BINAR

Unit Test Part 2

Silver - Chapter 3 - Topic 3

BINAR

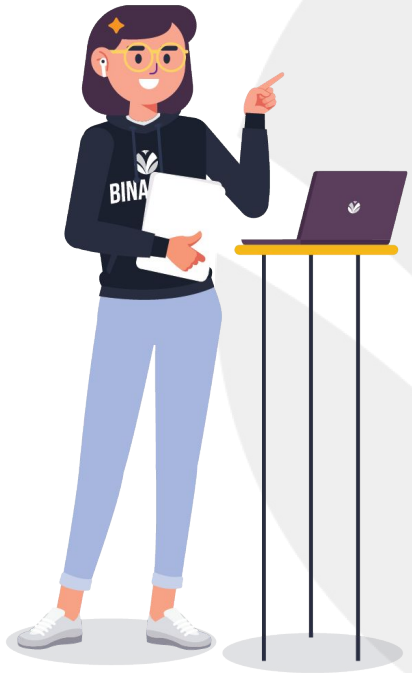
Selamat datang di Chapter 3 Topic 3 online course Back End Java dari Binar Academy!

Hello, Everyone! 😀

Gimana nih masih inget kan sama konsep Unit Testing? Seperti yang udah dijanjiin sebelumnya, menuju pada topik tiga kita bakal mengelaborasi tentang **Unit Test Part 2.**

Kita akan kupas tuntas mulai dari konsep mock, spy, stub, cara implementasi, bahkan sampai cara mengintegrasikan test pada unit test.

Yuk, langsung aja kita kepoin~



BINAR

Dari sesi ini, kita bakal bahas hal-hal berikut:

- Konsep dan implementasi Mock, Spy dan Stub
- Cara integration test pada unit testing

Bukan cuma gambar aja yang bisa di rekayasa, tapi method juga bisa, lho!

Dalam istilah Java, proses merekayasa ini dinamakan sebagai **Mock.**

Tapi, merekayasa seperti apa sih yang dimaksud di Java ini? Yuk, kita simak slide selanjutnya~

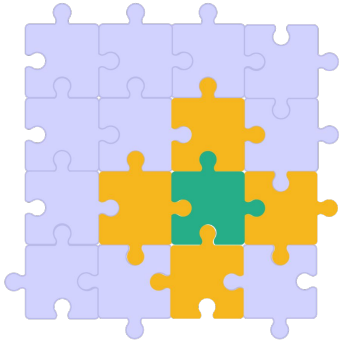
BINAR

“Sebenarnya Mock itu apa, ya?”

Dalam membuat sebuah unit test, sebaiknya dilakukan dengan menghilangkan dependency terhadap sistem lain.

Walaupun demikian, bisa aja nih kita masih menemukan adanya dependency di dalam sebuah class atau method yang mau kita test.

Real System



Green = Class in focus

Yellow = Dependencies

Grey = Other unrelated classes

Class in Unit Test

Green = Class in focus

Yellow = Mocks for the unit test

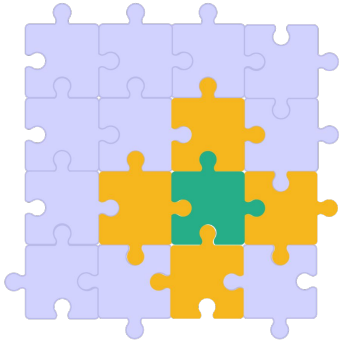
Unit Test (Part 2)

Kalau kasusnya kita menemukan adanya dependency di dalam sebuah class atau method yang mau kita test, kita bisa pakai cara rekayasa.

Yaitu, merekayasa method dari dependency lain. Sehingga seolah-olah sistem kita udah melakukan request dan menerima response.

Proses rekayasa ini disebut juga dengan melakukan mocking.

Real System



Green = Class in focus

Yellow = Dependencies

Grey = Other unrelated classes

Class in Unit Test

Green = Class in focus

Yellow = Mocks for the unit test

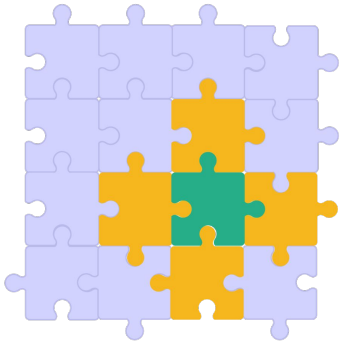
BINAR

Kalau pakai Mock, kita bisa menentukan sendiri response yang diharapkan dari method, lho!

Sebuah mock merupakan **object tiruan yang dibuat untuk memenuhi behaviour dari business logic dari class yang di-test**.

Walaupun dapat menentukan response sendiri, penggunaan Mock tetap harus berdasarkan asumsi dan/atau kondisi yang diinginkan dalam unit test tersebut.

Real System



Green = Class in focus

Yellow = Dependencies

Grey = Other unrelated classes

Class in Unit Test

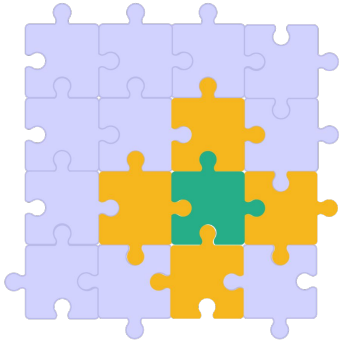
Green = Class in focus

Yellow = Mocks for the unit test

Unit Test (Part 2)

Selain itu, kita juga perlu fokus melakukan verifikasi behavior terhadap satu class aja tanpa harus memikirkan cara kerja class lain.

Real System



Green = Class in focus

Yellow = Dependencies

Grey = Other unrelated classes

Class in Unit Test

Green = Class in focus

Yellow = Mocks for the unit test

“Emang gimana sih cara melakukan Mocking?”

Untuk melakukan mocking, kita bisa pakai **framework Mockito yang paling populer di Java**.

Oh iya, Mockito juga bisa digunakan dalam bahasa pemrograman turunan kayak Kotlin, serta bisa diintegrasikan dengan JUnit.

Dokumentasi Mockito bisa dilihat di sini, ya.

Unit Test (Part 2)

Untuk menambahkan Mockito, ada dua cara. **Pertama, kita cukup menambahkan Spring Boot Starter Test yang terbaru**. Yaitu yang udah disediakan oleh Spring Initializr.

Kedua, kita juga bisa pakai JUnit 5 dan Mockito tanpa pakai Spring Boot Starter Test. Hal ini supaya bisa leluasa mengatur versinya, terlepas dari modul test dari Spring Boot.

Sekarang, silakan kamu pilih mau pakai yang mana ya, bestie~

```
<dependency>
        <groupId>org.junit.jupiter</groupId>
        <artifactId>junit-jupiter-engine</artifactId>
        <version>5.2.0</version>
        <scope>test</scope>
    </dependency>
    <dependency>
        <groupId>org.mockito</groupId>
        <artifactId>mockito-junit-jupiter</artifactId>
        <version>2.19.0</version>
        <scope>test</scope>
    </dependency>
```

BINAR

Di bawah adalah contoh penggunaan Mockito untuk melakukan Mocking~

Lebih detailnya, contoh dibawah ini menggunakan sebuah repository yang jadi dependency dari sebuah service. Cekidot~

```
public interface PersonRepo extends JpaRepository<Person, Integer> {
    @Query(
        "SELECT CASE WHEN COUNT(s) > 0 THEN TRUE ELSE FALSE END FROM Person s WHERE s.personId = ?1")
    Boolean isPersonExitsById(Integer id);
}
```

Kalau ini adalah class entity-nya, gengs~

```java
@Entity
@Data
@NoArgsConstructor
@AllArgsConstructor

// Class
public class Person {
    @Id
    private Integer personId;
    private String personName;
    private String personCity;
}
```

BINAR

Sedangkan class service yang pakai Repo tersebut, yaitu PersonService ada di contoh kayak di bawah ini!

Coba perhatikan baik-baik, yaaa~

```java
@Service
public class PersonService {

    @Autowired private PersonRepo repo;

    public List<Person> getAllPerson()
    {
        return this.repo.findAll();
    }

    public PersonService(PersonRepo repo)
    {
        // this keyword refers to current instance
        this.repo = repo;
    }
}
```

Unit Test (Part 2)

Berikut adalah unit test buat class service-nya~

```
@ExtendWith(MockitoExtension.class)

// Main class
class PersonServiceTest {

    @Mock private PersonRepo personRepo;

    private PersonService personService;

    @BeforeEach void setUp()
    {
        this.personService
            = new PersonService(this.personRepo);
    }

    @Test void getAllPerson()
    {
        personService.getAllPerson();
        Mockito.verify(personRepo).findAll();
    }
}
```

Oke, selanjutnya adalah cara untuk melakukan test-nya, gengs!

Untuk menjalankan test-nya, kita pakai runner annotation yang ditulis di awal test suite, yaitu:

@ExtendWith(MockitoExtension.class).

DI unit test untuk class service tersebut, kita melihat dependency dari PersonService, yaitu PersonRepo yang dijadikan mock pakai annotation **@Mock**.

Sedangkan PersonService merupakan class yang di-test menggunakan object sungguhan.

```
@ExtendWith(MockitoExtension.class)

// Main class
class PersonServiceTest {

    @Mock private PersonRepo personRepo;

    private PersonService personService;

    @BeforeEach void setUp()
    {
        this.personService
            = new PersonService(this.personRepo);
    }

    @Test void getAllPerson()
    {
        personService.getAllPerson();
        Mockito.verify(personRepo).findAll();
    }
}
```

Unit Test (Part 2)

Untuk melakukan dependency injection, kita cuma perlu memasukkan dependencynya pada constructor.

Cara lain untuk mendeklarasikan sebuah mock selain pakai annotation adalah dengan menggunakan cara di samping.

```
PersonRepo repo = Mockito.mock(PersonRepo.class)
```

Unit Test (Part 2)

Dari contoh tersebut, kita bakal melakukan verifikasi apakah ada pemanggilan method findAll dilakukan pakai Mockito.verify atau enggak. Lalu melakukan pengecekkan untuk mencari tahu ada berapa kali pemanggilan pada method.

Selain itu, kita juga bisa pakai:

Mockito.verify(personRepo, times(1)).findAll();

```
@ExtendWith(MockitoExtension.class)

// Main class
class PersonServiceTest {

    @Mock private PersonRepo personRepo;

    private PersonService personService;

    @BeforeEach void setUp()
    {
        this.personService
            = new PersonService(this.personRepo);
    }

    @Test void getAllPerson()
    {
        personService.getAllPerson();
        Mockito.verify(personRepo).findAll();
    }
}
```

Untuk case yang lebih kompleks, kita bisa menambahkan Assertions dalam menilai isi method. Sedangkan untuk mengubah kondisi return dari mock object, kita bisa pakai **Mockito.when().thenReturn()** kayak contoh berikut:

```
Mockito.when(personRepo.findAll()).thenReturn(personList)
```

Setelah ada bayangan tipis-tipis tentang Mock, selanjutnya kita masuk ke materi tentang Spy.

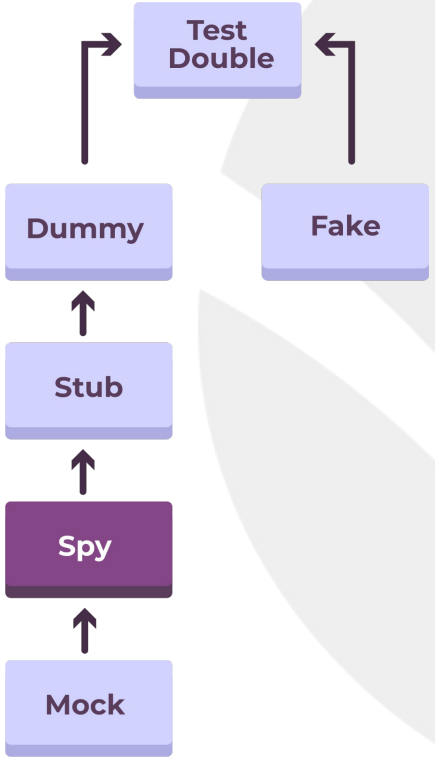
Hmm.. apakah Spy ini mirip kayak mata-mata itu?

Apa itu Spy?

Spy merupakan cara untuk melakukan partial mocking.

Bukan seperti konsep mata-mata yang ada di film, spy adalah cara kita dapat melakukan mocking di sebagian method aja.

Perbedaannya dengan mocking adalah spy digunakan pada object yang riil.

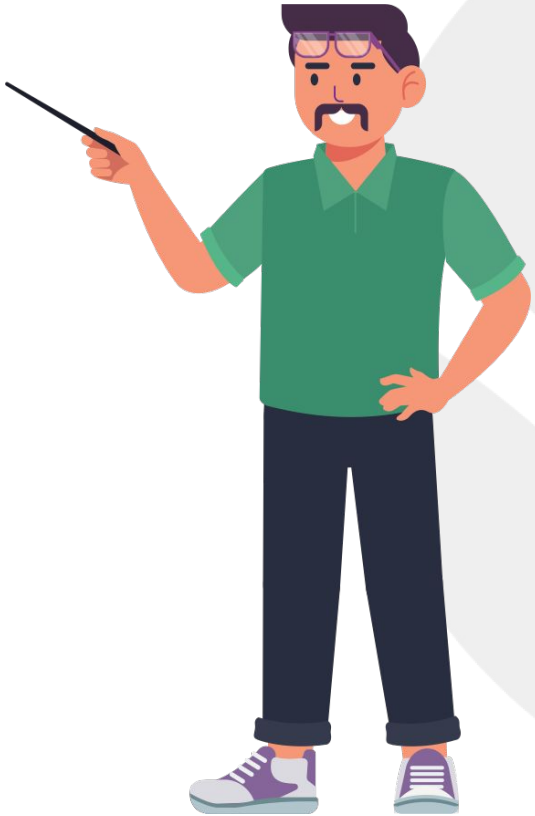


Unit Test (Part 2)

Kenapa Spy perlu dipakai untuk melakukan mocking secara sebagian?

Yaitu, supaya kita bisa menghindari sebagian method, dan mampu menguji sebagian method secara riil.

Spy juga biasanya digunakan pada class yang besar.



Spy juga bisa dilakukan tanpa menggunakan library kayak Mockito, lho!

“Yang bener? Caranya gimana?”

Caranya adalah dengan bikin class tiruan yang isinya sebagian method yang bersifat riil.

Tapi, di subtopic ini kita bakal pakai Mockito untuk melakukan spy dulu.

BINAR

Berikut contoh penggunaan Spy!

```
@Test
public void whenAddStringAndTheSizeIsCorrect_thenCorrect() {
    List<String> list = new ArrayList<String>();
    List<String> spyList = Mockito.spy(list);

    spyList.add("A");
    spyList.add("B");

    Mockito.verify(spyList).add("A");
    Mockito.verify(spyList).add("B");

    assertEquals(2, spyList.size());
}
```

Pada contoh spy tadi, tentunya kita pakai object real.

Method Add yang dilakukan merupakan method add kayak yang dilakukan ketika menggunakan object real.

```
@Test
public void whenAddStringAndTheSizeIsCorrect_thenCorrect() {
    List<String> list = new ArrayList<String>();
    List<String> spyList = Mockito.spy(list);

    spyList.add("A");
    spyList.add("B");

    Mockito.verify(spyList).add("A");
    Mockito.verify(spyList).add("B");

    assertEquals(2, spyList.size());
}
```

“Terus, kenapa sih kita harus pakai Spy?”

Hal yang menunjukkan penggunaan spy lebih baik dibandingkan mocking adalah **kalau pakai spy, kita bisa melakukan tracking untuk melakukan verifikasi behavior pakai object yang riil**.

Cara lain untuk mendeklarasikan suatu spy adalah pakai annotation spy.

```
@Spy
List<String> spyList = new ArrayList<String>();
```

BINAR

Tadi kita udah bahas Mock dan Spy.

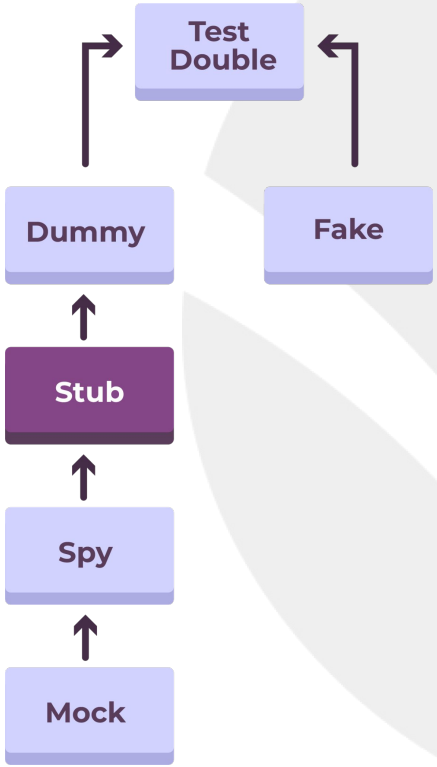
Ternyata ada yang lain yang bisa kita pakai juga buat melakukan unit test nih, namanya Stub.

“Emangnya Stub itu bermanfaat banget, ya?”

Bermanfaat dong! **Stub dipakai untuk memanipulasi data dari sebuah unit test**.

Pada library Mockito, kita bisa pakai mock sekaligus stub secara bersamaan, lho.

Kalau mocking fokus pada verifikasi apakah method tersebut dipanggil atau nggak. Kalau di stub, kita fokus pada manipulasi datanya, sob.



Untuk mengubah kondisi data return dari mock object, kita bisa pakai **Mockito.when().thenReturn()**

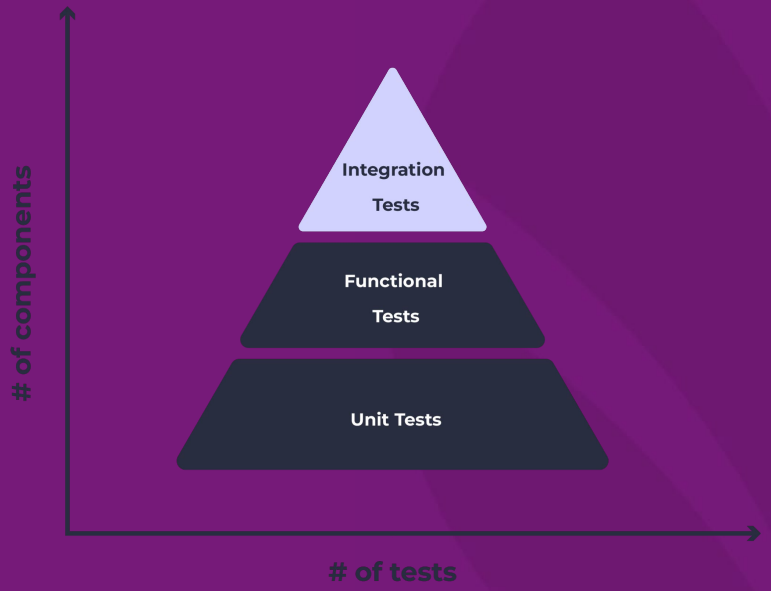
Contohnya bisa langsung kamu cek di bawah, ya!

```
Mockito.when(personRepo.findAll()).thenReturn(personList)
```

Okey! Terakhir, kita bakal bahas tentang **Integration Test.**

Test ini masih menjadi bagian dari unit testing.

Biar ada bayangan, langsung aja deh kita geser slide-nya!



BINAR

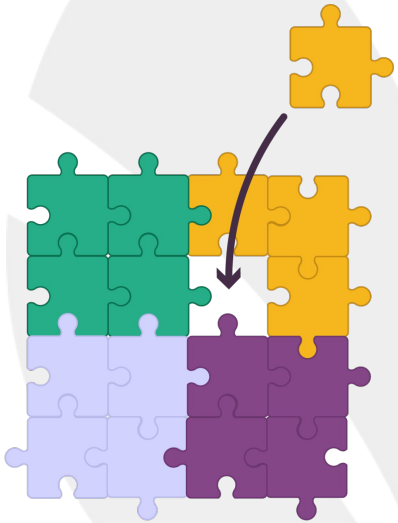
“Emangnya Integration Test itu apa, sih?”

Pasti kamu sempat berpikir kalau itu adalah cara dalam mengetes functionality modul pas aplikasi dinyalakan, kan?

Sebenarnya nih, definisi dari integration testing di sini bukan kayak gitu.

Integration testing ini masih berupa unit testing yang dipakai dalam mengetes penggunaan class kalau udah diintegrasikan dengan dependency yang dimiliki.

Unit Test



Integration Test

Unit Test (Part 2)

Dalam mengimplementasikan integration testing ini biasanya aplikasi yang dijalankan harus melakukan dependency injection secara riil.

Harus dipastikan dulu dependency injection udah dilakukan dengan benar.

Untuk melakukan dependency injection ketika test, kita bisa pakai annotation **@TestConfiguration** buat bikin Bean dari class yang bakal di test.

Berikut contohnya~

```
:
@TestConfiguration
public class RouterTestCofiguration {

    @Bean
    public HttpClient httpClient() {
        return HttpClient.create()
                .keepAlive(true)
                .compress(true)
                .responseTimeout(Duration.ofSeconds(120));
    }

    @Bean
    public DefaultRouterAll defaultRouterAll(HttpClient httpClient){
        return new DefaultRouterAll(httpClient);
    }
}
```

Atau kita juga bisa bikin autowire secara langsung di class yang mau di test, lho~

Contoh yang bakal dipakai kali ini adalah integration testing yang dipakai untuk mengetes service yang pakai repository.

Dari sini kita bakal menguji apakan repository atau query yang digunakan udah benar atau belum.

Unit Test (Part 2)

Untuk mengetes repo yang berhubungan dengan database, maka database yang dipakai untuk testing juga harus ada.

Database yang bakal dipakai buat scope test ini adalah H2.

H2 merupakan relational database yang embedded atau datanya disimpan ke dalam memori program Java.

Database ini biasanya dipakai untuk keperluan testing aja.

Selain itu, untuk menambahkan H2 di database, kita bisa menambahkan library-nya di pom.xml.

```
<dependency>
    <groupId>com.h2database</groupId>
    <artifactId>h2</artifactId>
    <scope>test</scope>
</dependency>
```

Kita bakal pakai contoh kayak yang ada di contoh mock, yaaa.

Berikut adalah class dari repository dan class entity~

```
public interface PersonRepo extends JpaRepository<Person, Integer> {
    @Query(
        "SELECT CASE WHEN COUNT(s) > 0 THEN TRUE ELSE FALSE END FROM Person s WHERE s.personId = ?1")
    Boolean isPersonExitsById(Integer id);
}
```

Yang ini adalah class entity-nya, bestie~

Perhatikan baik-baik, ya!

```java
@Entity
@Data
@NoArgsConstructor
@AllArgsConstructor

// Class
public class Person {
    @Id
    private Integer personId;
    private String personName;
    private String personCity;
}
```

Unit Test (Part 2)

Selanjutnya, yang di bawah ini adalah contoh integration testing dari class service yang dipakai~

```java
package com.demo.repo;

import com.demo.entites.Person;
import org.junit.jupiter.api.Test;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.context.SpringBootTest;

import static org.assertj.core.api.AssertionsForClassTypes.assertThat;

@SpringBootTest
class PersonRepoTest {

    @Autowired
    private PersonRepo personRepo;

    @Test
    void isPersonExitsById() {
        Person person = new Person(1001, "Amiya", "Bhubaneswar");
        personRepo.save(person);
        Boolean actualResult = personRepo.isPersonExitsById(1001);
        assertThat(actualResult).isTrue();
    }
}
```

BINAR

“Terus cara menjalankan test-nya gimana, dong?”

Untuk menjalankan test-nya, kita bisa pakai annotation **@SpringBootTest** di test suite-nya.

Assertions dilakukan untuk mengecek output dari function.

Kalau mau mengetes repository, kita harus meng-inject reponya dulu ya, sob.

```java
package com.demo.repo;

import com.demo.entites.Person;
import org.junit.jupiter.api.Test;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.context.SpringBootTest;

import static org.assertj.core.api.AssertionsForClassTypes.assertThat;

@SpringBootTest
class PersonRepoTest {

    @Autowired
    private PersonRepo personRepo;

    @Test
    void isPersonExitsById() {
        Person person = new Person(1001, "Amiya", "Bhubaneswar");
        personRepo.save(person);
        Boolean actualResult = personRepo.isPersonExitsById(1001);
        assertThat(actualResult).isTrue();
    }
}
```

Kalau kita mau melakukan tes class service yang pakai repo, kita bisa pakai cara ini juga, yaitu dengan **meng-autowire class yang akan di test**.

Supaya class integration ini bisa terisolasi dari class lain tanpa harus memikirkan gimana dependency injection yang terjadi pada class lain, kita bisa pakai annotation di samping~

```
@SpringBootTest(classes = {BaseRouter.class})
@ExtendWith(SpringExtension.class)
@Import(RouterTestCofiguration.class)
```

Latihan lagi, latihan terus~

Eits, pembahasan kita tentang unit test part 2 belum selesai. Silakan kamu **implementasikan Mock pada project Java yang kamu buat di pertemuan sebelumnya**!

Latihan ini dilakukan di kelas dan silahkan diskusikan hasil jawaban kamu dengan teman sekelas dan fasilitator.

Selamat mencoba, yaa~

Coba deh kamu renungin. Kalo dipikir-pikir Mock, Spy dan Stub kayak pemeran pengganti (stunts) dalam film nggak sih? Apa yang mereka lakukan adalah sebagai pengganti dari function yang membutuhkan dependency yang ada pada code kita.

Yuk kita rinci lagi, apa sih manfaat dari penggunaan Mock, Spy dan Stub ini bagi developer?

Waah, selesai deh pembahasan kita di Chapter 3 Topic 3 ini.

Pada topik berikutnya, kita bakal bahas tentang Java 8 Part 1 alias ngobrolin Java dengan segala fitur-fitur baru yang super canggih.

Penasaran kayak gimana? Yuk langsung ke topik selanjutnya~