
  - If all buildings are of same height
  - -ve

=> Properties for Unit test Cases.

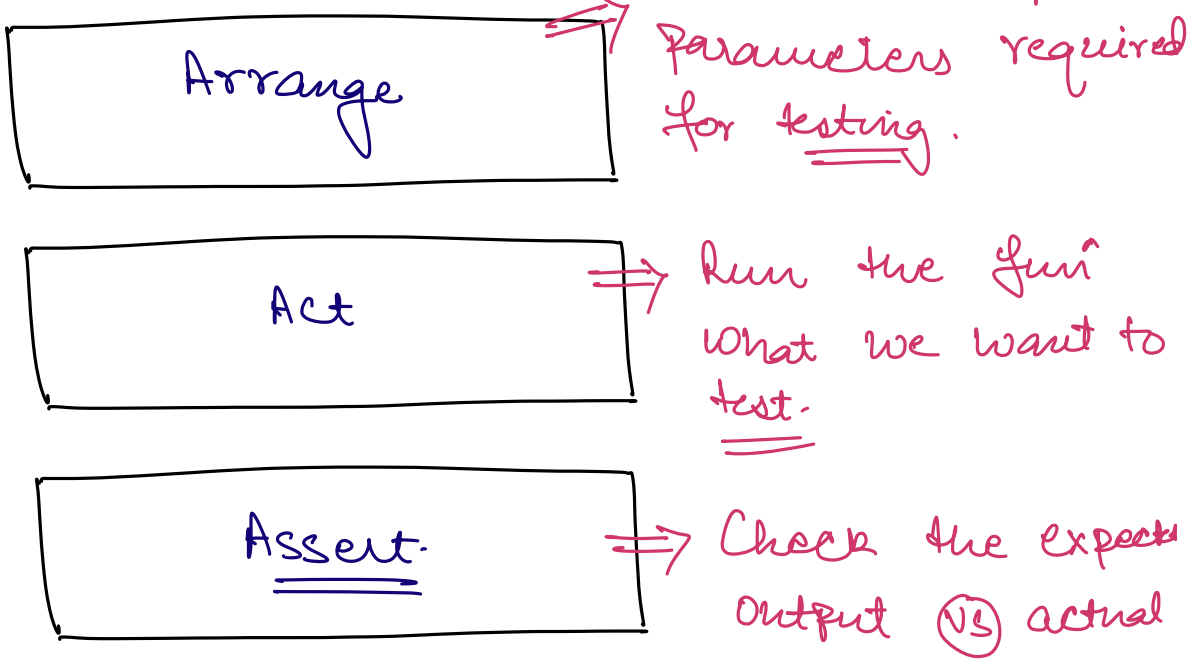
1) fast

a()
{
b()
{
}
c()
{
}
}

Mocking these dependencies.

2) 3 A's framework.

test() {



3

3 C's.
- C: Create
- C: Call
- C: Check

```
testAddition() {
    int a = 10;
    int b = 5;
    Calculator c = new Calculator()      } Arranged
    int x = c.add(a,5);                  } Act
    if (x != 15) // fail
```

3

↓

Hard Coded.

⇒ Always we should hard code the expected output, as we don't want expected output to change.

3. Isolation.

⇒ Success or failure of a TC shouldn't depend on the output of a dependency.

⇒ Ideally all the dependencies should be hard Coded.

↓

Mocking

4. Repeatable.

⇒ Our Test Cases should return the same output for same input.

whenever you are running the code

⇒ TC's shouldn't be flaky.

test getProductById() { // ProductController.

int id = 10;

in Unit test we donot call the actual database.

Calling db can be flaky, if connection with db is not established

```
when(productService.getProductById(10))
    .thenReturn(_____);
```

==

you should test the functionality of product controller individually without connecting with 3rd party Api

if there is productcontroller class, it should have corresponding test class productControllerTest.java

ProductController ⇒ ProductControllerTest.java

ProductService ⇒ ProductServiceTest.java.

```
ProductController {
    ProductService ps;

    Product getProductById(long id) {
        Product p = ps.getProductById(id);
        return new Product();
    }
}
```

==

Here instead of returning P product it is returning new Product

```
ProductControllerTest {

    @MockBean
    ProductService productService;

    @Autowired
    ProductController pc;

    testgetProductById(

        int id = 1;

        // Hard Code (mock) the output from PS.
        Product p = new Product();
        p.setId(1)
        p.setTitle(Macbook)

        when(productService.getProductById(1))
            .thenReturn(p);

        if(assertEquals(pc.getProductById(1), p)){

            ✓ PASS

        }

        ✗ FAIL.
    }

}
```

Double of ProductService.

you should also create object for controller. But this object of controller should be not be mocked. you should call actual controller method since this is a test case for Product Controller.

So Product Controller object should be actual, rest all other methods could be mocked.

this is the test service for ProductController

this is the test case for ProductService

you can hard code the value of id =1

here you can return anything

this will return P object coz you have hard coded Product Service and you are returning P

Here instead of returning P product ProductController is returning new Product

5. Self Checking

Every unit test case should be self checking

↳ Self sufficient.

⇒ To run any TC, NO human intervention should be required.

6. Test behaviours, Not the implementation.

As a developer you check the expected and actual value. if you want to check sort() function. To sort() you give unsorted array and it return sorted array -> so you are checking the behaviors.

⇒ Because implementation can change over the time.

But if you are trying to check the implementation someone can change the implementation of sort method, every test case will start failing. implementation means what sort method logic is suppose merge sort or insertion sort whether it is sorting step by step or not

⇒ We should only check if the expected output is same as actual output returned by the fun.

In Github actions, if you are trying to submit a code or push or mergel a code request--> you have triggered some actions to happen --> they will automatically run a test cases file. If all cases are passing then only you will raise a pull request else not.

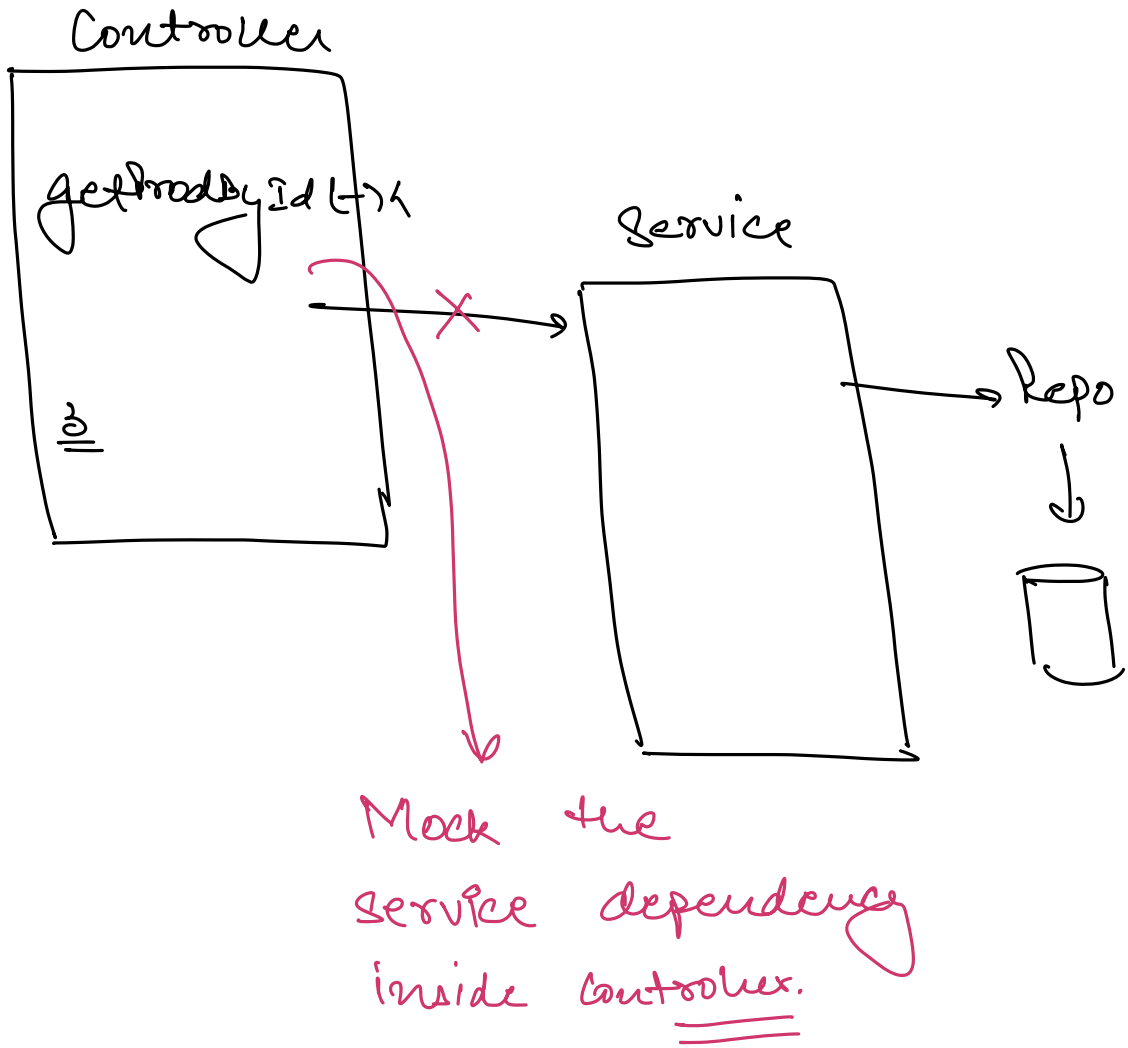
It means in any of the test cases if you need any data from the user, will server be able to run the test case automatically? If you are expecting any input from the user, will server be able to run the test case automatically? No. Ideally all of your test cases should be hard coded.

Hard coded means you should have fixed input values, fixed output values and check the funcationality you are returning the data equal to the expected data. --> No human intervention is required

Implementation driven testing -- when we test based on implementation.

MOCKING.

- Isolation — we do mocking to achieve isolation -> test each method independent of other
- Repeatable — And to make our code repeatable -> to get same output value when same input is shared even you run the code thousand times



=> In Unit test cases to test the functionality in isolation, we mock the external dependencies.

```
when(productService.getAllProducts())
    .thenReturn(new ArrayList<>());
```

when we need output from productservice.getallproducts methods return new List

⇒ When we need the output from getAllProducts API of ProductService, then instead of calling the actual method return the hard coded value.

⇒ because of hard coding the dependencies the success/failure of our fun will be dependent on them.

function

⇒ To achieve Mocking, we use test doubles.

body doubles is just like in movie in which for doing stunt a looklike person do the stunts for hero but it seems that hero is doing all the stunts.

Here in our scenario test doubles is an object or a sample object that is going to replace the actual object or also called as mocked object, copy object

- Sample object that is going to replace the actual object.
- Mocked object

⇒ Inside ProductControllerTest, instead of using the actual object ProductService we'll use the mocked object = Double of ProductService.

will be called as

⇒ Type of Doubles.

- MOCK
- STUB
- FAKE

Mocking is an action whereas Mock is a type of double. These two are different.

MOCK.

⇒ A double where we just handle the return value.

```
when(productService.getAllProducts())
    .thenReturn(new ArrayList<>());
```

This is a Mock Double

→ MOCK double.

Always return a same value.

⇒ Here dynamism isn't possible.

Java
JUnit - Most popular unit testing framework for Java.

Tool: JUnit 4 / JUnit 5 (Jupiter)
Mockito - For mocking dependencies in unit tests.
Spring Boot Test - Integrated with JUnit/Mockito for testing Spring Boot applications.

Example: @SpringBootTest

Scenario:

1) Create 5 products
2) get the count of products ⇒ (5)
3) Create 1 product
4) get the count of products ⇒ (6)

In Mock double

⇒ when(productService.getCount()).thenReturn(5) ×

Here we are hard coding the value, it will always return value 5. Dynamism is not possible.

Solution is create a Stub class

② STUB.

↳ A class that we create to replicate the behaviours of original class.

can also be called as Proxy, Duplicate

Class ProductServiceStub implements ProductService {

```
int count = 0;

void createProduct() {
    count++;
}

int getCount() {
    return count;
}
}
```

Stub is more closer to actual class behavior coz it is creating a product 5 times -> pss.createProduct. this not exactly what a real class would but somewhat similar.

In Mock, we are just incrementing the count

If you want to test the scenarios, create 5 product, get the count of products you donot create a db for that. Then how do you test it.

You create a Stub, double class name will have Stub keyword here.

So you will create a int count =0, and have a method void createProduct and do count++.

So what you will test here, if you call this method 5 times, it will return value of count as 5

ProductService pss; ← Inject ProductServiceStub.

testCase() {

In actual test case, we will create object of stub here of ProductService

```
pss.createProduct()
pss.createProduct()
pss.createProduct()
pss.createProduct()
pss.createProduct()
if (pss.getCount() != 5) {
```

here we are calling the function 5 times and if count != 5 throw an exception

throw an Exception()

3

```
pss.createProduct()
if (pss.getCount() != 6) {
```

throw an Exception()

3

✓

stub class used only for unit test

3

<< ProductService >>

FakeStore ProductService

ProductService Stub.

FAKE.

↳ More Closer to realistic implementation.

Testing

Class

ProductService is an interface

```
Class ProductServiceFake implements ProductService {
    HashMap<Integer, Product> map = ____;
    int id = 0;
```

Creating db using Hashmap

```
    createProduct(){
        Product p = new Product();
        p.setId(++id)
        ____
        ____
        ____
        map.put(id, p);
    }

    getCount(){
        return id
    }
}
```

Mock < Stub < Fake

Reality ↑

Hard Coding ↑

— * —