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.

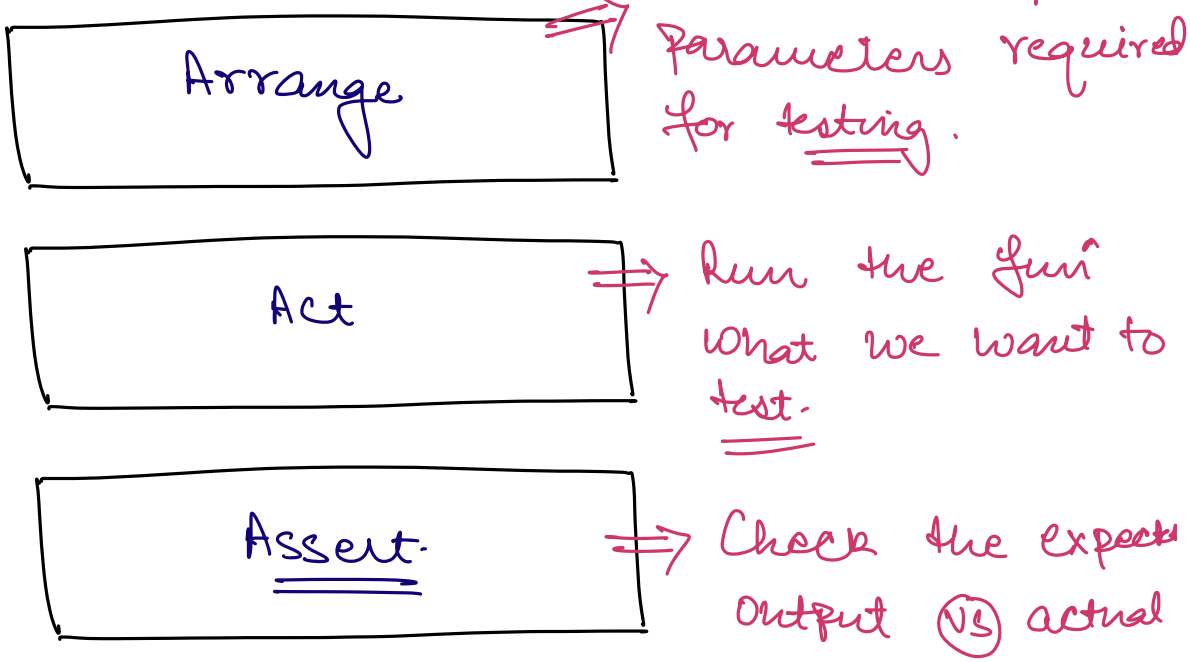
1) fast

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.

⇒ TC's shouldn't be flaky.

test getProductById() { // ProductController.

int id = 10;

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

}

ProductController ⇒ ProductControllerTest.java

ProductService ⇒ ProductServiceTest.java.

```
ProductController {
    ProductService ps;

    Product getProductById(long id) {
        Product p = ps.getProductById(id);

        return 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)){
```

Double of ↓ ProductService.

✓ PASS

```
        }
```

✗ FAIL.

```
}
```

5. Self Checking

↳ Self sufficient.

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

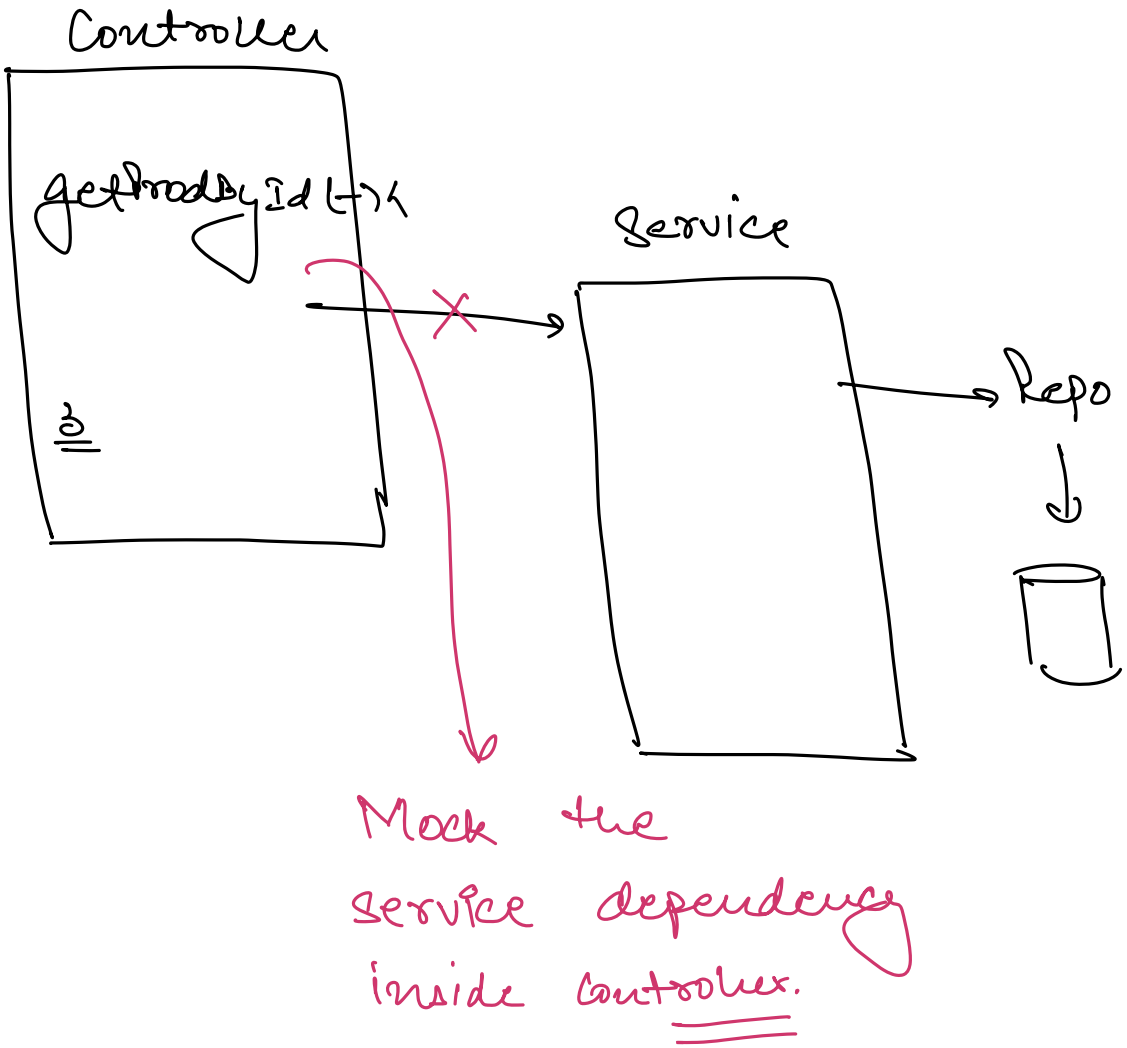
6. Test behaviours, Not the implementation.

⇒ Because implementation can change over the time.

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

MOCKING.

- Isolation
- Repeatable.



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

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

⇒ 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.

⇒ To achieve Mocking, we use test doubles.

- 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.

⇒ Type of Doubles.

- MOCK
- STUB
- FAKE

MOCK.

⇒ A double where we just handle the return value.

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

→ MOCK double.

Always return a same value

⇒ Here dynamism isn't possible.

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) } ×

(2) STUB.

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

```
Class ProductServiceStub implements ProductService{

    int count = 0;

    void createProduct(){
        count++;
    }

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

ProductService pss; ← Inject ProductServiceStub.

testCase() {

pss.createProduct()
pss.createProduct()
pss.createProduct()
pss.createProduct()
pss.createProduct()
if (pss.getCount() != 5) {
throw an Exception()
}

pss.createProduct()
if (pss.getCount() != 6) {
throw an Exception()
}

✓

<<ProductService>>

FakeStore ProductService

ProductService Stub.

FAKE.

↳ More closer to realistic implementation.

↓ Testing

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

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

═══

Mock < Stub < Fake

——————→ Reality ↑

Hard Coding ↑ ←——————

——— * ———