

Unit Testing Part 1

Silver - Chapter 3 - Topic 2

**Selamat datang di Chapter 3 Topic 2
online course Back End Java dari
Binar Academy!**



Hai, Binarian!

Masih inget nggak pada topik sebelumnya kita belajar apa? buat kalian yang menjawab Error dan Exceptions, atau Try-catch blocks, selamat ya kalian emezing~

Move on pada topik ini kita bakal membahas tentang **Unit Testing**. As we expected, isinya bakal berkelana seputar cara mengetes program supaya tahu apa udah berjalan baik atau belum.

Kalau gitu, yuk langsung aja kita kepoin~



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

- Definisi Unit Testing
- Pengantar Junit
- Melakukan Unit Testing pada Java
- Konsep Assertions
- Penggunaan BeforeEach dan AfterEach
- Konsep TDD



Ibarat lagi masak, pasti kamu suka
icip-icip buat ngetes rasanya, kan?

Mirip kayak masakan, aplikasi juga perlu
dites dulu. Bedanya, pake metode khusus
yang namanya **Unit Testing**.

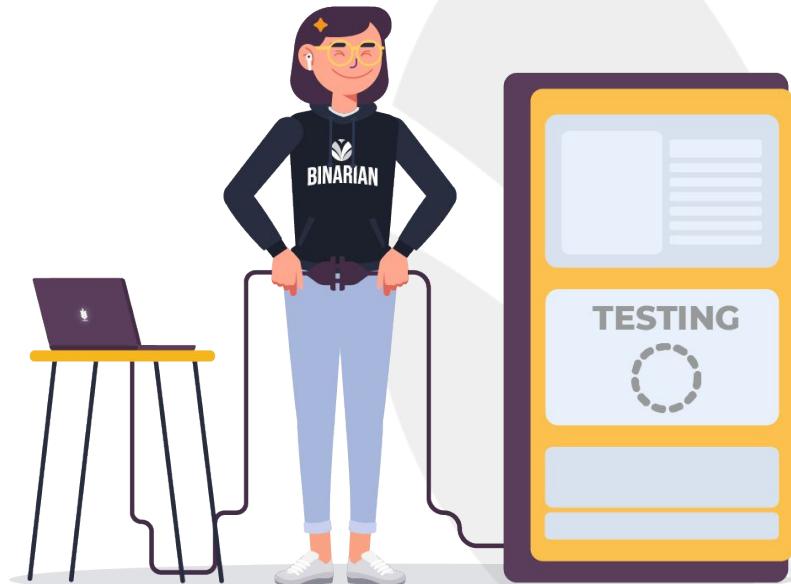


Testing itu apa, ya?

Setiap developer yang melakukan development untuk suatu program, pasti akan melakukan pengetesan pada program yang dibuat juga.

Pengetesan ini punya tujuan penting.

Yaitu, tujuannya **untuk mengetahui apakah program berjalan dengan baik atau tidak**.



Yang pasti nih, untuk melakukan pengetesan tersebut perlu effort yang besar, bahkan ketika baru mempersiapkan testing tersebut.

Soalnya gini, misalnya kalau pas program dites, ternyata muncul dependency atau library tambah di luar JDK terhadap sistem lain. Kalau gitu rupa-rupanya bakal cukup wakwaw kan ya~



Belum lagi pengetesan juga harus mencakup banyak banget kasus-kasus.

Termasuk juga untuk kasus yang emang jarang terjadi dan susah banget untuk direplikasi.

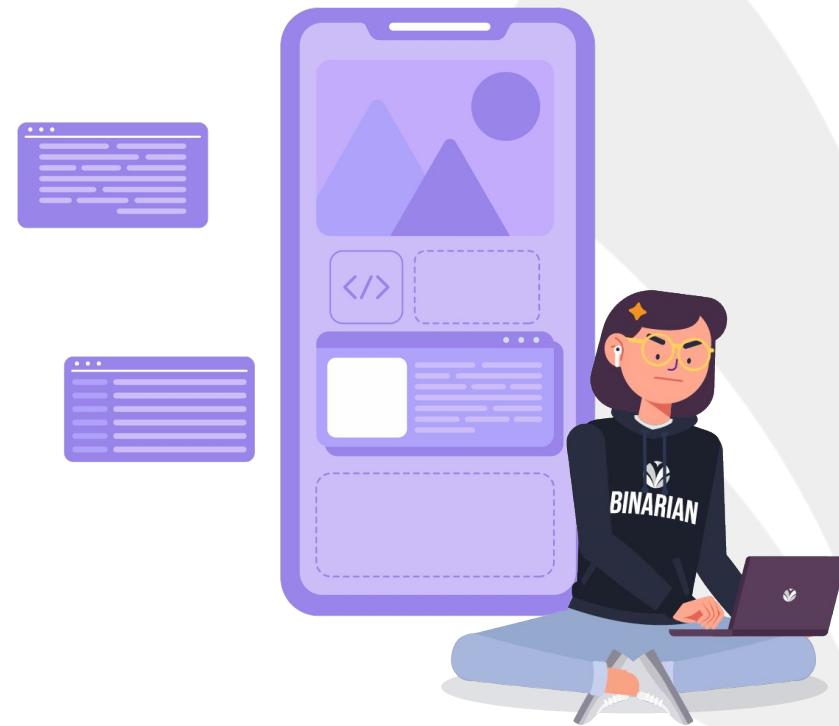
Berhubung semuanya pun harus tetap dilakukan pengetesan, tentunya pengetesan ini bakal makan waktu yang lama banget, gengs.



Testing yang dilakukan dengan cara tadi disebut sebagai end-to-end testing~

“Gimana tuh, maksudnya?”

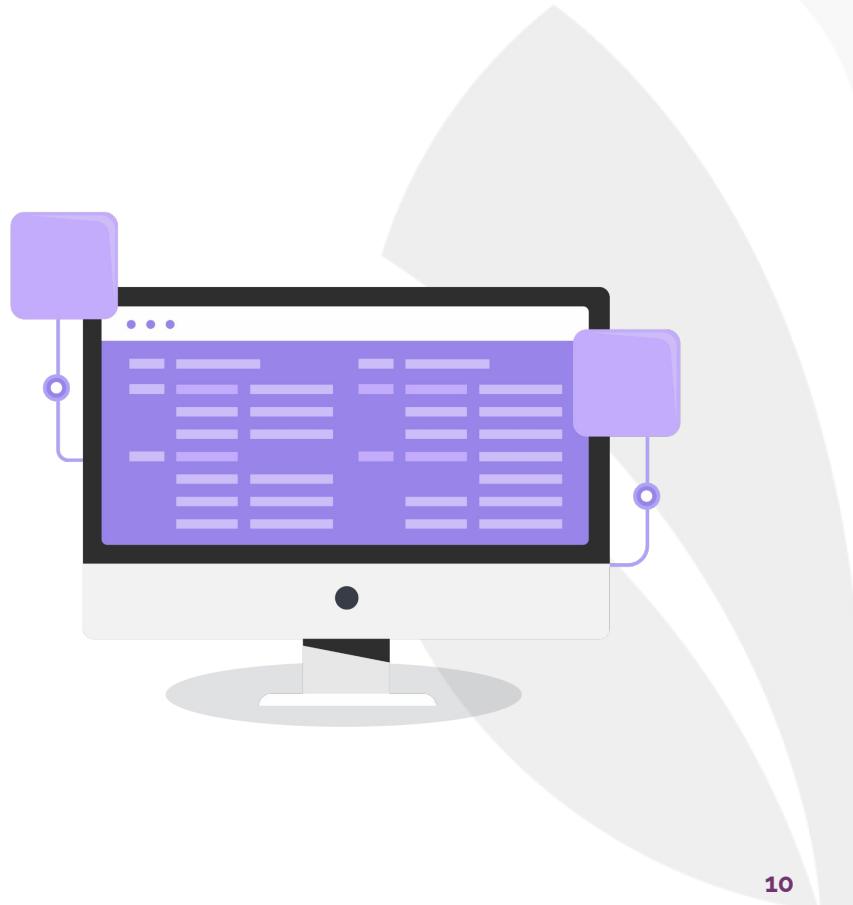
Jadi, End-to-end testing adalah test yang biasanya mereplikasi kasus-kasus yang terjadi saat program atau aplikasi berjalan di environment production (pas udah bisa diakses secara public).



Ada end-to-end testing, ada juga unit testing~

Beda sama end-to-end testing, **unit testing merupakan pengetesan di dalam level code.**

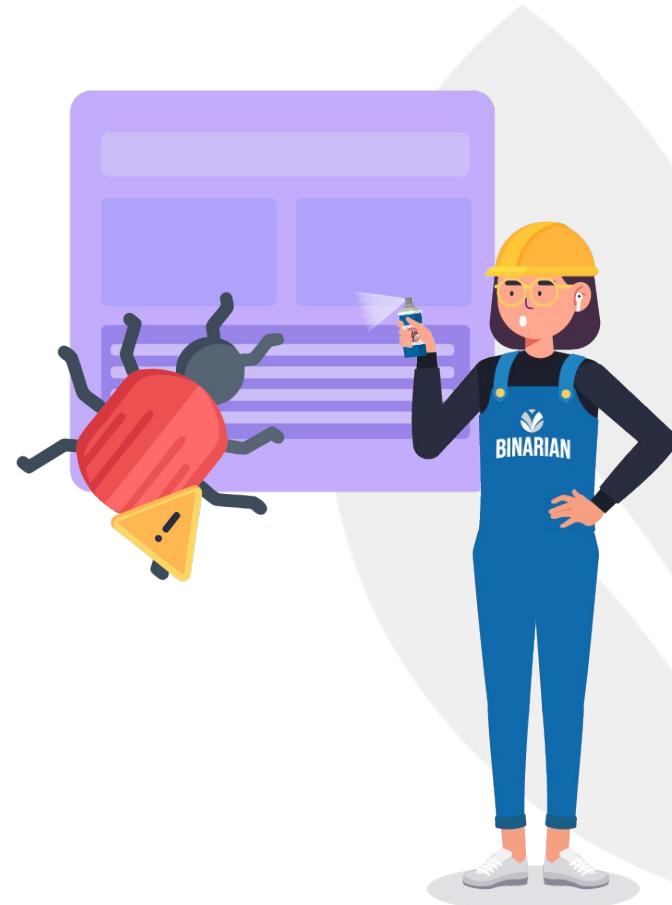
Kalau pake unit testing, developer bisa melakukan pengetesan tanpa membutuhkan dependency dari system lain, class lain, bahkan method lain.



**Bukan cuma sekedar pengetesan,
unit testing juga bisa mendeteksi
bugs, lho~**

True!! Selain pengetesan, **unit testing bisa bantu developer untuk mendeteksi bugs lebih awal dibandingkan dengan end to end testing.**

Hal ini karena pengetesan udah bisa dilakukan ketika fase development atau saat melakukan code. Dari situ, developer bisa dengan cepat melakukan fixing atau perbaikan.



Terus, cara kerja unit testing itu kayak gimana?

Good question, **unit testing ini berjalan seperti sebuah program yang mengetes suatu program sampai di level method.** Oleh karena itu unit testing bisa dieksekusi dengan cepat.

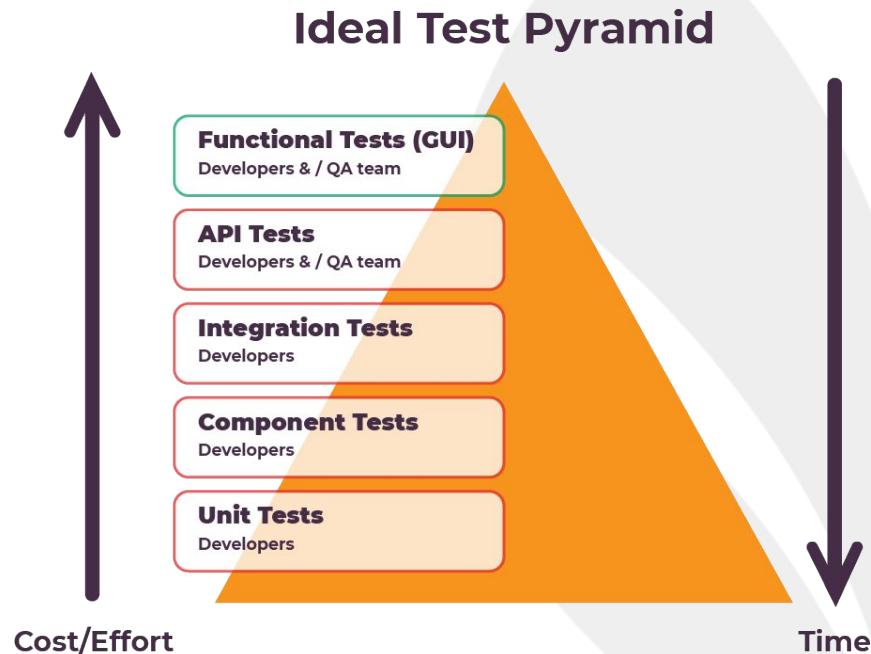
Kita bisa melakukan beribu-ribu test case bahkan melakukan simulasi untuk edge cases yang susah banget dilakukan saat end-to-end testing.



Tingkatan Testing

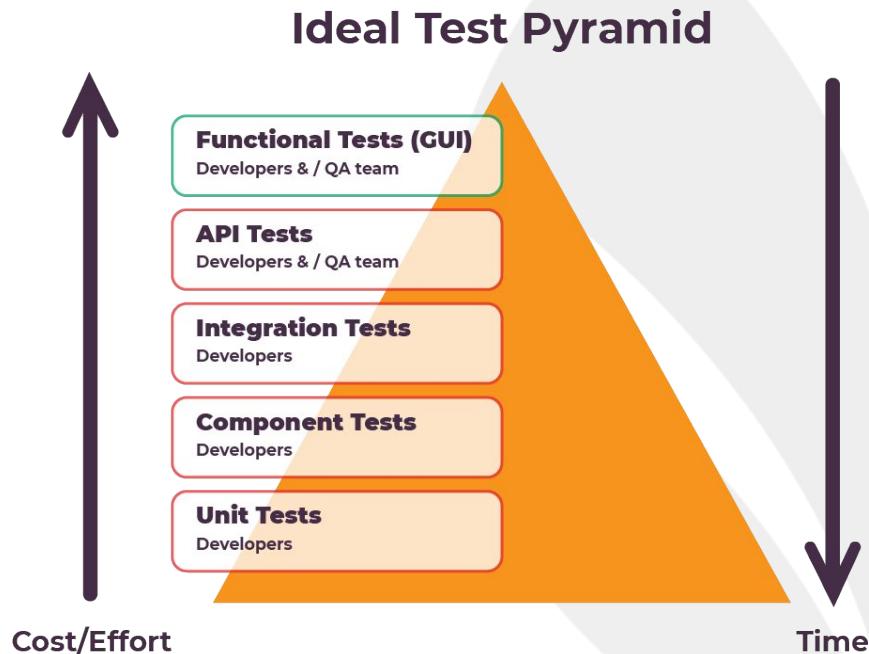
Kalau kalian intip gambar di samping, testing ada yang bisa dilakukan oleh beberapa pihak seperti developer dan Quality Assurance (QA Team). Dibagi menjadi:

- **End to End Testing**
- **Integration Testing**
- **Unit Testing**



Mungkin kalian nanya, **kan udah ada QA Engineer, kenapa kita harus test juga?**

Soalnya hal itu berkaitan sama tingkatan-tingkatan testing yang bakal kita bahas detail pada slide berikutnya.



End-to-End Testing

Pertama nih, inti dari jenis ini adalah **kita ngetest sebagai user di dalam aplikasi kita**. Jadi kita nggak boleh ngomongin konteks teknis lagi disini.

Yang perlu kita lakukan adalah mendefinisikan secara spesifik parameter testingnya.

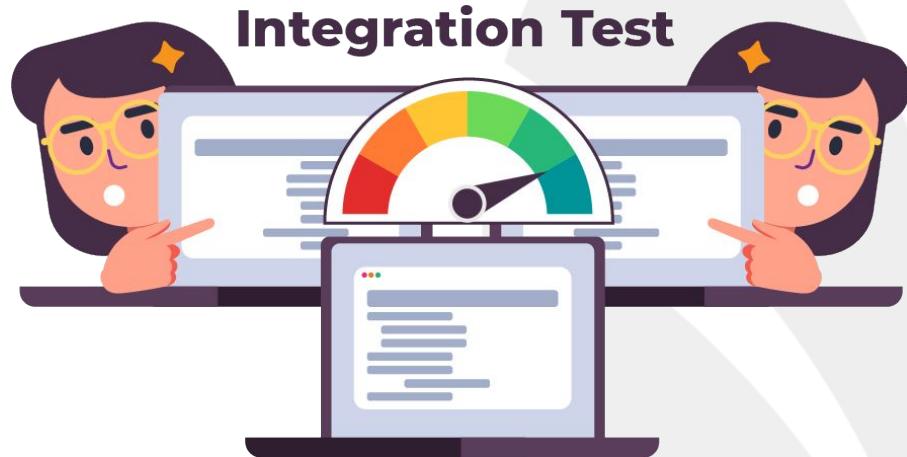
Sebagai contoh, **jika aku membuka halaman Home, maka aku akan melihat teks "Hello World!"**.



Integration Test

Inti integration test adalah **kita melakukan test integrasi antar unit.**

Kalau konteksnya back-end, kita bakal test bagaimana back-end nge-handle request pada endpoint X.



Integration Test

Unit Test



Unit Test

Unit test adalah **metode testing yang hanya berfokus pada satu unit aja dalam 1 test case**.

Di dalam kode, apa yang kita test di dalam unit testing adalah class, dan function tanpa dependensi ke hal lain.

Nah, dikaitin sama pembagian team testing tadi, kita sebagai **developer itu biasanya cuma nulis test sampe unit testing dan integration testing aja.**

Sedangkan end to end testing itu biasanya masuk scope QA Engineer.



Unit testing itu murah, lho!

Iya, jadi karena sistemnya yang cepat dan efisien, unit testing cuma butuh resource yang murah. Nggak kayak end-to-end testing yang bisa menjadi costly.

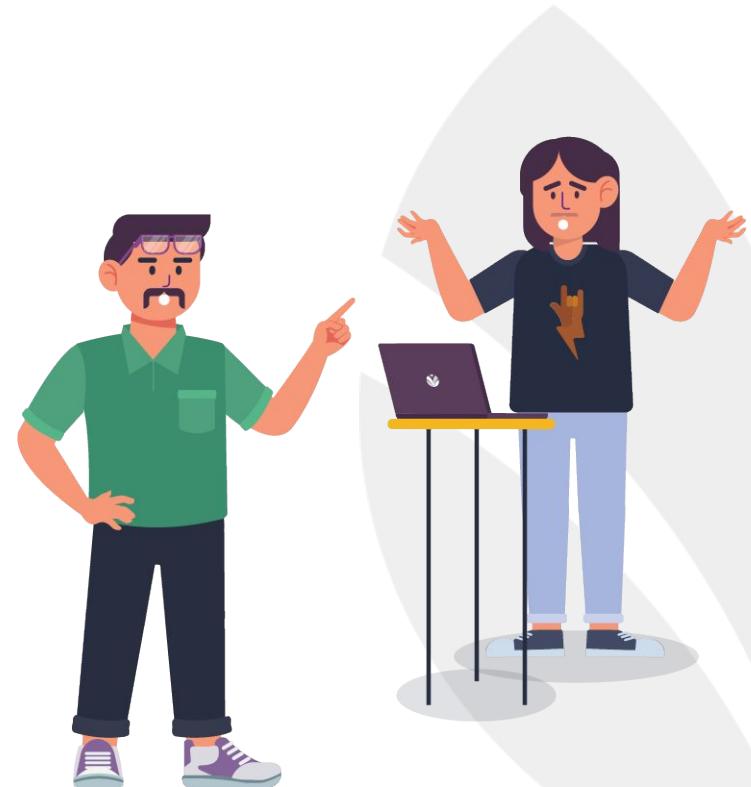
Nggak nyangka, kan?



Biar makin paham kenapa unit testing itu penting, kita pakai contoh ya!

Gini.. pada suatu hari, Jimi Jamet sedang menambahkan fitur di aplikasi yang ia kembangkan.

Terus Mas Gun lewat sambil tanya-tanya tentang proses pembuatannya. Sayangnya, pas ditanya gimana cara buat ngetes aplikasinya, ternyata Jimi Jamet nggak melakukan unit testing 😞



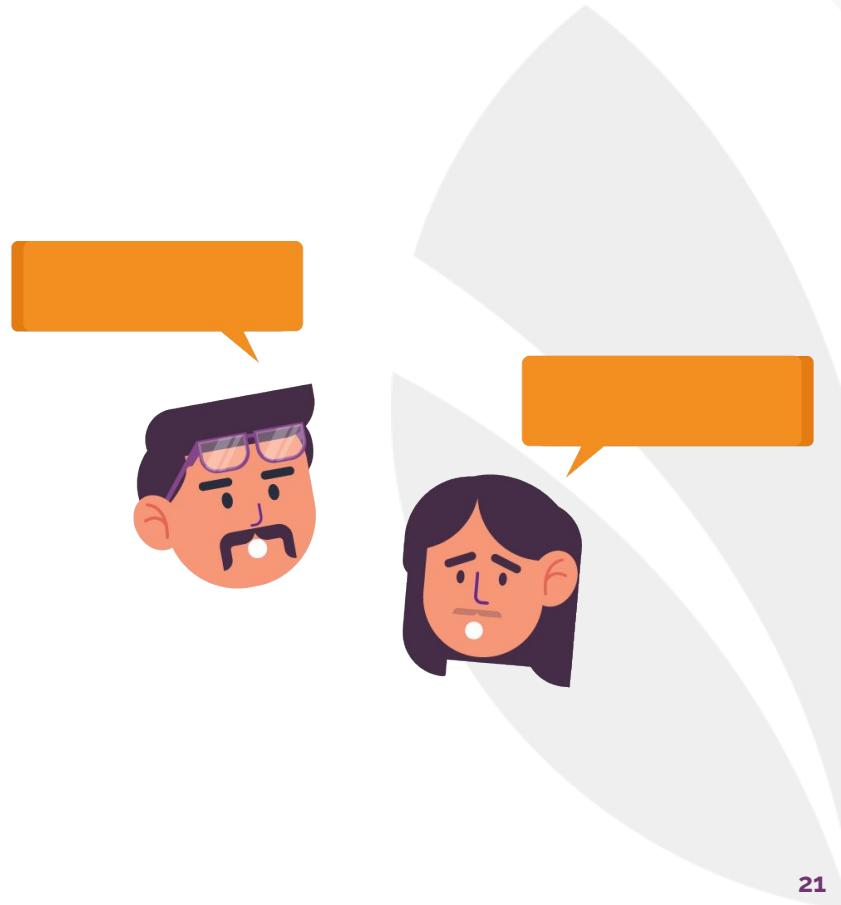
Percakapan antara Jimi Jamet dan Mas Gun kayak gini, nih~

Mas Gun: "Gimana nih gan kabar penggerjaan aplikasinya? Udah ditest belum?"

Jimi Jamet: "Udah dong, udah aku test manual".

Mas Gun: "Terus udah tambahin unit testnya buat tau coverage-nya? udah sepenuhnya dicover belum?"

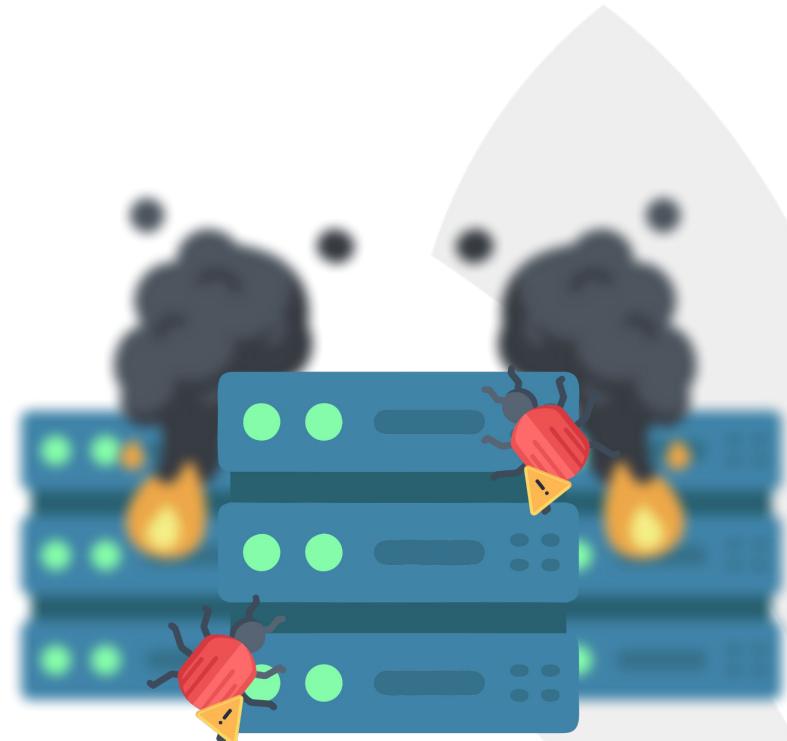
Jimi Jamet: "Buat apa? Lagian udah aku test dengan manual lewat API, udah jalan dengan benar kok aplikasinya"



Terus, dengan percaya diri Jimi Jamet malah melakukan deploy ke production di aplikasinya. Dan boooomm! Banyak kesalahan data terjadi dan bug ada di mana-mana.

Jadi, yaaa terpaksa harus rollback ke versi sebelumnya deh~

Tapi nih, kalau misalnya Jimi Jamet melakukan unit testing, meskipun nantinya terjadi kesalahan, seharusnya udah bisa ditanggulangi dari awal, lho~



Sebelum pakai testing, kita juga perlu tahu beberapa keyword terkait dengan testing, gengs~

- **Test case**, kasus untuk menguji suatu program. Misal, kasus dengan input yang beda-beda bisa menghasilkan beragam test case.
- **Bugs**, temuan yang didapatkan ketika program dijalankan tapi nggak sesuai dengan ekspektasi.
- **Test Scenario**, kumpulan dari test case.



- **Positive Test Case**, test case dengan data yang valid sehingga menghasilkan output yang sesuai.
- **Negative Test Case**, test case dengan data yang nggak valid. Biasanya dipakai untuk mengecek exception handling biar program nggak error.
- **Edge Cases**, case yang jarang banget terjadi.



Supaya lebih kebayang, coba kita lihat deh perbedaan antara Unit Testing sama End-to-End Testing pada gambar di bawah ini!

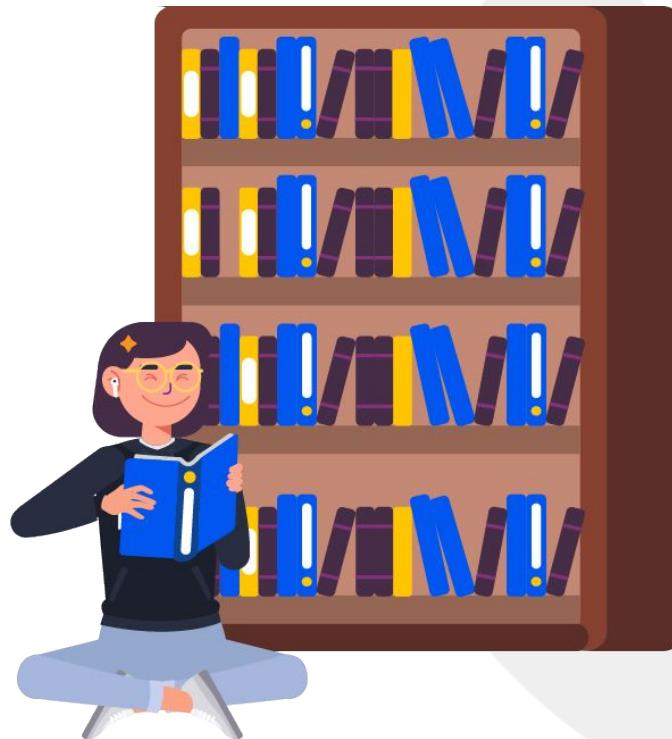


Kalau tadi udah bahas tentang konsep User Testing, sekarang kita masuk ke framework untuk melakukan Unit Testing yang namanya **Spring Test**.

Untuk bisa melakukan unit test, kita memerlukan library untuk menulis dan menjalankan test-nya.



Untuk implementasi unit test pada Java, kita membutuhkan library tambahan karena Java sendiri tidak menyediakan. Salah satunya adalah dengan bantuan annotation `@SpringBootTest`.



Project Setup

Bentar, kita coba recall dulu deh. Dalam melakukan unit testing dibutuhkan framework Spring Test. Nah, untuk menjalankan framework ini kita perlu library tambahan yang bisa dibantu dengan annotation @SpringBootTest.

Kalau kamu penasaran cara melakukan project set up annotation, kamu bisa cek gambar di samping.

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-test</artifactId>
    <scope>test</scope>
    <version>2.5.0</version>
</dependency>
<dependency>
    <groupId>com.h2database</groupId>
    <artifactId>h2</artifactId>
    <scope>test</scope>
</dependency>
```

Belum selesai, guys. Di samping ini ada lanjutan dari cara untuk set up annotation @springBootTest. Perhatikan baik-baik ya~

```
● ● ●  
@Test  
public void givenEmployees_whenGetEmployees_thenStatus200()  
throws Exception {  
  
    createTestEmployee("bob");  
  
    mvc.perform(get("/api/employees")  
        .contentType(MediaType.APPLICATION_JSON))  
        .andExpect(status().isOk())  
        .andExpect(content()  
            .contentTypeCompatibleWith(MediaType.APPLICATION_JSON))  
        .andExpect(jsonPath("$.name", is("bob")));  
}
```

Integration Testing with @SpringBootTest

Masih ngomongin annotation `@SpringBootTest`, tadi kan baru cara set up-nya tuh. Sekarang kita masuk ke integrasi `@SpringBootTest`. Ada tiga cara yang bisa kamu lakukan untuk melakukan integrasi. Gambar di samping adalah contoh yang pertama.

```
● ● ●

@RunWith(SpringRunner.class)
@SpringBootTest(
    webEnvironment = WebEnvironment.MOCK,
    classes = Application.class)
@AutoConfigureMockMvc
@TestPropertySource(
    locations = "classpath:application-integrationtest.properties")
public class EmployeeRestControllerIntegrationTest {

    @Autowired
    private MockMvc mvc;

    @Autowired
    private EmployeeRepository repository;

    // write test cases here
}
```

Kamu masih bingung integrasinya? Tenang aja~ Di samping ini ada penjabaran lebih detailnya.

```
● ● ●

@RunWith(SpringRunner.class)
public class EmployeeServiceImplIntegrationTest {

    @TestConfiguration
    static class EmployeeServiceImplTestContextConfiguration {
        @Bean
        public EmployeeService employeeService() {
            return new EmployeeService() {
                // implement methods
            };
        }
    }

    @Autowired
    private EmployeeService employeeService;
}
```

Cara kedua integrasi `@SpringBootTest`

Untuk cara kedua ada integrasi secara umum. Implementasinya akan kayak di samping ini~

```
● ● ●  
@RunWith(SpringRunner.class)  
@SpringBootTest  
public class EmployeeServiceImplIntegrationTest {  
  
    @Autowired  
    private EmployeeService employeeService;  
  
    // class code ...  
}
```

Cara ketiga integrasi `@SpringBootTest`.

Untuk langkah implementasinya, kamu bisa lihat yang ada di samping ini. Perhatikan baik-baik, ya dua kodingan di samping 😊

```
● ● ●  
@TestConfiguration  
public class EmployeeServiceImplTestContextConfiguration {  
  
    @Bean  
    public EmployeeService employeeService() {  
        return new EmployeeService() {  
            // implement methods  
        };  
    }  
}
```

```
● ● ●  
@RunWith(SpringRunner.class)  
@Import(EmployeeServiceImplTestContextConfiguration.class)  
public class EmployeeServiceImplIntegrationTest {  
  
    @Autowired  
    private EmployeeService employeeService;  
  
    // remaining class code  
}
```

Kalau tadi udah bahas tentang konsep User Testing, sekarang kita masuk ke framework untuk melakukan Unit Testing yang namanya JUnit.

JUnit ya gengs, bukan julit atau julid atau junet 😞



JUnit sebagai framework buat melakukan unit testing~

Junit merupakan framework yang disediakan oleh Java untuk membantu melakukan unit testing.

Kalau pakai Junit, **memungkinkan kita buat melakukan testing sampai level method.**

Junit yang bakal dipakai adalah Junit 5 ya! Kamu bisa akses [di sini](#) ~



JUnit adalah framework yang nggak disediakan oleh JDK, gengs~

Jadi, kalau mau pakai Junit, kita harus menambahkan dependency-nya juga karena kita pakai Maven sebagai build automation tools-nya.

Oh iya, untuk fungsinya sendiri Spring Test dan JUnit punya fungsi yang sama. Bedanya ada di cara implementasinya aja~



Berikut cara menambahkan Junit pada project yang menggunakan Maven!

```
● ● ●  
<dependency>  
  <groupId>org.junit.jupiter</groupId>  
  <artifactId>junit-jupiter-api</artifactId>  
  <version>5.8.2</version>  
  <scope>test</scope>  
</dependency>
```

Tambahkan dependency tersebut di file pom.xml pada tag <dependencies>

Karena scopenya hanya test, library dari framework ini cuma bisa dipakai di directory test aja ya!

Setelah ada bayangan tentang User Testing dan JUnit, selanjutnya kita bakal mencoba **Create a Test**.

Betul, kita coba praktik, dulu~

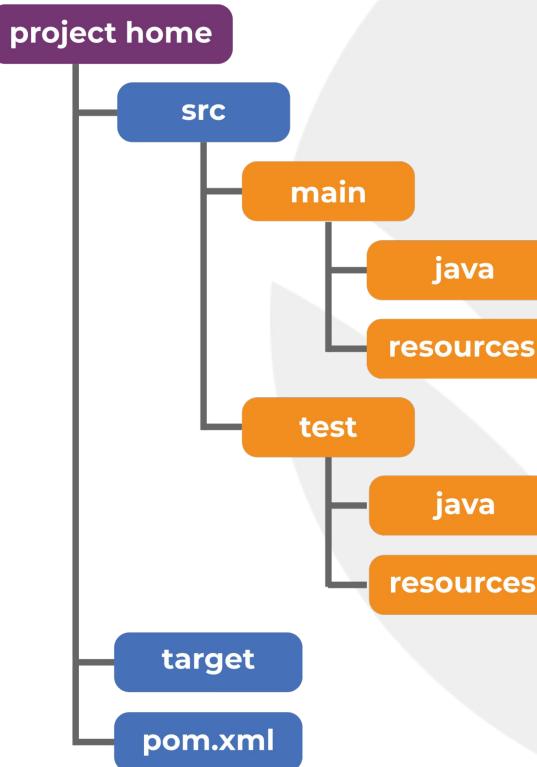


Terus, gimana dong caranya buat create a test?

Pada struktur Maven project, direktori yang dipakai untuk membuat unit test adalah **folder test**.

Folder java adalah directory dalam membuat source code unit test.

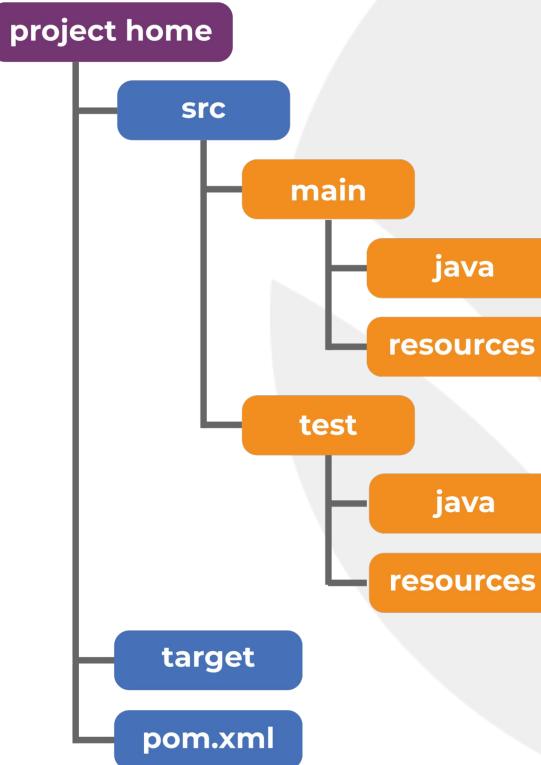
Folder resources merupakan directory untuk menyimpan resource berupa variable atau file static lain yang dibutuhkan di dalam unit test.



Biar ada bayangan, kita pakai contoh, ya~

Misalnya gini..

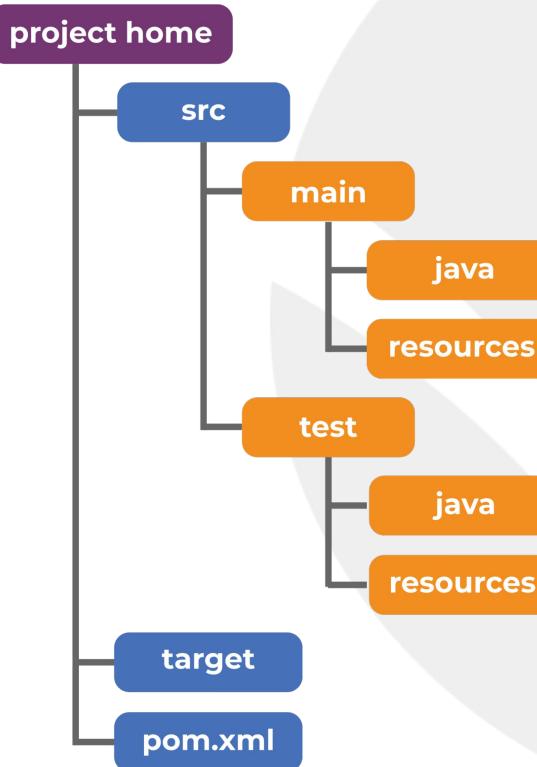
Kita mau bikin unit test dari HelloWorld.class, berarti kita sebaiknya bikin class dari unit test pake nama HelloWorldTest.class untuk menandakan bahwa class ini merupakan class yang isinya unit test untuk HelloWorld.class.



Selain itu, package dari `HelloWorldTest.class` juga disamakan dengan `HelloWorld.class` yang dimaksud, tujuannya biar ngak ambigu atau bingung kalau ada class `HelloWorld` di package lain juga.

Test cases dari suatu class ditulis dalam bentuk method-method dengan annotation `@Test` di class unit test.

Oh iya, penamaan method bisa disesuaikan dengan test case yang dilakukan, ya~



Berikut adalah contoh dari unit test dari sebuah class bernama Calculator.

Penamaan class dari unit test ini harus sesuai sama kaidah yang udah dijelaskan sebelumnya, ya.

- **Annotation @Test** merupakan annotation yang **wajib** supaya test case ini bisa dijalankan. Kalau nggak ada annotation ini, berarti method nggak bakal dikenali sebagai test case.

```
public class CalculatorTest {  
  
    @Test  
    @DisplayName("Do Nothing")  
    void testNothing(){  
    }  
  
    @Test  
    @DisplayName("Positive Test – Successful Addition")  
    void testAdditionSuccess(){  
        Calculator calculator = new Calculator();  
        var result = calculator.add(5, 10);  
        Assertions.assertEquals(15, result);  
    }  
}
```

- **Annotation** `@DisplayName` merupakan annotation yang **nggak wajib**. Annotation ini digunakan untuk membuat custom display name dari test case ini saja

Oh iya, penamaan method perlu disesuaikan sama test case yang dilakukan, ya. Penamaan method yang nggak sesuai bisa menyebabkan kebingungan.

Jadi, jangan kayak si dia yang kalau ditanya mau apa, jawabnya malah “terserah kamu aja” ya~

```
public class CalculatorTest {  
  
    @Test  
    @DisplayName("Do Nothing")  
    void testNothing(){  
    }  
  
    @Test  
    @DisplayName("Positive Test – Successful Addition")  
    void testAdditionSuccess(){  
        Calculator calculator = new Calculator();  
        var result = calculator.add(5, 10);  
        Assertions.assertEquals(15, result);  
    }  
}
```

Content dari method juga bisa apa aja, lho. Bahkan ada juga method yang nggak punya content sama sekali.

Selain itu, di dalam suatu class unit test bisa terdiri dari lebih dari satu test case, gengs.

Mantap betul~

```
public class CalculatorTest {  
  
    @Test  
    @DisplayName("Do Nothing")  
    void testNothing(){  
    }  
  
    @Test  
    @DisplayName("Positive Test – Successful Addition")  
    void testAdditionSuccess(){  
        Calculator calculator = new Calculator();  
        var result = calculator.add(5, 10);  
        Assertions.assertEquals(15, result);  
    }  
}
```

User Testing, udah ✓

JUnit, udah ✓

Create a test, udah juga ✓

Berarti sekarang kita bakal kupas tuntas tentang **Assertions**.

Berangkattt~



“Assertions? Apaan tuh?”

Assertions adalah statement yang dipakai untuk **melakukan validasi, apakah output dari program yang berjalan udah sesuai atau belum.**

Dengan assertions, kita bisa bikin ekspektasi output dari program yang dites.

Pada JUnit, kalau kamu mau melakukan assertions, tersedia di class Assertions.



Berikut adalah contoh dari Assertions~

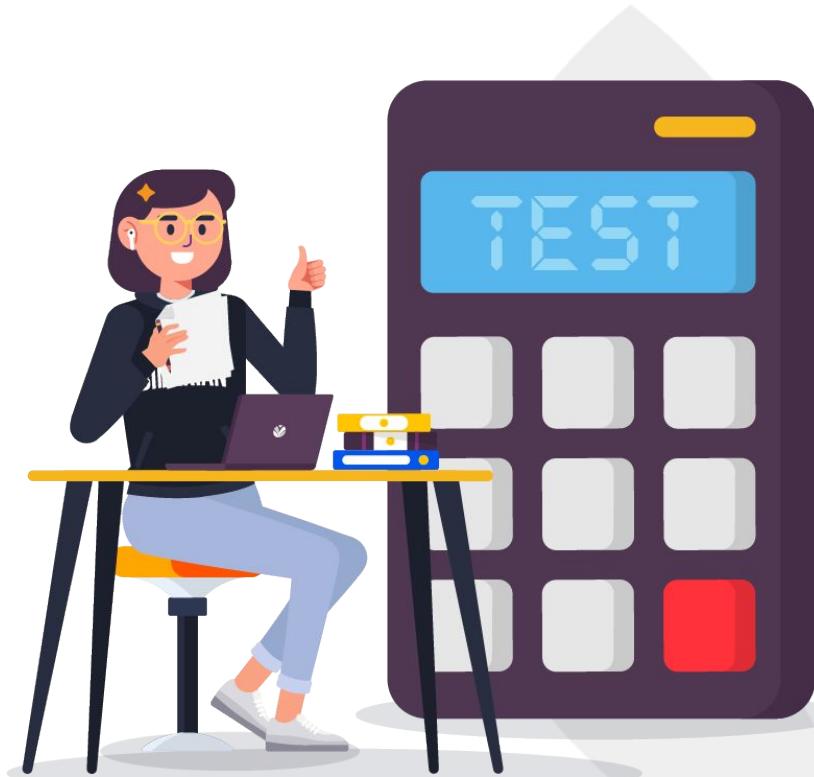
```
● ● ●

public class CalculatorTest {
    @Test
    @DisplayName("Positive Test – Successful Addition")
    void testAdditionSuccess(){
        Calculator calculator = new Calculator();
        var result = calculator.add(5, 10);
        Assertions.assertEquals(15, result);
    }
    @Test
    @DisplayName("Negative Test – Invalid Request")
    void testInvalidRequestAddition(){
        Calculator calculator = new Calculator();
        Exceptions e = Assertions.assertThrows(InvalidRequestException.class, () ->
            calculator.add(null, null));
        assertEquals("Request cannot be null", e.getMessage());
    }
}
```

Kamu tahu nggak, kalau di contoh tersebut ada dua test case, lho~

Test case yang pertama adalah **positive test case**, sedangkan yang kedua adalah **negative test case**

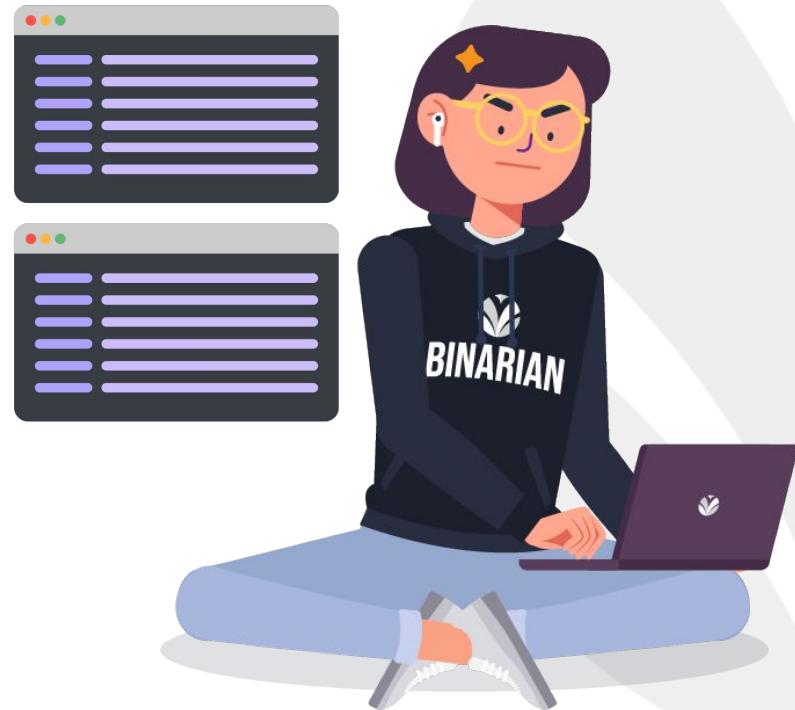
Di contoh tersebut, method add calculator merupakan operasi penambahan yang mau diuji pakai unit test. Parameter dari method add adalah dua buah integer.



Kalau kita mau menguji output dari method tersebut, kita bisa pakai method assertEquals.

Method ini punya parameter expected result dan output dari method yang diuji.

Test bakal passed kalau output sesuai sama expected result, tapi kalau output-nya nggak sesuai sama expected result, test bakalan failed, sob.



Misalnya di Class calculator, kalau parameternya null., berarti InvalidRequestException bakal dilempar.

Buat menguji apakah Exception yang dilempar udah sesuai atau belum, kita bisa pakai **assertThrows**.

Sedangkan untuk menguji message dari exception tersebut, kita bisa pakai **assertEquals**.



Setelah create a test, selanjutnya ada annotation yang bakal membantu eksekusi test makin efektif dan efisien, yaitu **BeforeEach** dan **AfterEach**.

Cekidot~

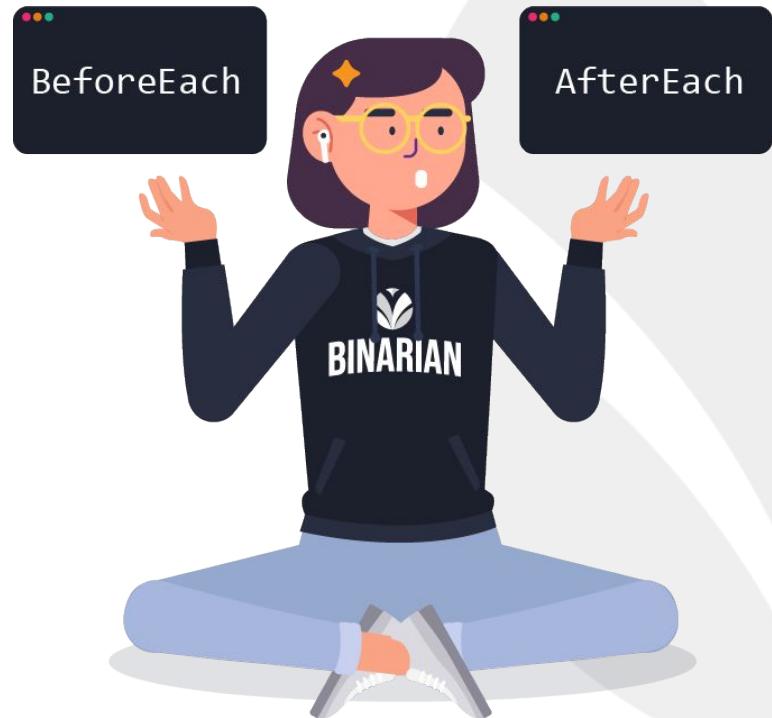


“Emangnya BeforeEach dan AfterEach itu penting, ya?”

Penting, dong!

Kita bisa aja mengeksekusi seluruh unit test dari seluruh project atau kita bisa mengeksekusi seluruh test case yang ada di dalam unit test secara langsung.

Proses eksekusinya yaitu menjalankan test case-nya secara satu per-satu secara bergiliran.



Dengan adanya @BeforeEach dan @AfterEach ini bakal bikin kita jadi satset.

Iya, karena **@BeforeEach dan @AfterEach merupakan annotation untuk membantu persiapan dari setiap eksekusi test case.**

Hal ini dilakukan supaya hasil pengetesan dari satu test case ke test case lainnya nggak berpengaruh pada test case berikutnya.



Terus, bedanya @BeforeEach dan @AfterEach itu apa?

Ini dia perbedaannya~

- Method dengan annotation `@BeforeEach` dieksekusi setiap test case **akan dieksekusi**.
- Sedangkan Method dengan `@AfterEach` dieksekusi setiap test case **sudah dieksekusi**.

Coba deh kita lihat contoh setelah ini~

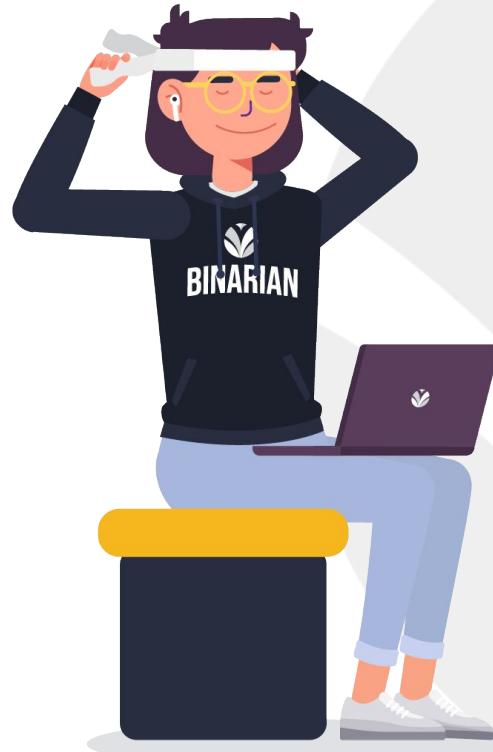


```
public class CalculatorTest {
    @Test
    @DisplayName("Positive Test – Successful Addition")
    void testAdditionSuccess(){
        Calculator calculator = new Calculator();
        var result = calculator.add(5, 10);
        Assertions.assertEquals(15, result);
    }
    @Test
    @DisplayName("Negative Test – Invalid Request")
    void testInvalidRequestAddition(){
        Calculator calculator = new Calculator();
        Exceptions e = Assertions.assertThrows(InvalidRequestException.class, () ->
calculator.add(null, null));
        assertEquals("Request cannot be null", e.getMessage());
    }
}
```

**“Kalau ada banyak test class,
gimana?”**

Tenang aja, sobat. Kalau ada puluhan test case di dalam class tersebut, maka proses instance dari class Calculator bakal dilakukan berkali-kali dan code-nya bakal jadi redundant.

Canggih kan?!



```
public class CalculatorTest {  
  
    Calculator calculator;  
  
    @BeforeEach  
    void setup(){  
        calculator = new Calculator();  
    }  
    @Test  
    @DisplayName("Positive Test – Successful Addition")  
    void testAdditionSuccess(){  
        var result = calculator.add(5, 10);  
        Assertions.assertEquals(15, result);  
    }  
    @Test  
    @DisplayName("Negative Test – Invalid Request")  
    void testInvalidRequestAddition(){  
        Exceptions e = Assertions.assertThrows(InvalidRequestException.class, () ->  
calculator.add(null, null));  
        assertEquals("Request cannot be null", e.getMessage());  
    }  
}
```

Jadi, manfaat `@BeforeEach` dan `@AfterEach` itu apa?

Kalau pakai annotation `@BeforeEach`, kita bisa mengeksekusi code yang memang diperlukan di setiap test case yang bakal dieksekusi.

Begitu juga dengan `@AfterEach`, kita bisa mengeksekusi code yang memang diperlukan buat dieksekusi setelah test case dieksekusi.



Eitss, ternyata ada annotation yang mirip sama `@BeforeEach` dan `@AfterEach` sob!

Annotation lain yang mirip sama `@BeforeEach` dan `@AfterEach`, yaitu **`@BeforeAll` dan `@AfterAll`**.

Terus bedanya di mana?

Bedanya, method dari `@BeforeAll` dan `@AfterAll` dijalankan sebelum atau sesudah keseluruhan test di dalam suatu class dijalankan, gengs~



Selanjutnya kita bakal kupas tuntas tentang **TDD Concept**.

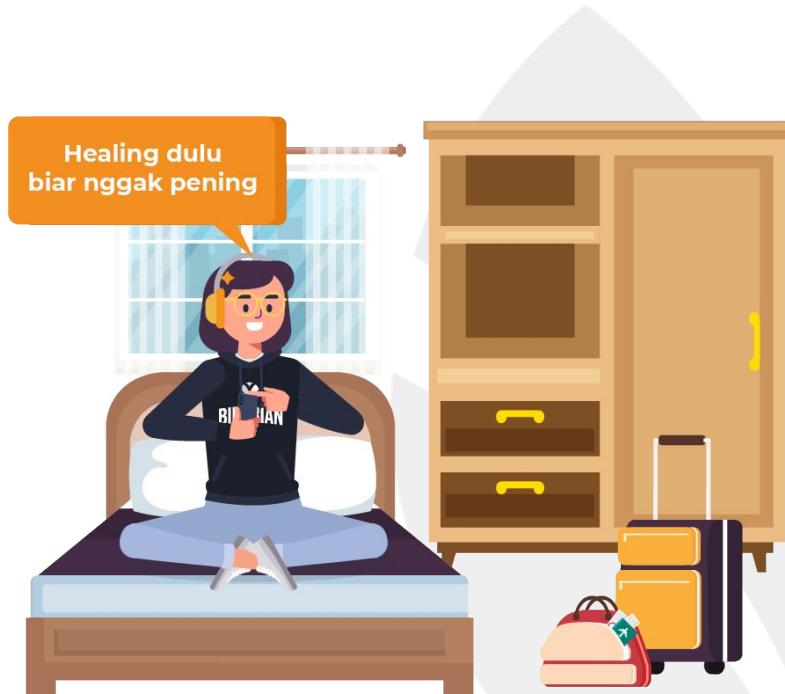
TDD sendiri merupakan singkatan dari Test Driven Development.

Hmm.. apakah TDD merupakan jenis test baru lagi? Yuk, kita cari tahu!



Dari pengalaman kalian membuat kode, pasti nggak jauh dari yang namanya bug. Kejadian ini bikin kalian capek dan akhirnya harus healing dulu.

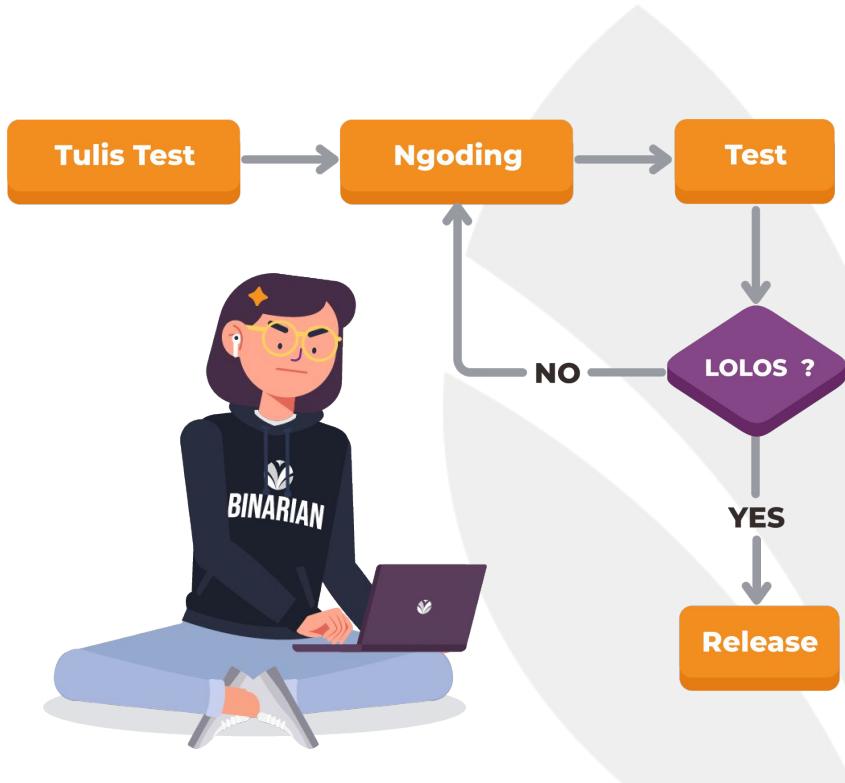
Nah, karena bug ini sesuatu yang nggak bisa kita hempaskan dalam bumi ini, terciptalah prinsip development TDD untuk mengatasi masalah bug ini.



Test dulu, production code kemudian~

TDD adalah akronim dari Test Driven Development.

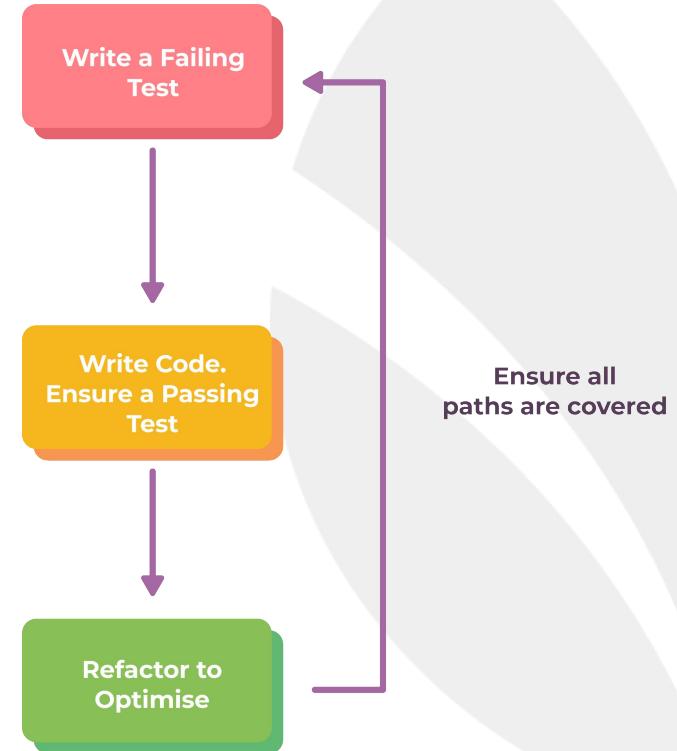
TDD merupakan sebuah pendekatan dari suatu development, yaitu dengan cara **mengembangkan** development pada suatu test case terlebih dahulu supaya test case tersebut bisa memvalidasi code yang akan berjalan.



Berikut adalah alur TDD secara umum ya, sob!

Test case bakal dibikin terlebih dahulu. Tapi, testnya ini bakal dibikin gagal karena test case belum menggunakan code dengan business logic-nya.

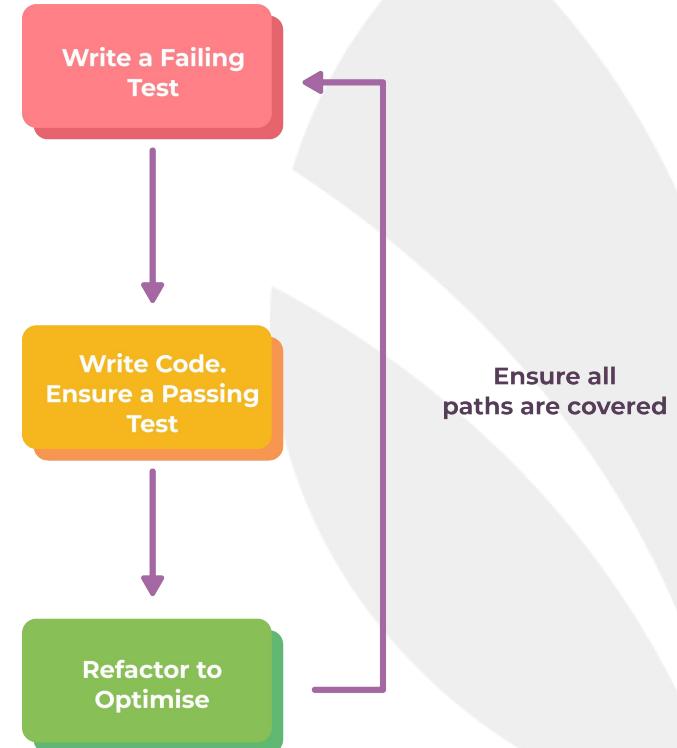
Setelah itu, nantinya code dari business logic baru bakal dibikin, sehingga test bisa jadi pass.



Kalau yang ini adalah alur TDD pada Java~

Pada Java, class dari unit test dibikin terlebih dahulu. Pas udah bikin test case, class yang isinya business logic baru dibikin sesuai dengan package yang dibuat pada class unit test.

Implementasi dari business logic ditulis satu-per satu dari test case yang udah dibuat, jadi nantinya seluruh test case bisa passed.



“Terus kelebihan kalau pakai TDD itu apa, dong?”

Keuntungan dari TDD adalah sebelum melakukan development, para developer ini telah memiliki **ekspektasi-ekspektasi dari code yang bakal dibuat.**



Dari ekspektasi ini, code dari business logic bakal dites secara menyeluruh sampai ekspektasi-ekspektasi tersebut bisa tercapai.

Hal ini bakal menjamin mutu dari code tersebut karena meningkatkan ketelitian dari code yang dibuat.



Kalau ada kelebihan, pasti ada kekurangannya juga dong~

Kekurangan TDD adalah proses development yang **butuh waktu yang lebih lama**.

Kadang nih ya, untuk melakukan TDD di modul yang kompleks butuh waktu untuk menentukan test case yang diperlukan, bahkan untuk meng-cover edge cases yang juga sama.



Selain itu, diperlukan juga pengetahuan yang mendalam terhadap modul tersebut supaya bisa menghasilkan test case yang berkualitas untuk meng-cover semua ekspektasi.



Pengumuman

Sebelum kita move on ke Quiz, silakan **implementasikan unit test pada project Java yang kamu buat di pertemuan sebelumnya!**

Latihan ini dilakukan di kelas dan silahkan diskusikan hasil jawabannya dengan teman sekelas serta fasilitator.

Selamat mencoba, yaa~



TDD alias testing dulu baru coding emang ciamik deh! Dengan TDD, ketelitian coding yang kita buat bisa meningkat.

Walaupun begitu, TDD ini emang perlu waktu development yang lama sob 😢

Nah kalo menurutmu, apakah kita perlu untuk melakukan TDD di setiap pengembangan produk kita? Atau hanya pada case tertentu aja? Boleh dong bisikin Sabrina jawabannya~



Nah, selesai sudah pembahasan kita di Chapter 3 Topic 2 ini.

Selanjutnya, kita bakal bahas tentang Unit Testing Part 2 alias lebih detail ngomongin cara mengintegrasikan test pada unit test.

Penasaran kayak gimana? Yuk langsung ke topik selanjutnya~

