Daftar Isi

Kenapa Cucumber?

Dalam dunia pengujian perangkat lunak, khususnya dalam pengujian otomatisasi, pemilihan framework yang tepat sangat penting untuk memastikan efisiensi, keterbacaan, dan kolaborasi yang optimal. Cucumber adalah salah satu framework pengujian yang populer, terutama karena pendekatannya yang berbasis Behavior-Driven Development (BDD). Namun, mengapa kita harus memilih Cucumber dibandingkan framework lain? Berikut adalah beberapa alasan utama.

Pendekatan Behavior-Driven Development (BDD)

Cucumber memungkinkan pengujian ditulis dalam format yang dapat dimengerti oleh semua pemangku kepentingan, termasuk pengembang, penguji, dan tim bisnis. Dengan menggunakan Gherkin sebagai bahasa deskriptif berbasis teks, Cucumber menjembatani kesenjangan komunikasi antara tim teknis dan non-teknis.

Sebagai contoh, skenario pengujian dalam Cucumber ditulis dengan struktur yang mudah dipahami:

```
Feature: Login ke aplikasi
  Scenario: Pengguna berhasil login dengan kredensial valid
    Given Pengguna berada di halaman login
    When Pengguna memasukkan username dan password yang valid
    And Pengguna menekan tombol login
    Then Pengguna diarahkan ke halaman utama aplikasi
```

Format ini memungkinkan semua pihak memahami apa yang diuji tanpa harus memahami kode pemrograman secara mendalam.

Kolaborasi Lebih Baik antara Tim

Karena skenario pengujian ditulis dalam bahasa alami, komunikasi antara tim bisnis dan tim teknis menjadi lebih efektif. Tim QA dapat menulis skenario pengujian tanpa perlu terlalu bergantung pada tim pengembang, sementara stakeholder bisnis dapat memahami cakupan pengujian secara langsung.

Meningkatkan Reusability dan Maintainability

Cucumber memisahkan langkah-langkah pengujian (step definitions) dari spesifikasi bisnis yang ditulis dalam Gherkin. Dengan pendekatan ini, kode pengujian lebih modular, dapat digunakan kembali, dan lebih mudah dikelola dalam jangka panjang. Jika ada perubahan

dalam aplikasi, hanya langkah-langkah tertentu yang perlu diperbarui tanpa harus menulis ulang seluruh skenario.

Sebagai contoh, implementasi dari skenario Gherkin di atas dalam Java dapat seperti ini:

```
@Given("Pengguna berada di halaman login")
public void penggunaDiHalamanLogin() {
  driver.get("https://contohapp.com/login");
}

@When("Pengguna memasukkan username dan password yang valid")
public void penggunaMemasukkanKredensialValid() {
  driver.findElement(By.id("username")).sendKeys("user");
  driver.findElement(By.id("password")).sendKeys("password123");
}

@And("Pengguna menekan tombol login")
public void penggunaMenekanTombolLogin() {
  driver.findElement(By.id("loginButton")).click();
}

@Then("Pengguna diarahkan ke halaman utama aplikasi")
public void penggunaDiHalamanUtama() {
  Assert.assertTrue(driver.findElement(By.id("dashboard"))
    .isDisplayed());
}
```

Kode ini menunjukkan bagaimana Gherkin diterjemahkan ke dalam implementasi berbasis Java, sehingga dapat dijalankan sebagai bagian dari pengujian otomatis.

Dukungan Multi-Bahasa dan Integrasi yang Kuat

Cucumber mendukung berbagai bahasa pemrograman seperti Java, JavaScript, Ruby, dan Python, sehingga dapat digunakan dalam berbagai lingkungan pengembangan. Selain itu, Cucumber juga dapat dengan mudah diintegrasikan dengan berbagai alat otomatisasi lainnya seperti:

- **Selenium/Appium:** untuk pengujian UI dan mobile
- **JUnit/TestNG**: untuk pengelolaan pengujian di Java
- **Extent Reports/Allure:** untuk pelaporan pengujian yang lebih interaktif

Dokumentasi Otomatis

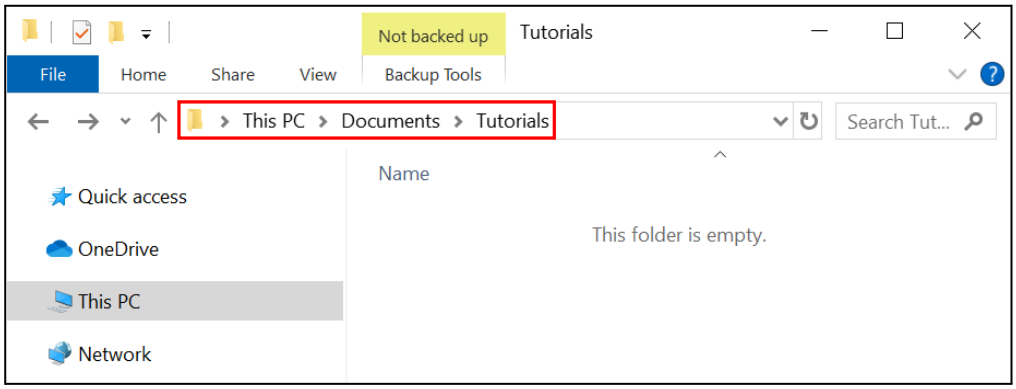
Karena skenario pengujian ditulis dalam bahasa alami, Cucumber secara otomatis berfungsi sebagai dokumentasi hidup dari sistem yang diuji. Hal ini sangat berguna dalam proyek jangka panjang karena tim dapat dengan mudah memahami dan memperbarui skenario pengujian tanpa kehilangan konteks bisnis.

Membuat Project Pertama

Pada bab ini kita akan mempelajari cara membuat project pertama untuk automation testing menggunakan Cucumber dan mengintegrasikannya dengan library lain seperti Selenium dan TestNG.

Membuat Project

Langkah pertama, kita harus menyiapkan folder untuk menyimpan project yang akan kita buat. Di sini saya akan menyimpannya di dalam direktori **Documents\Tutorials**:



Buka command prompt di direktori saat ini. Kemudian, jalankan perintah berikut untuk membuat project bernama `BelajarCucumber`:

```
mvn archetype:generate -DgroupId=com.belajar.cucumber
-DartifactId=BelajarCucumber
-DarchetypeArtifactId=maven-archetype-quickstart
-DinteractiveMode=false
```

Buka project yang baru saja kita buat dengan VS Code:

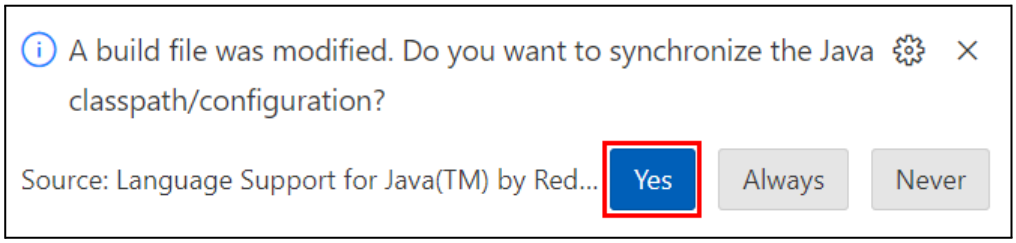
```
cd BelajarCucumber
code .
```

Menambahkan Dependensi

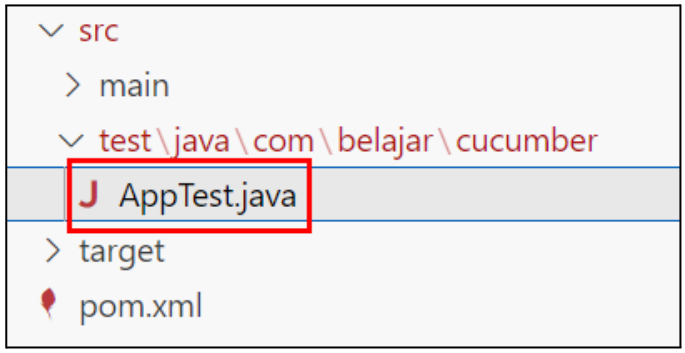
Buka file **pom.xml** dan tambahkan dependensi-dependensi yang kita perluka seperti berikut:

```
<dependencies>
  <dependency>
    <groupId>org.slf4j</groupId>
    <artifactId>slf4j-simple</artifactId>
    <version>1.7.36</version>
  </dependency>
  <dependency>
    <groupId>org.testng</groupId>
    <artifactId>testng</artifactId>
    <version>7.10.2</version>
    <scope>test</scope>
  </dependency>
  <dependency>
    <groupId>io.cucumber</groupId>
    <artifactId>cucumber-java</artifactId>
    <version>7.15.0</version>
  </dependency>
  <dependency>
    <groupId>io.cucumber</groupId>
    <artifactId>cucumber-testng</artifactId>
    <version>7.15.0</version>
    <scope>test</scope>
  </dependency>
</dependencies>
```

Setelah melakukan perubahan pada file **pom.xml**, simpan dengan menekan **CTRL + S**. Akan muncul jendela kecil di pojok kanan bawah untuk melakukan synchronize. Klik saja tombol **Yes**:



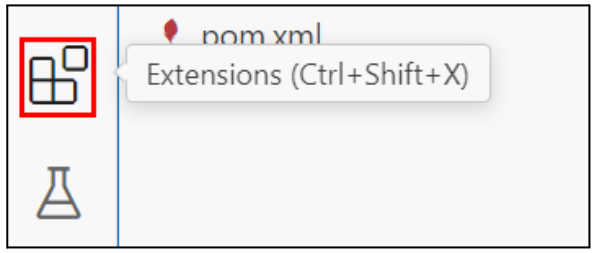
Setelah itu, hapus file **AppTest.java** dari package `com.belajar.cucumber` pada folder **test\java**:



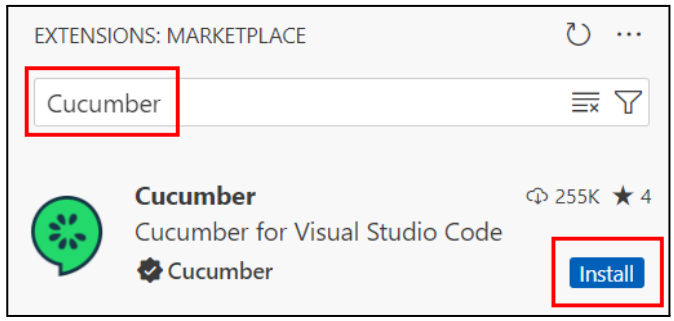
Kita menghapus file **AppTest.java** karena file tersebut mengimplementasikan test dengan JUnit sedangkan kita sudah menghapus JUnit dari **pom.xml** dan menggantinya dengan TestNG.

Instalasi Gherkin Extension

Ekstensi Gherkin di VS Code dirancang untuk mendukung penulisan file **.feature** yang digunakan dalam pengujian berbasis Behavior-Driven Development (BDD). Untuk menginstalnya, pertama-tama buka bagian Extensions di kiri:



Pada kolom pencarian, ketik **Cucumber**. Jika pencarian berhasil, di bawahnya terdapat **Cucumber** extension. Klik tombol **Install** untuk menginstal extension tersebut:



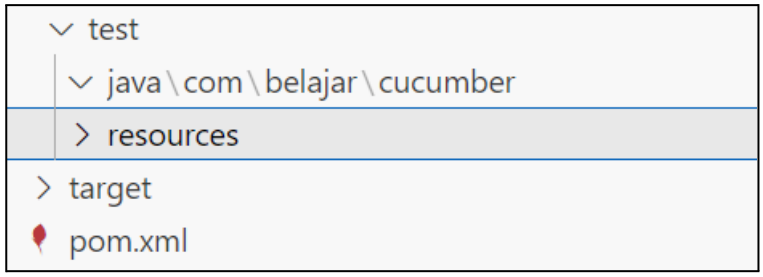
Tunggu hingga instalasinya selesai.

Membuat File Feature

File feature (dengan ekstensi **.feature**) adalah file teks yang digunakan dalam Behavior-Driven Development (BDD) untuk mendefinisikan skenario pengujian dalam format yang mudah dipahami oleh manusia.

File ini ditulis dengan bahasa Gherkin dan berisi struktur seperti `Feature`, `Scenario`, `Given`, `When` dan `Then` untuk menjelaskan fitur aplikasi dan langkah-langkah pengujiannya.

Untuk menulisnya, pertama-tama kita perlu membuat direktori **resources** di dalam direktori **test**:



Kemudian, di dalam **resources** buat direktori baru bernama **features**:

Selanjutnya, buat file bernama **Persegi.feature** di dalam folder **features**:

Jika sudah, buka file tersebut dan isi dengan kode Gherkin berikut ke dalamnya:

```
Feature: Persegi dalam Matematika Bangun Datar

  Scenario: Hitung luas persegi panjang
```

```
    Given saya punya meja dengan panjang 10 cm.
    And lebarnya 12 cm.
    When saya menghitung luas meja tersebut.
    Then hasilnya harus 120.
```

Membuat Definitions dan Runners

Buat dua buah package baru bernama `definitions` dan `runners` di dalam package sebelumnya, yaitu package `com.belajar.cucumber` pada folder **test\java**:

Buat class baru di dalam package `definitions` dengan nama `PersegiPanjangDefinition`. Kemudian ketik kode berikut ke dalamnya:

```
package com.belajar.cucumber.definitions;

import org.testng.Assert;

import io.cucumber.java.After;
import io.cucumber.java.Before;
import io.cucumber.java.en.And;
import io.cucumber.java.en.Given;
import io.cucumber.java.en.Then;
import io.cucumber.java.en.When;

public class PersegiPanjangDefintion {
  private int panjang;
  private int lebar;
  private int hasil;

  @Before
```

```
  public void setup() {
    panjang = 0;
    lebar = 0;
    hasil = 0;
  }

  @Given("saya punya meja dengan panjang {int} cm.")
  public void punyaPanjang(Integer panjang) {
    this.panjang = panjang;
  }

  @And("lebarnya {int} cm.")
  public void punyaLebar(Integer lebar) {
    this.lebar = lebar;
  }

  @When("saya menghitung luas meja tersebut.")
  public void calculate() {
    hasil = panjang * lebar;
  }

  @Then("hasilnya harus {int}.")
  public void resultMustBe(Integer expected) {
    Assert.assertEquals(hasil, expected);
  }

  @After
  public void teardown() {
    hasil = 0;
  }
}
```

Kemudian, buat class baru bernama `RunnerTest` di dalam package `runners` dan ketik kode berikut ke dalamnya:

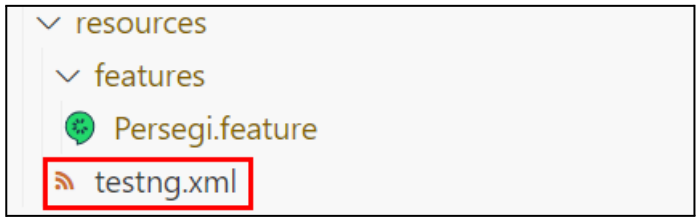
```
package com.belajar.cucumber.runners;

import io.cucumber.testng.AbstractTestNGCucumberTests;
import io.cucumber.testng.CucumberOptions;

@CucumberOptions(
  features = {
```

```
    "src/test/resources/features/Persegi.feature"
  },
  glue = {
    "com.belajar.cucumber.definitions"
  }
)
public class RunnerTest extends AbstractTestNGCucumberTests { }
```

Konfigurasi TestNG

TestNG (Test Next Generation) adalah framework pengujian yang dirancang untuk meningkatkan proses pengujian otomatis. Untuk mengintegrasikan TestNG dengan Cucumber, kita perlu membuat file konfigurasi bernama **testng.xml**.

Buat file **testng.xml** di dalam folder **resources**:



Isi file tersebut dengan kode berikut:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE suite SYSTEM "http://testng.org/testng-1.0.dtd">
<suite name="Belajar Cucumber">
  <test name="Test Demo">
    <classes>
      <class
name="com.belajar.cucumber.runners.RunnerTest"></class>
    </classes>
  </test>
</suite>
```

Surefire Plugin

Surefire plugin kita gunakan untuk menjalankan unit test atau test automation. Lebih khususnya, plugin ini digunakan untuk mengeksekusi file konfigurasi **testng.xml** yang sudah kita buat, yang berisi definisi pengujian dan class yang kita uji.

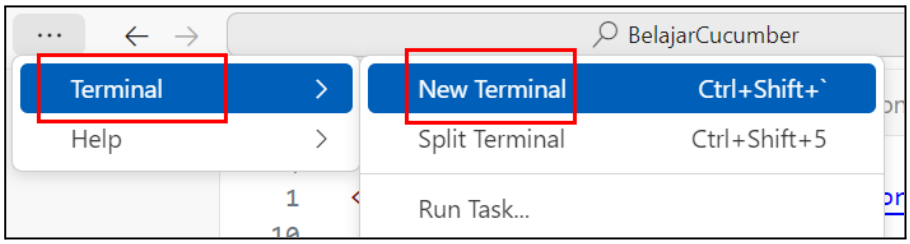
Untuk melakukannya, buka file **pom.xml** dan tambahkan konfigurasi Surefire plugin berikut (di bawah penutup `<dependencies>`):

```
<build>
  <pluginManagement>
    <plugins>
      <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-surefire-plugin</artifactId>
        <version>3.2.5</version>
        <configuration>
          <suiteXmlFiles>

<suiteXmlFile>src/test/resources/testng.xml</suiteXmlFile>
          </suiteXmlFiles>
        </configuration>
      </plugin>
    </plugins>
  </pluginManagement>
</build>
```

Jangan lupa untuk synchronize setelah kita melakukan perubahan pada file **pom.xml**.

Menjalankan Test

Buka menu **Terminal** dan klik **New Terminal**:



Di terminal yang sudah terbuka (di bagian bawah), jalankan perintah berikut:

```
mvn test
```

Saat test dijalankan, aplikasi web demo akan melakukan automation sesuai dengan kode test yang kita buat. Jika berhasil, hasilnya:

```
[INFO] Results:
[INFO]
[INFO] Tests run: 1, Failures: 0, Errors: 0, Skipped: 0
[INFO]
[INFO]
------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO]
------------------------------------------------------------------------
```

Apa itu Gherkin?

Gherkin menggunakan serangkaian kata kunci khusus untuk memberikan struktur dan makna pada spesifikasi yang dapat dieksekusi. Setiap kata kunci berisi bahasa lisan yang mudah dipahami oleh kita sebagai manusia seperti Bahasa Inggris, Indonesia dan lain sebagainya. Namun, di seluruh buku ini ini, kita akan menggunakan Bahasa Inggris. Gherkin memiliki beberapa tujuan:

- Spesifikasi dapat dijalankan dengan jelas.
- Automation testing dengan Cucumber.
- Mendokumentasikan bagaimana sistem sebenernya berperilaku.

Sebagian besar baris-baris di dalam file Gherkin dimulai dengan salah satu kata kunci. Baris komentar boleh kita tempatrkan di mana saja dalam file yang dimulai dengan tanda pagar (#).

Sintaks Gherkin ditulis dalam bahasa Inggris sederhana (atau bisa juga dalam bahasa lain yang didukung), membuatnya dapat dipahami oleh pemangku kepentingan bisnis (business stackholders) yang mungkin tidak memiliki latar belakang teknis.

Apa itu File Feature?

File feature adalah tempat kita menyimpan feature, feature description dan scenario yang akan kita test.

File feature terdiri dari kata kunci yang disebutkan di bawah ini:

Feature

`Feature` digunakan untuk memberikan deskripsi tentang skenario pengujian serta mengelompokkan skenario yang saling berkaitan. Dalam file feature, `Feature` diikuti dengan tanda titik dua (`:`) dan teks singkat yang menjelaskan fitur yang diuji.

Scenario

`Scenario` digunakan untuk mendeskripsikan langkah-langkah yang perlu dilakukan dalam suatu pengujian serta hasil yang diharapkan dari skenario tersebut.

Scenario Outline

`Scenario Outline` memungkinkan eksekusi satu skenario pengujian yang sama berulang kali dengan berbagai kumpulan data yang berbeda.

Background

Jika ditemukan bahwa langkah-langkah `Given` dalam beberapa skenario di dalam satu file feature selalu sama, maka `Background` dapat digunakan. Langkah-langkah `Given` yang berulang dapat dipindahkan ke bagian `Background`, sehingga tidak perlu ditulis ulang di setiap skenario.

Steps (Langkah-langkah dalam Gherkin)

Setiap langkah dalam skenario dimulai dengan salah satu kata kunci berikut: `Given`, `When`, `Then`, `And` atau `But`. Cucumber akan mengeksekusi setiap langkah dalam skenario satu per satu sesuai urutan yang telah ditentukan. Saat Cucumber mengeksekusi sebuah step atau langkah, ia akan mencari definisi langkah (step definition) yang sesuai untuk dijalankan.

- `Given`: menjelaskan kondisi awal sebelum skenario dijalankan. Saat langkah `Given` dieksekusi, sistem dikonfigurasi ke keadaan tertentu, misalnya dengan membuat dan mengonfigurasi objek atau menambahkan data ke dalam basis data pengujian.
- `When`: menjelaskan aksi atau kejadian dalam skenario. Aksi ini bisa berupa interaksi pengguna dengan sistem atau event yang dipicu oleh sistem lain.
- `Then`: menjelaskan hasil yang diharapkan dari skenario pengujian.
- `And`/`But`: jika terdapat lebih dari satu langkah `Given`, `When` atau `Then`, maka kata `And` atau `But` dapat digunakan untuk menghubungkan langkah-langkah tersebut agar lebih mudah dibaca.

Berikut adalah contoh file feature punya kita di bab sebelumnya:

```
Feature: Persegi dalam Matematika Bangun Datar

  Scenario: Hitung luas persegi panjang
    Given saya punya meja dengan panjang 10 cm.
    And lebarnya 12 cm.
    When saya menghitung luas meja tersebut.
    Then hasilnya harus 120.
```

Apa itu Step Definition?

Step definition adalah sebuah method dalam Java yang memiliki ekspresi untuk menghubungkannya dengan langkah-langkah (steps) dalam Gherkin. Ketika Cucumber mengeksekusi sebuah langkah Gherkin, Cucumber akan mencari step definition yang sesuai untuk dijalankan.

Cucumber membaca skrip Gherkin dan mencocokkan setiap langkah dengan potongan kode yang dapat dieksekusi (dikenal sebagai step definition). Step definition ini ditulis dalam bahasa pemrograman seperti Java, Ruby atau Python. Fungsinya adalah untuk menjalankan langkah-langkah yang tertulis dalam Gherkin guna memverifikasi perilaku aplikasi.

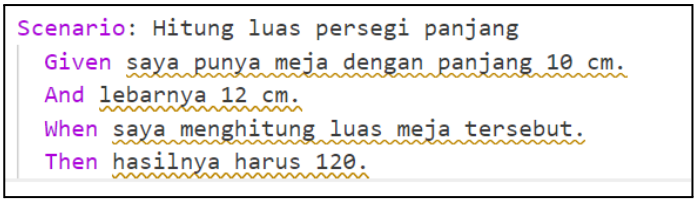
Sebagai contoh, pada Gherkin di file feature terdapat:

```
Feature: Persegi dalam Matematika Bangun Datar

  Scenario: Hitung luas persegi panjang
    Given saya punya meja dengan panjang 10 cm.
```

Maka, implementasi step definition di Java adalah sebagai berikut:

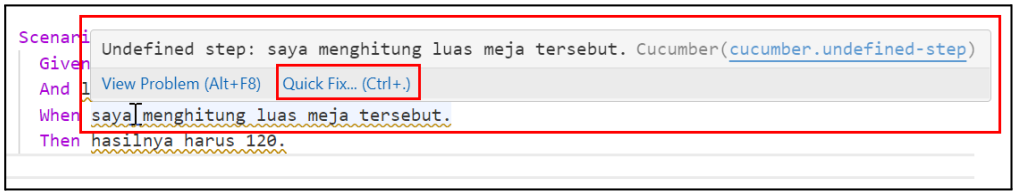
```
@Given("saya punya meja dengan panjang {int} cm.")
public void punyaPanjang(Integer panjang) {
  this.panjang = panjang;
}
```

Jika sebuah langkah dalam Scenario belum diimplementasikan dalam step definition, maka langkah tersebut akan ditandai dengan garis bawah bergerigi berwarna kuning. Ini menunjukkan bahwa Cucumber belum menemukan definisi langkah yang sesuai di dalam kode Java:



Untuk mengatasi ini, kita perlu membuat step definition yang sesuai dengan langkah yang belum diimplementasikan tersebut. Biasanya, kamu juga bisa menjalankan tes untuk mendapatkan snippet yang bisa langsung digunakan sebagai dasar dalam implementasi.

Jika kursor mouse diarahkan ke langkah yang bergaris bawah kuning dalam setiap step, VS Code akan menampilkan saran untuk menambahkan langkah tersebut ke step definition. Biasanya, akan muncul seperti yang ditunjukkan pada gambar berikut:



Untuk solusinya, klik **Quick Fix**. Di sini VS Code akan bertanya dimanakah kita ingin mengimplementasikan step ini sebagai step definition:

Jika kita klik yang ditandai dengan angka **1**, maka method step definition untuk implementasi step tersebut akan dibuat otomatis di dalam class `PersegiPanjangDefinition`:

```
public class PersegiPanjangDefintion {

  @When("saya menghitung luas meja tersebut.")
  public void saya_menghitung_luas_meja_tersebut() {
    // Write code here that turns the phrase above into concrete actions
  }
```

Hanya saja nama method yang dibuat tidak sesuai dengan konvensi penamaan nama method di Java.

File Feature

Pada bab sebelumnya kita sudah mempelajari tentang Gherkin. Pada bab ini kita akan fokus apa yang ada di dalam file feature dan bagaimana membuat file ini kembali. Di bab ini kita masih menggunakan project sebelumnya yang sudah pernah kita buat di bab dua.

Apa itu File Feature?

Dalam Cucumber, pengujian dikelompokkan ke dalam feature. File feature adalah tempat di mana kita menyimpan deskripsi fitur dan skenario yang akan diuji. File ini digunakan untuk memberikan gambaran umum tentang skenario pengujian. File ini juga berfungsi sebagai titik masuk utama untuk menjalankan pengujian dengan Cucumber.

Kata kunci pertama di dalam feature file adalah `Feature`, diikuti dengan tanda titik dua (`:`) dan teks singkat yang menjelaskan fitur tersebut. Di bawah bagian `Feature`, kita dapat menambahkan teks deskriptif dalam format bebas untuk memberikan penjelasan lebih lanjut tentang fitur yang diuji. Baris deskripsi ini tidak akan diproses oleh Cucumber saat pengujian dijalankan, tetapi tetap tersedia dalam laporan pengujian (secara default, deskripsi ini disertakan dalam laporan HTML).

Nama feature dan deskripsinya adalah opsional,yaitu tidak memiliki makna khusus bagi Cucumber. Tujuannya hanya sebagai dokumentasi untuk menjelaskan feature tersebut, termasuk ringkasan feature dan aturan bisnis (acceptance criteria) yang harus dipenuhi.

Deskripsi dalam format bebas ini berakhir ketika kita mulai menulis kata kunci `Example`, `Scenario Outline` (termasuk sinonimnya) atau `Scenario`. Selain itu, kita dapat menambahkan tag di atas `Feature` untuk mengelompokkan fitur yang terkait, terlepas dari struktur file atau direktori dalam proyek. Sebuah file feature yang sederhana biasanya terdiri dari beberapa kata kunci atau bagian utama berikut:

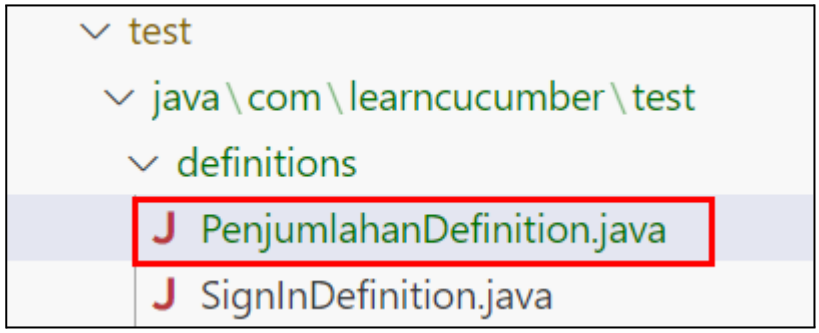
- `Feature`: menjelaskan fitur yang sedang diuji, seperti "Login ke Aplikasi Y".
- `Scenario`: Apa skenario pengujian yang ingin kita uji.
- `Given`: Prasyarat sebelum langkah-langkah pengujian dieksekusi.
- `When`: Kondisi spesifik yang harus terpenuhi untuk mengeksekusi langkah selanjutnya.
- `Then`: Apa yang seharusnya terjadi jika kondisi yang disebutkan dalam `When` terpenuhi.
- `And`: Jika kita memiliki beberapa `Given`, `When` atau `Then`, kita dapat menggunakan `And`.

Membuat File Feature dan Step Definition

Buat sebuah file feature baru bernama **Kalkulator.feature** di dalam folder **test\resources\features** dan tambahkan skenario berikut ke dalamnya:

```
Feature: Aplikasi Kalkulator
  Sebagai pengguna kalkulator,
  Saya ingin dapat menggunakan kalkulator
  untuk melakukan operasi matematika
  dengan dua nilai, agar saya bisa mendapatkan
  hasil penjumlahan dengan cepat dan akurat.

  Scenario: Menjumlahkan dua bilangan positif
    Given saya memiliki bilangan pertama 5
    And saya memiliki bilangan kedua 3
    When saya menjumlahkan kedua bilangan tersebut
    Then hasilnya harus 8.

  Scenario: Menjumlahkan bilangan positif dan nol
    Given saya memiliki bilangan pertama 7
    And saya memiliki bilangan kedua 0
    When saya menjumlahkan kedua bilangan tersebut
    Then hasilnya harus 7.

  Scenario: Menjumlahkan bilangan negatif dan positif
    Given saya memiliki bilangan pertama -4
    And saya memiliki bilangan kedua 6
    When saya menjumlahkan kedua bilangan tersebut
    Then hasilnya harus 2.

  Scenario: Menjumlahkan dua bilangan negatif
    Given saya memiliki bilangan pertama -2
    And saya memiliki bilangan kedua -3
    When saya menjumlahkan kedua bilangan tersebut
    Then hasilnya harus -5.
```

Setelah itu, buat step definition dengan nama class `PenjumlahanDefinition` di dalam package definitions pada folder **test\java**:



Tambahkan kode berikut ke dalam class `PenjumlahanDefinition`:

```
package com.belajar.cucumber.definitions;

import org.testng.Assert;

import io.cucumber.java.en.And;
import io.cucumber.java.en.Given;
import io.cucumber.java.en.Then;
import io.cucumber.java.en.When;

public class PenjumlahanDefinition {
  private int bilanganPertama;
  private int bilanganKedua;
  private int hasil;

  @Given("saya memiliki bilangan pertama {int}")
  public void setBilanganPertama(Integer bilanganPertama) {
    this.bilanganPertama = bilanganPertama;
  }

  @And("saya memiliki bilangan kedua {int}")
  public void setBilanganKedua(Integer bilanganKedua) {
    this.bilanganKedua = bilanganKedua;
  }

  @When("saya menjumlahkan kedua bilangan tersebut")
  public void calculate() {
    hasil = bilanganPertama + bilanganKedua;
  }

  @Then("hasilnya harus {int}")
```

```
  public void hasilnya(Integer expected) {
    Assert.assertEquals(hasil, expected);
  }
}
```

Method-method pada class di atas dengan anotasi `Given`, `When` dan `Then` disebut sebagai step definition yang menerapkan step dari file feature yang sudah kita buat, yaitu **KalkulatorFeature.feature**.

Pada kode di atas kita menggunakan tipe `Integer` (objek) dan bukan `int` (primitif). Dalam kasus lain, ada kemungkinan sebuah parameter tidak diberikan untuk tujuan negative testing dengan `Scenario Outline` di mana nilainya sengaja dibuat tidak valid. Jika menggunakan `int`, Cucumber tidak bisa menangani kasus di mana nilai parameter tidak ada (`null`). Dengan menggunakan `Integer`, Cucumber dapat menangani kasus ini dengan lebih baik. Artinya, jika parameter tidak diberikan, Cucumber dapat mengirim `null` ke step definition atau method tersebut. Namun, kita masih bisa merubahnya menjadi `int`.

Selanjutnya, buka class `TestRunner` dari package `.runners`. Tambahkan `Kalkulator.feature` ke dalamnya:

```
package com.belajar.cucumber.runners;

import io.cucumber.testng.AbstractTestNGCucumberTests;
import io.cucumber.testng.CucumberOptions;

@CucumberOptions(
  features = {
    "src/test/resources/features/Kalkulator.feature"
  },
  glue = {
    "com.belajar.cucumber.definitions"
  }
)
public class RunnerTest extends AbstractTestNGCucumberTests { }
```

Jalankan test dengan perintah `mvn test`. Jika test berhasil, hasilnya akan tercetak di console seperti berikut:

```
[ERROR] Failures:
[ERROR]   RunnerTest>AbstractTestNGCucumberTests.runScenario:35 »
```

```
DuplicateStepDefinition Duplicate step definitions in
com.belajar.cucumber.definitions.PenjumlahanDefinition.hasilnya(ja
va.lang.Integer) and
com.belajar.cucumber.definitions.PersegiPanjangDefintion.resultMus
tBe(java.lang.Integer)
```

Kenapa error tersebut terjadi? Error terjadi terjadi karena Cucumber menemukan duplikasi step definition untuk langkah yang sama (`Then hasilnya harus {int}`) di dua file berbeda (**Persegi.feature** dan **Kalkulator.feature**). Cucumber tidak bisa membedakan step definition yang sama untuk konteks yang berbeda (meskipun angkanya berbeda), sehingga menganggapnya sebagai duplikat.

Coba buka file **Persegi.feature**. Di dalamnya memiliki step:

```
Then hasilnya harus 120.
```

Dan pada file **Kalkulator.feature** memiliki step:

```
Then hasilnya harus 8.
Then hasilnya harus 7.
Then hasilnya harus 2.
Then hasilnya harus -5.
```

Solusinya adalah membuat step definition yang spesifik atau unik untuk setiap fitur dengan menambahkan konteks di dalam langkahnya. Dengan cara ini, Cucumber dapat membedakan step definition untuk persegi dan kalkulator tanpa menimbulkan error.

Kita akan ubah di bagian **Kalkulator.feature** menjadi seperti berikut agar berbeda dari file **Persegi.feature**:

```
Feature: Aplikasi Kalkulator
  ...

  Scenario: Menjumlahkan dua bilangan positif
    ...
    Then hasil penjumlahan harus 8

  Scenario: Menjumlahkan bilangan positif dan nol
    ...
    Then hasil penjumlahan harus 7
```

```
  Scenario: Menjumlahkan bilangan negatif dan positif
    ...
    Then hasil penjumlahan harus 2

  Scenario: Menjumlahkan dua bilangan negatif
    ...
    Then hasil penjumlahan harus -5
```

Kemudian, ubah juga method hasilnya(Integer expected) pada class PenjumlahanDefinition seperti berikut:

```
@Then("hasil penjumlahan harus {int}")
public void hasilnya(Integer expected) {
  Assert.assertEquals(hasil, expected);
}
```

Jalankan test kembali, hasilnya sekarang test berjalan dengan sukses:

```
[INFO] Results:
[INFO]
[INFO] Tests run: 4, Failures: 0, Errors: 0, Skipped: 0
[INFO]
[INFO]
------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO]
------------------------------------------------------------------------
```

Tugas Mandiri

Kamu bisa menambahkan scenario untuk menguji perhitungan lainnya seperti pengurangan, pembagian, perkalian dan modulo.

Step Definition

Step definition adalah bagian penting di Cucumber yang menghubungkan skenario dalam file feature (ditulis dalam bahasa Gherkin) dengan kode implementasi yang sebenarnya. Step definition berisi kode program yang mendefinisikan apa yang harus dilakukan ketika suatu langkah (step) dalam skenario dijalankan.

Dalam bahasa sederhana, Step definition adalah **"jembatan"** antara bahasa manusia (Gherkin) dan bahasa pemrograman (misalnya Java, Python, Ruby, dll).

Struktur Step Definition

Step Definition biasanya terdiri dari:

- **Regular Expression (Regex):** digunakan untuk mencocokkan langkah-langkah dalam skenario Gherkin.
- **Method/Function:** berisi kode implementasi yang akan dijalankan ketika langkah tersebut dipanggil.

Contohnya:

```
@Given("saya memiliki bilangan pertama {int}")
public void setBilanganPertama(Integer bilanganPertama) {
  this.bilanganPertama = bilanganPertama;
}
```

Cara kerjanya cukup mudah:

1. Cucumber membaca skenario dari file feature.
2. Cucumber mencocokkan setiap langkah (`Given`, `When`, `Then`) dengan step definition yang sesuai menggunakan regex.
3. Jika cocok, Cucumber menjalankan kode yang ada di dalam method/function tersebut.

Jika step pada file feature tidak diimplementasikan di dalam class Java sebagai method (step definition), saat menjalankan test, Cucumber akan menampilkan pesan yang memberi tahu kita untuk mengimplementasikan step tersebut dengan menampilkan contoh-contoh kode implementasi method/function di console (snippet code).

Untuk mencobanya, mari kita buat file feature baru bernama **SearchProduct.feature**. Kemudian, masukkan kode skenario berikut ke dalamnya:

```
Feature: Pencarian Produk
  Sebagai pengguna e-commerce,
  Saya ingin dapat mencari produk berdasarkan kata kunci,
  Agar saya bisa menemukan produk yang saya inginkan dengan cepat.

  Scenario: Mencari produk dengan kata kunci yang valid
    Given saya berada di halaman pencarian produk
    When saya memasukkan kata kunci "laptop"
    And saya menekan tombol cari
    Then saya harus melihat daftar produk yang mengandung kata
"laptop"
```

Setelah itu, tambahkan feature yang sudah kita buat di dalam RunnerTest seperti berikut:

```
@CucumberOptions(
  features = {
    "src/test/resources/features/SearchProduct.feature"
  },
  glue = {
    "com.belajar.cucumber.definitions"
  }
)
public class RunnerTest extends AbstractTestNGCucumberTests { }
```

Jalankan test dengan perintah `mvn test`. Di console, hasilnya akan terlihat seperti berikut:

```
[ERROR] Failures:
[ERROR]   RunnerTest>AbstractTestNGCucumberTests.runScenario:35 »
UndefinedStep The step 'saya berada di halaman pencarian produk'
and 3 other step(s) are undefined.
You can implement these steps using the snippet(s) below:

@Given("saya berada di halaman pencarian produk")
public void saya_berada_di_halaman_pencarian_produk() {
    // Write code here that turns the phrase above into concrete
actions
    throw new io.cucumber.java.PendingException();
}
@When("saya memasukkan kata kunci {string}")
public void saya_memasukkan_kata_kunci(String string) {
    // Write code here that turns the phrase above into concrete
```

```
actions
    throw new io.cucumber.java.PendingException();
}
@When("saya menekan tombol cari")
public void saya_menekan_tombol_cari() {
    // Write code here that turns the phrase above into concrete
actions
    throw new io.cucumber.java.PendingException();
}
@Then("saya harus melihat daftar produk yang mengandung kata
{string}")
public void
saya_harus_melihat_daftar_produk_yang_mengandung_kata(String
string) {
    // Write code here that turns the phrase above into concrete
actions
    throw new io.cucumber.java.PendingException();
}
```

Dari hasil yang diperoleh, setiap snippet yang dihasilkan Cucumber akan melemparkan `PendingException`. Ini adalah penanda bahwa step tersebut belum diimplementasikan. Kita dapat menyalin snippet tersebut ke dalam class sebagai method step definition dan mengimplementasikan logika yang sesuai. Hanya saja, method-method yang dihasilkan oleh Cucumber di dalam snippet di atas tidak sesuai dengan konvensi penamaan method Java pada umumnya. Jadi kita harus menyesuaikan nama method-method tersebut sesuai konvensi.

Cucumber Hook

Hook adalah blok kode yang dapat dijalankan di berbagai titik dalam siklus eksekusi Cucumber. Hook biasanya digunakan untuk pengaturan dan teardown untuk environment tertentu yang dibtuhkan sebelum dan sesudah skenario atau step dijalankan. Hook tidak memengaruhi skenario atau step. Kita dapat mendeklarasikan hook di class manapun.

Kenapa butuh Hook?

Ada kasus di mana kita harus melakukan sesuatu sebagai syarat sebelum menjalankan skenario pengujian, seperti menginisialisasi `WebDriver`, menyiapkan koneksi database, menyiapkan data pengujian, menyiapkan cookie browser dan lain sebagainya.

Demikian pula, ada beberapa kondisi yang perlu kita lakukan setelah menyelesaikan setiap skenario pengujian seperti menutup driver menggunakan method `quit()`, menutup koneksi database, menghapus data pengujian, menghapus cookie browser dan lain sebagainya.

Scenario Hook

Scenario hook dijalankan setiap skenario. Ada dua jenis scenario hook, yaitu `@Before` dan `@After`. `@Before` adalah hook yang berjalan sebelum setiap skenario sedangkan `@After` berjalan sesudah setiap skenario. Sebagai contoh:

```
public class PersegiPanjangDefintion {
  private int panjang;
  private int lebar;
  private int hasil;

  @Before
  public void setup() {
    panjang = 0;
    lebar = 0;
    hasil = 0;
  }

  ...

  @After
  public void teardown() {
    hasil = 0;
  }
}
```

Scenario hook sudah kita coba di bab kedua bagaimana membuat project pertama Cucumber. Jika kamu lupa, silahkan baca dan coba kembali bab tersebut.

Biasanya pengujian dengan hook ini dapat diterapkan menggunakan Selenium seperti potongan kode berikut untuk menginisialisasi `driver` sebelum skenario dijalankan dan menutup driver setelah skenario selesai dijalankan:

```
public class SignInDefinition {
  private WebDriver driver;
  private ChromeOptions options;

  @Before
  public void setup() {
    options = new ChromeOptions();
    options.addArguments("--start-maximized");
    driver = new ChromeDriver(options);
  }

  ...

  @After
  public void teardown() {
    driver.quit();
  }
}
```

Step Hook

Selain scenario hook, Cucumber juga menyediakan hook lain, yaitu step hook. Step hook memiliki dua jenis, yaitu `@BeforeStep` dan `@AfterStep`. **Step hook dipanggil sebelum (`@BeforeStep`) dan sesudah (`@AfterStep`) setiap step definition (method)**. Jika hook `@BeforeStep` dijalankan, hook `@AfterStep` juga akan dijalankan tanpa mempedulikan hasil.

Maksud tidak peduli dengan hasil pada hook `@AfterStep` adalah bahwa hook `@AfterStep` tidak peduli apakah langkah sebelumnya berhasil, gagal, atau dilewati (skip). Hook ini akan tetap dijalankan setelah step definition tersebut selesai diproses.

Untuk memahaminya, mari kita buat file feature baru bernama **Segitiga.feature** di dalam folder **features**. Lalu tambahkan skenario berikut ke dalamnya:

```
Feature: Menghitung Luas Segitiga
  Sebagai pengguna aplikasi kalkulator,
  Saya ingin dapat menghitung luas segitiga,
  Agar saya bisa mendapatkan hasil perhitungan
  dengan cepat dan akurat.

  Scenario: Menghitung luas segitiga dengan alas dan tinggi yang
valid
    Given saya memiliki alas segitiga 10
    And saya memiliki tinggi segitiga 5
    When saya menghitung luas segitiga
    Then hasil hitung luas segitiga harus 25
```

Langkah berikutnya, tambahkan feature tersebut ke dalam class `RunnerTest` seperti berikut:

```
@CucumberOptions(
  features = {
    "src/test/resources/features/Segitiga.feature"
  },
  ...
)
public class RunnerTest extends AbstractTestNGCucumberTests { }
```

Selanjutnya, buat class `LuasSegitigaDefinition` di package `.definitions` dan tambahkan kode berikut ke dalamnya:

```
package com.belajar.cucumber.definitions;

import org.testng.Assert;

import io.cucumber.java.AfterStep;
import io.cucumber.java.BeforeStep;
import io.cucumber.java.en.Given;
import io.cucumber.java.en.Then;
import io.cucumber.java.en.When;

public class LuasSegitigaDefinition {
  private double alas;
  private double tinggi;
  private double luas;
```

```
  @BeforeStep
  public void beforeStep() {
    System.out.println("Before Step: Persiapan sebelum menjalankan
langkah...");
  }

  @AfterStep
  public void afterStep() {
    System.out.println("After Step: Pembersihan setelah langkah
selesai...");
  }

  @Given("saya memiliki alas segitiga {double}")
  public void punyaAlas(double alas) {
    this.alas = alas;
    System.out.println("Alas segitiga: " + alas);
  }

  @And("saya memiliki tinggi segitiga {double}")
  public void punyaTinggi(double tinggi) {
    this.tinggi = tinggi;
    System.out.println("Tinggi segitiga: " + tinggi);
  }

  @When("saya menghitung luas segitiga")
  public void hitungLuas() {
    this.luas = (this.alas * this.tinggi) / 2;
    System.out.println("Menghitung luas segitiga...");
  }

  @Then("hasil hitung luas segitiga harus {double}")
  public void hasilnyaHarus(double expectedLuas) {
    System.out.println("Memeriksa hasil perhitungan...");
    Assert.assertEquals(this.luas, expectedLuas, 0.001);
    // Delta 0.001 untuk perbandingan double
  }
}
```

Jika sudah, jalankan test. Hasil yang tercetak di console adalah:

```
Before Step: Persiapan sebelum menjalankan langkah...
```

```
Alas segitiga: 10.0
After Step: Pembersihan setelah langkah selesai...
Before Step: Persiapan sebelum menjalankan langkah...
Tinggi segitiga: 5.0
After Step: Pembersihan setelah langkah selesai...
Before Step: Persiapan sebelum menjalankan langkah...
Menghitung luas segitiga...
After Step: Pembersihan setelah langkah selesai...
Before Step: Persiapan sebelum menjalankan langkah...
Memeriksa hasil perhitungan...
After Step: Pembersihan setelah langkah selesai...
[INFO] Tests run: 1, Failures: 0, Errors: 0, Skipped: 0, Time
elapsed: 0.896 s -- in TestSuite
[INFO]
[INFO] Results:
[INFO]
[INFO] Tests run: 1, Failures: 0, Errors: 0, Skipped: 0
[INFO]
[INFO]
------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO]
------------------------------------------------------------------------
```

Dari hasil yang kita peroleh, fokus pada bagian berikut:

```
Before Step: Persiapan sebelum menjalankan langkah...
Alas segitiga: 10.0
After Step: Pembersihan setelah langkah selesai...
Before Step: Persiapan sebelum menjalankan langkah...
Tinggi segitiga: 5.0
After Step: Pembersihan setelah langkah selesai...
Before Step: Persiapan sebelum menjalankan langkah...
Menghitung luas segitiga...
After Step: Pembersihan setelah langkah selesai...
Before Step: Persiapan sebelum menjalankan langkah...
Memeriksa hasil perhitungan...
After Step: Pembersihan setelah langkah selesai...
```

Kita dapat melihat bahwa `@BeforeStep` dan `@AfterStep` dijalankan sebelum dan setelah setiap langkah (step) dalam skenario.

Sebelum langkah pertama, `@BeforeStep` dijalankan di mana hook ini kita asumsikan akan melakukan persiapan sebelum langkah pertama dijalankan. Pada tahap ini, kita bisa melakukan inisialisasi data, logging atau persiapan lainnya. Kemudian, langkah pertama (`Given saya memiliki alas segitiga 10`) dijalankan dimana langkah ini digunakan berisi kode untuk menginisialisasi nilai alas segitiga. Kemudian, `@AfterStep` dijalankan di mana hook ini diasumsikan melakukan pembersihan setelah langkah pertama selesai, seperti membersihkan data sementara atau logging. Flow ini akan terus sama sampai step atau langkah ke empat.

Pada intinya, ketika test berjalan, Cucumber akan memproses setiap langkah (step) dalam skenario secara berurutan. Sebelum setiap langkah dijalankan, hook `@BeforeStep` akan dipanggil, dan setelah setiap langkah selesai, hook `@AfterStep` akan dipanggil. Ini terjadi terlepas dari apakah langkah tersebut berhasil, gagal atau dilewati.

Dari percobaan ini menunjukkan bahwa hook `@BeforeStep` dan `@AfterStep` selalu dijalankan sebelum dan setelah setiap langkah (step definition yang diwakili method), memberikan fleksibilitas untuk melakukan persiapan atau pembersihan di sekitar setiap langkah.

Jika suatu step gagal atau mengalami error atau dilewati (skip), `@AfterStep` akan tetap di jalankan. Untuk membuktikannya, kembali ke class `LuasSegitigaDefinition`. Di sini kita akan membuat sebuah error di dalam method `punyaTinggi()`:

```
@Given("saya memiliki tinggi segitiga {double}")
public void punyaTinggi(double tinggi) {
  this.tinggi = tinggi;
  System.out.println("Tinggi segitiga: " + tinggi);
  throw new IllegalArgumentException("Error:
IllegalArgumentException");
}
```

Jika test dijalankan kembali, hasilnya:

```
Before Step: Persiapan sebelum menjalankan langkah...
Alas segitiga: 10.0
After Step: Pembersihan setelah langkah selesai...
Before Step: Persiapan sebelum menjalankan langkah...
Tinggi segitiga: 5.0
After Step: Pembersihan setelah langkah selesai...
[ERROR] Tests run: 1, Failures: 1, Errors: 0, Skipped: 0
```

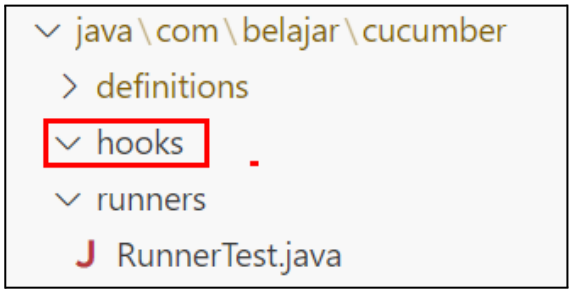
Dari hasil yang kita peroleh, pada saat error terjadi pada method `punyaTinggi(double tinggi)`, method `afterStep()` sebagai method hook dengan anotasi `@AfterStep` tetap dijalankan tanpa peduli hasil yang diperoleh dari method step `punyaTinggi(double tinggi)`. Namun, setelah itu, pengujian akan berhenti dan tidak melanjutkan ke step berikutnya (jika dan hanya jika kita menguji satu skenario. Jika pengujian menggunakan lebih dari satu skenario, maka dia akan melanjutkan skenario lainnya). Hal yang sama berlaku jika kita menggunakan Assert TestNG:

```
@Given("saya memiliki tinggi segitiga {double}")
public void punyaTinggi(double tinggi) {
  this.tinggi = tinggi;
  System.out.println("Tinggi segitiga: " + tinggi);
  Assert.assertTrue(false);
}
```

Sifat Hook

Scenario hook dan step hook bersifat global. Hook akan selalu dijalankan, tidak peduli tag-tag apapun yang kita berikan untuk mem-filter scenario (tag akan di bahas pada bab selanjutnya). Oleh sebab itu kita harus memisahkan file khusus untuk hook-hook tersebut agar lebih mudah dikelola dan tidak bercampur dengan step definition. Selain itu kode atau proyek yang kita buat tentunya lebih mudah dikelola.

Mari kita buat package bernama `hooks` di dalam package `com.belajar.cucumber` pada folder **test\java** seperti contoh berikut:



Di dalam package tersebut kita buat class bernama `Hook` dan tambahkan implementasi scenario hook dan step hook seperti berikut:

```
package com.belajar.cucumber.hooks;

import io.cucumber.java.After;
```

```
import io.cucumber.java.AfterAll;
import io.cucumber.java.AfterStep;
import io.cucumber.java.Before;
import io.cucumber.java.BeforeAll;
import io.cucumber.java.BeforeStep;

public class Hook {
  @BeforeAll
  public static void initialize() {}

  @Before
  public void setup() {}

  @BeforeStep
  public void beforeStep() {}

  @AfterStep
  public void afterStep() {}

  @After
  public void teardown() {}

  @AfterAll
  Public static void finalTeardown() {}
}
```

Untuk hook `@BeforeAll`, dijalankan hanya sekali sebelum semua scenario dijalankan dalam satu sesi uji coba. Bisa kita gunakan untuk setup awal, seperti membuka koneksi database atau menginisialisasi `WebDriver`.

Hook `@AfterAll` dijalankan hanya sekali setelah semua skenario selesai dijalankan. Digunakan untuk menutup koneksi database setelah semua pengujian selesai, menghasilkan laporan pengujian di akhir atau bahkan bisa digunakan untuk menutup browser dalam pengujian Selenium.

Method pada hook `@BeforeAll` dan `@AfterAll` harus `static`. Jika tidak, Cucumber akan menampilkan pengecualian berupa `InvalidMethodSignature`.

Hook `@Before` dijalankan sebelum setiap skenario dijalankan. Bisa kita gunakan untuk menyiapkan data awal sebelum skenario dijalankan.

Hook `@After` dijalankan setelah setiap skenario selesai dijalankan. Dapat kita gunakan untuk membersihkan data uji, menutup browser dalam pengujian Selenium dan logging atau menghapus cache.

Hook `@BeforeStep` dijalankan sebelum setiap step di dalam skenario dijalankan. Bisa kita gunakan untuk logging atau debugging sebelum setiap langkah (step) dijalankan. Selain itu kita bisa memanfaatkannya untuk mengambil screenshot sebelum menjalankan setp dalam pengujian UI.

Hook `@AfterStep` dijaankan setelah setiap step di dalam skenario dijalankan. Fungsinya sama, untuk logging dan mengambil screenshot setelah setiap langkah selesai dijalankan. Bisa juga kita gunakan untuk menyimpan informasi hasil eksekusi setiap langkah.

Untuk saat ini kita biarkan class Hook ini kosong karena kita belum menentukan aksi spesifik yang perlu dijalankan dalam setiap hook. Namun, struktur file ini tetap diperlukan agar ketika diperlukan nanti, kita bisa langsung menambahkan kode tanpa harus mengubah struktur proyek.

Selain itu, dengan adanya file ini, Cucumber tetap mengenali keberadaan hook, sehingga ketika fitur pengujian berkembang, kita bisa dengan mudah menyesuaikan kebutuhan.

Perlu diingat baik-baik, hook yang kita buat tidak akan pernah dijalankan. Jadi, jika kita ingin menjalankan hook yang ada di dalam package `.hooks` pada saat proses pengujian, kita bisa menambahkan package tersebut ke dalam `glue` pada class `RunnerTest` seperti berikut:

```
@CucumberOptions(
  ...,
  glue = {
    "com.belajar.cucumber.hooks",
    "com.belajar.cucumber.definitions",
  }
)
```

Ada hal penting yang harus kita lakukan setelah membuat class Hook. Kita perlu menghapus atau mengubah class-class pada definition step yang mengimplementasikan hook. Buka class `PersegiPanjangDefintion`, di sini kita akan menghapus implementasi scenario hook dengan mengubah kodenya menjadi seperti berikut:

```
public class PersegiPanjangDefintion {
  private int panjang;
  private int lebar;
  private int hasil;
```

```
  @Given("saya punya meja dengan panjang {int} cm.")
  public void punyaPanjang(Integer panjang) {
    this.panjang = panjang;
  }

  @And("lebarnya {int} cm.")
  public void punyaLebar(Integer lebar) {
    this.lebar = lebar;
  }

  @When("saya menghitung luas meja tersebut.")
  public void calculate() {
    hasil = panjang * lebar;
  }

  @Then("hasilnya harus {int}.")
  public void resultMustBe(Integer expected) {
    int actual = hasil;
    hasil = 0;
    Assert.assertEquals(actual, expected);
  }
}
```

Selanjutnya pada class `LuasSegitigaDefinition`, hapus implementasi step hook sehingga kode akhirnya terlihat seperti berikut:

```
public class LuasSegitigaDefinition {
  private double alas;
  private double tinggi;
  private double luas;

  @Given("saya memiliki alas segitiga {double}")
  public void punyaAlas(double alas) {
    this.alas = alas;
    System.out.println("Alas segitiga: " + alas);
  }

  @Given("saya memiliki tinggi segitiga {double}")
  public void punyaTinggi(double tinggi) {
    this.tinggi = tinggi;
    System.out.println("Tinggi segitiga: " + tinggi);
```

```
  }

  @When("saya menghitung luas segitiga")
  public void hitungLuas() {
    this.luas = (this.alas * this.tinggi) / 2;
    System.out.println("Menghitung luas segitiga...");
  }

  @Then("hasil hitung luas segitiga harus {double}")
  public void hasilnyaHarus(double expectedLuas) {
    System.out.println("Memeriksa hasil perhitungan...");
    Assert.assertEquals(this.luas, expectedLuas, 0.001);
    // Delta 0.001 untuk perbandingan double
  }
}
```

Cucumber Tag

Tag adalah cara yang mudah untuk mengatur `Feature` dan `Scenario`. Secara default, Cucumber menjalankan semua skenario yang ada dalam file feature. Jika kita ingin menjalankan skenario tertentu dari file feature, maka tag dapat digunakan. Dengan begitu kita dapat menjalankan skenario atau feature secara fleksibel berdasarkan tag tertentu.

Tag dapat dideklarasikan seperti contoh berikut:

```
@NamaTag
Scenario: Example scenario
```

Dari contoh di atas, simbol `@` digunakan untuk mendeklarasikan tag. `NamaTag` adalah nama dari tag yang ingin kita buat.

Membuat Tag

Untuk memahaminya, di sini kita akan membuat empat tag berbeda. Tag pertama adalah tag `@Auth`, tag kedua adalah `@SignIn`, tag ketiga adalah `@Registration` dan tag keempat adalah `@Checkout`.

Selanjutnya, mari kita buat dua file feature bernama **Authentication.feature** dan **Checkout.feature**. Berikut isi masing-masing feature:

Authentication.feature

```
@Auth
Feature: Auth Functionality

  @SignIn
  Scenario: Successful SignIn with valid credentials
    Given User is on the signin page
    When User enters valid username and password
    And User clicks the signin button
    Then User should be redirected to the homepage

  @Registration
  Scenario: Successful user registration
    Given User is on the registration page
    When User enters valid registration details
    And User submits the registration form
```

```
    Then User should see a success message
```

Checkout.feature

```
Feature: Checkout Process

  @Checkout
  Scenario: Successful checkout with valid payment details
    Given User has items in the cart
    When User proceeds to checkout
    And User enters valid payment details
    And User confirms the order
    Then User should see an order confirmation message
```

Langkah selanjutnya, buat tiga class definition bernama `SignInDefinition`, `RegisterDefinition` dan `CheckoutDefinition`. Berikut masing-masing kode pada setiap class:

SignInDefinition.java

```
package com.belajar.cucumber.definitions;

import io.cucumber.java.en.And;
import io.cucumber.java.en.Given;
import io.cucumber.java.en.Then;
import io.cucumber.java.en.When;

public class SignInDefinition {

  @Given("User is on the signin page")
  public void step01() {
    System.out.println("SignInDefinition:step01()");
  }

  @When("User enters valid username and password")
  public void step02() {
    System.out.println("SignInDefinition:step02()");
  }

  @And("User clicks the signin button")
  public void step03() {
```

```
    System.out.println("SignInDefinition:step03()");
  }

  @Then("User should be redirected to the homepage")
  public void step04() {
    System.out.println("SignInDefinition:step04()");
  }
}
```

RegisterDefinition.java

```
package com.belajar.cucumber.definitions;

import io.cucumber.java.en.And;
import io.cucumber.java.en.Given;
import io.cucumber.java.en.Then;
import io.cucumber.java.en.When;

public class RegisterDefinition {

  @Given("User is on the registration page")
  public void step01() {
    System.out.println("RegisterDefinition:step01()");
  }

  @When("User enters valid registration details")
  public void step02() {
    System.out.println("RegisterDefinition:step02()");
  }

  @And("User submits the registration form")
  public void step03() {
    System.out.println("RegisterDefinition:step03()");
  }

  @Then("User should see a success message")
  public void step04() {
    System.out.println("RegisterDefinition:step04()");
  }
}
```

CheckoutDefinition.java

```
package com.belajar.cucumber.definitions;

import io.cucumber.java.en.And;
import io.cucumber.java.en.Given;
import io.cucumber.java.en.Then;
import io.cucumber.java.en.When;

public class CheckoutDefinition {

  @Given("User has items in the cart")
  public void step01() {
    System.out.println("CheckoutDefinition:step01()");
  }

  @When("User proceeds to checkout")
  public void step02() {
    System.out.println("CheckoutDefinition:step02()");
  }

  @And("User enters valid payment details")
  public void step03() {
    System.out.println("CheckoutDefinition:step03()");
  }

  @And("User confirms the order")
  public void step04() {
    System.out.println("CheckoutDefinition:step04()");
  }

  @Then("User should see an order confirmation message")
  public void step05() {
    System.out.println("CheckoutDefinition:step05()");
  }
}
```

Buka class `RunnerTest`. Mari kita tambahkan tag `@SignIn` ke dalamnya:

```
@CucumberOptions(
  tags = "@SignIn",
  ...
)
```

Tambahkan juga `Authentication.feature` dan `Checkout.feature` ke dalam `features`:

```
@CucumberOptions(
  tags = "@SignIn",
  features = {
    "src/test/resources/features/Authentication.feature",
    "src/test/resources/features/Checkout.feature"
  },
  glue = {
    "com.belajar.cucumber.definitions"
  }
)
```

Langkah berikutnya mari kita coba jalankan test dengan perintah `mvn test`. Hasilnya tercetak di console seperti berikut:

```
SignInDefinition:step01()
SignInDefinition:step02()
SignInDefinition:step03()
SignInDefinition:step04()
```

Dari hasil yang diperoleh, Cucumber menjalankan class step definition `SignInDefinition` yang terkait dengan tag `@SignIn` yang ada di dalam file **Authentication.feature**.

Coba ubah `tags` di dalam `RunnerTest` menjadi `@Auth` seperti berikut:

```
@CucumberOptions(
  tags = "@Auth",
  ...
}
```

Hasil yang diperoleh setelah menjalankan test adalah:

```
SignInDefinition:step01()
SignInDefinition:step02()
SignInDefinition:step03()
SignInDefinition:step04()
```

```
RegisterDefinition:step01()
RegisterDefinition:step02()
RegisterDefinition:step03()
RegisterDefinition:step04()
```

Dua scenario yang ada dalam file **Authentication.feature** dieksekusi karena kedua skenario berada dalam file yang diberi tag `@Auth`:

```
@Auth
Feature: Auth Functionality

  @SignIn
  Scenario: Successful SignIn with valid credentials
    Given ...

  @Registration
  Scenario: Successful user registration
    Given ...
```

Tag `@Auth` sendiri kita tempatkan di level `Feature` sehingga Cucumber akan menjalankan skenario di bawahnya meskipun setiap skenario tersebut memiliki tag lain, sehingga class definition step yang terkait dengan tag `@SignIn` (class `SignInDefinition`) dan `@Registration` (class `RegisterDefinition`) juga ikut di jalankan.

Operator And

Jika kita menerapkan operator `and` pada `tags`, itu artinya skenario yang memiliki semua tag yang disebutkan akan dipilih untuk dijalankan. Cucumber hanya akan mengeksekusi skenario yang memenuhi semua kondisi tag yang diterapkan.

Misalnya, jika kita menggunakan `tags` di `RunnerTest` seperti berikut:

```
@CucumberOptions(
  tags = "@Registration and @SignIn",
  features = {
    "src/test/resources/features/Authentication.feature",
    "src/test/resources/features/Checkout.feature",
  },
  glue = {
    "com.belajar.cucumber.definitions"
  }
```

```
)
```

Hasilnya tidak ada satu test pun dijalankan. Hal ini terjadi karena kita tidak memiliki dua tag `@Registration` dan `@SignIn` bersamaan dalam satu skenario. Untuk mencobanya, di scenario registrasi, coba tambahkan tag `@Web` seperti berikut:

```
@Registration @Web
Scenario: Successful user registration
  Given User is on the registration page
  When User enters valid registration details
  And User submits the registration form
  Then User should see a success message
```

Lalu ubah `tags` di dalam `RunnerTest` menjadi seperti berikut:

```
tags = "@Web and @Registration"
```

Hasilnya, skenario yang memiliki dua tag tersebut dijalankan:

```
RegisterDefinition:step01()
RegisterDefinition:step02()
RegisterDefinition:step03()
RegisterDefinition:step04()
```

Sekarang, coba ubah tags menjadi seperti berikut:

```
tags = "@Auth and @Web"
```

Hasilnya:

```
RegisterDefinition:step01()
RegisterDefinition:step02()
RegisterDefinition:step03()
RegisterDefinition:step04()
```

Meskipun `@Auth` ada di level `Feature`, Cucumber masih memilih skenario yang memiliki tag `@Web`, selama skenario tersebut ada di dalam `Feature` yang memiliki tag `@Auth`. Oleh sebab itu skenario register yang memiliki tag `@Web` akan dijalankan (tidak peduli apakah di skenario regiser punya tag lain atau tidak).

Pada contoh di atas, kita menggunakan dua kondisi tag. Apakah bisa menggunakan lebih dari dua tag menggunakan `and`? Ya, tentu saja bisa. Kita bisa menggunakan lebih dari dua kondisi tag dengan `and` di Cucumber untuk memfilter skenario yang memenuhi semua kondisi tersebut. Cucumber mendukung penggunaan lebih dari dua kondisi tag dengan menggabungkan tag dengan operator `and` secara berurutan.

Misalnya, kita memiliki tiga tag yang berbeda di file feature: `@Auth`, `@SignIn` dan `@Mobile`. Kita ingin mengeksekusi hanya skenario yang memiliki semua tiga tag tersebut:

```
@Auth
Feature: Authentication

  @SignIn @Mobile
  Scenario: Successful SignIn from mobile device
    Given User is on the signin page
    When User enters valid username and password
    Then User should be redirected to the homepage

  @SignIn @Web
  Scenario: Successful SignIn from web
    Given User is on the signin page
    When User enters valid username and password
    Then User should be redirected to the homepage

  @Registration
  Scenario: Successful user registration
    Given User is on the registration page
    When User enters valid registration details
    Then User should see a success message
```

Di `RunnerTest`, kita bisa menulis `tags` seperti berikut:

```
tags = "@Auth and @SignIn and @Mobile"
```

Dengan kombinasi tag `@Auth and @SignIn and @Mobile`, Cucumber hanya akan mengeksekusi skenario yang memenuhi semua tiga tag tersebut. Dari contoh struktur file feature di atas, hanya skenario berikut yang akan dieksekusi:

```
@SignIn @Mobile
Scenario: Successful SignIn from mobile device
```

```
  Given User is on the signin page
  When User enters valid username and password
  Then User should be redirected to the homepage
```

Operator Or

Operator `or` digunakan untuk mengeksekusi skenario yang memiliki setidaknya satu dari beberapa tag yang diberikan. Artinya, jika sebuah skenario memiliki salah satu dari tag yang disebutkan, maka skenario tersebut akan dieksekusi. Misalnya:

```
tags = "@SignIn or @Registration"
```

Dari `tags` di atas, Cucumber akan menjalankan semua skenario yang memiliki tag `@SignIn` atau `@Registration`.

Untuk memahaminya, coba tambahkan tag `@SignIn` dan `@Checkout` di `RunnerTest` pada `CucumberOptions` seperti berikut:

```
@CucumberOptions(
  tags = "@SignIn or @Checkout",
  ...
)
```

Saat test selesai dijalankan, hasil yang diperoleh adalah sebagai berikut:

```
SignInDefinition:step01()
SignInDefinition:step02()
SignInDefinition:step03()
SignInDefinition:step04()
CheckoutDefinition:step01()
CheckoutDefinition:step02()
CheckoutDefinition:step03()
CheckoutDefinition:step04()
CheckoutDefinition:step05()
```

Dari hasil yang kita peroleh, Cucumber akan menjalankan semua skenario yang memiliki tag `@SignIn` atau `@Checkout`.

Sama seperti operator `and`. Dengan `or`, Kita juga bisa menggunakan lebih dari dua kondisi tag di Cucumber untuk memfilter skenario yang memenuhi semua kondisi tersebut:

```
tags = "@SignIn or @Checkout or @Web"
```

Operator Not

Operator `not` digunakan untuk mengecualikan skenario yang memiliki tag tertentu. Dengan kata lain, Cucumber akan menjalankan semua skenario kecuali yang memiliki tag yang dikecualikan.

Kita bisa menuliskan `not` sebelum sebuah tag untuk mengecualikan skenario yang memiliki tag tersebut. Misalnya:

```
tags = "not @Auth"
```

Setelah test selesai dijalankan, hasilnya akan tercetak seperti berikut di console:

```
CheckoutDefinition:step01()
CheckoutDefinition:step02()
CheckoutDefinition:step03()
CheckoutDefinition:step04()
CheckoutDefinition:step05()
```

Dari hasil yang kita peroleh, kode `not @Auth` menyebabkan Cucumber hanya menjalankan tag selain `@Auth`. Tag `@SignIn`, `@Registration` dan `@Web` juga tidak akan dijalankan karena ketiganya ada di dalam `Feature` yang memiliki tag `@Auth`:

```
@Auth
Feature: Auth Functionality

  @SignIn
  Scenario: Successful SignIn with valid credentials
    Given ...

  @Registration @Web
  Scenario: Successful user registration
    Given ...
```

Dari kasus ini, Cucumber hanya menjalankan file feature yang di dalamnya memiliki tag `@Checkout` saja baik di level `Feature` maupun level `Scenario` (coba lihat file `Checkout.feature`).

Selain itu, kita juga bisa mengombinasikan `not` dengan `and` atau `or` untuk seleksi yang lebih spesifik. Misalnya, di sini kita akan menjalankan semua skenario kecuali `@Web` atau `@Checkout`:

```
tags = "not (@Web or @Checkout)"
```

Dari tag di atas, Cucumber hanya menjalankan tag-tag selain tag `@Web` atau `@Checkout`. Artinya tag `@Web` dan `@Checkout` tidak akan dijalankan:

```
SignInDefinition:step01()
SignInDefinition:step02()
SignInDefinition:step03()
SignInDefinition:step04()
```

Dalam kasus ini, hanya tag `@SignIn` lah yang dijalankan oleh Cucumber.

CucumberOptions

Saat menggunakan Cucumber untuk automation testing, kita perlu mengonfigurasi bagaimana Cucumber dijalankan. Di sinilah `CucumberOptions` berperan. `CucumberOptions` adalah anotasi di Java yang memungkinkan kita mengontrol berbagai aspek eksekusi Cucumber, seperti lokasi file feature, plugin laporan, tag skenario dan lain sebaginya.

Pada bab ini, kita akan membahas penggunaan `@CucumberOptions`, parameter-parameter yang tersedia, serta contoh penerapannya dalam proyek pengujian otomatisasi.

Berikut contoh class `RunnerTest` yang diberi anotasi `@CucumberOptions` yang kita miliki saat ini:

```
@CucumberOptions(
  tags = "not (@Web or @Checkout)",
  features = {
    "src/test/resources/features/Authentication.feature",
    "src/test/resources/features/Checkout.feature",
  },
  glue = {
    "com.belajar.cucumber.definitions"
  }
)
public class RunnerTest extends AbstractTestNGCucumberTests { }
```

Parameter features

Parameter `features` digunakan dalam anotasi `@CucumberOptions` untuk menentukan lokasi file **.feature** yang akan dijalankan oleh Cucumber. Di `@CucumberOptions` yang kita punya saat ini, terdapat parameter `features` di dalamnya:

```
@CucumberOptions(
  tags = "not (@Web or @Checkout)",
  features = {
    "src/test/resources/features/Authentication.feature",
    "src/test/resources/features/Checkout.feature",
  },
  glue = {
    "com.belajar.cucumber.definitions"
  }
)
```

Di dalam features terdapat dua file feature di dalamnya, yaitu **Authentication.feature** dan **Checkout.feature**. Cucumber akan menjalankan dua file feature tersebut saat test dijalankan.

Selain itu, kita bisa menunjuk ke folder tempat semua file feature kita disimpan daripada menulis path untuk setiap file feature satu per satu:

```
@CucumberOptions(
  ...
  features = "src/test/resources/features",
  ...
)
public class RunnerTest extends AbstractTestNGCucumberTests { }
```

Dengan begitu, kita tidak perlu menyebutkan setiap file feature jika kita menambahkan file feature baru ke dalam folder **features**. Dengan menggunakan folder path seperti di atas, Cucumber akan secara otomatis mencari semua file dengan ekstensi **.feature** di dalam folder **features** dan subfoldernya.

Jika file **feature** ada di beberapa folder, kita bisa menulisnya seperti berikut:

```
@CucumberOptions(
  ...
  features = {
    "src/test/resources/features/UI",
    "src/test/resources/features/API"
  },
  ...
)
```

Dari nilai `features` di atas, Cucumber akan menjalankan semua file **feature** di dalam folder **UI** dan **API** saat kita melakukan test.

Parameter Glue

Parameter `glue` digunakan dalam anotasi `@CucumberOptions` untuk menentukan lokasi package yang berisi step definitions yang akan dijalankan oleh Cucumber. Di `@CucumberOptions` yang kita punya saat ini, terdapat parameter `glue` di dalamnya:

```
@CucumberOptions(
  tags = "not (@Web or @Checkout)",
```

```
  features = {
    "src/test/resources/features/Authentication.feature",
    "src/test/resources/features/Checkout.feature",
  },
  glue = {
    "com.belajar.cucumber.definitions"
  }
)
```

Penulisan pada parameter `glue` juga bisa diganti menjadi lebih sederhana, hanya menggunakan string tanpa harus menggunakan array seperti berikut:

```
@CucumberOptions(
  ...,
  glue = "com.belajar.cucumber.definitions"
)
public class RunnerTest extends AbstractTestNGCucumberTests { }
```

Pada `glue` di atas isinya hanya satu string yang merujuk ke satu package saja. Cucumber akan mencari step definitions dalam package `com.belajar.cucumber.definitions`. Ini berfungsi baik jika hanya ada satu lokasi step definitions.

Namun jika kita menggunakan array seperti:

```
glue = { "com.belajar.cucumber.definitions" }
```

Kita bisa menambahkan lebih dari satu package yang berisi step definition. Misalnya:

```
glue = {
  "com.belajar.cucumber.definitions",
  "com.belajar.cucumber.anotherdefinitions"
}
```

Hal ini memungkikan Cucumber mencari step definition di lebih dari satu package. Sangat berguna jika kita memiliki package tambahan seperti **hooks**, **utilities** atau **shared step definitions**.

Parameter Monochrome

Parameter `monochrome` digunakan dalam Cucumber untuk menentukan apakah output di console akan ditampilkan dengan format yang lebih bersih atau tidak. Contohnya:

```
@CucumberOptions(
  monochrome = true
)
```

Dengan mengatur nilainya ke true, output akan lebih mudah dibaca karena Cucumber menghapus karakter warna ANSI dari console.

Background

`Background` digunakan dalam Cucumber untuk menentukan langkah atau serangkaian langkah yang dibagikan oleh semua pengujian dalam file feature. Ini membantu kita untuk mengurangi redundansi dan membuat file feature lebih bersih dan mudah dibaca.

`Background` memberikan konteks untuk skenario yang ada di bawahnya. `Background` dapat memiliki satu atau beberapa step yang dijalankan sebelum setiap skenario tetapi setelah hook `@Before`.

Background ditempatkan pada tingkat atau level (indentasi) yang sama dengan `Scenario` atau `Example`. Step pada `Background` tidak bisa berisi scenario outline dan example.

Membuat Background

Di dalam folder **features**, buka file **Product.feature** dan tambahkan kode Gherkin berikut ke dalamnya:

```
Feature: Product Management

  Background:
    Given Pengguna sudah masuk ke dalam sistem

  Scenario: Berhasil melihat detail produk
    When Pengguna membuka halaman daftar produk
    And Pengguna memilih produk "Laptop XYZ"
    Then Pengguna melihat detail produk "Laptop XYZ"

  Scenario: Mencoba melihat detail produk yang tidak tersedia
    When Pengguna membuka halaman daftar produk
    And Pengguna memilih produk "Smartphone ABC"
    Then Pengguna melihat pesan error "Produk tidak tersedia"
```

Kemudian, tambahkan class definiton bernama `ProductDetailDefinition` dan tambahkan kode berikut ke dalamnya:

```
package com.belajar.cucumber.definitions;

import io.cucumber.java.en.And;
import io.cucumber.java.en.Given;
import io.cucumber.java.en.Then;
```

```
import io.cucumber.java.en.When;

public class ProductDetailDefinition {

  @Given("Pengguna sudah masuk ke dalam sistem")
  public void userInSystem() {
    System.out.println("Pengguna sudah masuk ke dalam sistem");
  }

  @When("Pengguna membuka halaman daftar produk")
  public void openProductList() {
    System.out.println("Pengguna membuka halaman daftar produk");
  }

  @And("Pengguna memilih produk {string}")
  public void choiceProduct(String title) {
    System.out.println("Pengguna memilih produk '" + title + "'");
  }

  @Then("Pengguna melihat detail produk {string}")
  public void seeDetailProduct(String title) {
    System.out.println("Pengguna melihat detail produk '" + title
+ "'");
  }

  @Then("Pengguna melihat pesan error {string}")
  public void seeError(String message) {
    System.out.println("Pengguna melihat pesan error '" + message
+ "'");
  }

}
```

Buka class `Hook` dari package `.hooks`. Tambahkan kode berikut di dalam method yang mengimpementasikan hook `@Before`:

```
@Before
public void setup() {
  System.out.println("Before");
}
```

Buka class RunnerTest, tambahkan feature Product.feature dan package .hook ke dalamnya:

```
@CucumberOptions(
  features = {
    "src/test/resources/features/Product.feature"
  },
  glue = {
    "com.belajar.cucumber.hooks",
    "com.belajar.cucumber.definitions",
  }
)
```

Jalankan test dan hasil yang tercetak di console adalah sebagai berikut:

```
Before
Pengguna sudah masuk ke dalam sistem
Pengguna membuka halaman daftar produk
Pengguna memilih produk 'Laptop XYZ'
Pengguna melihat detail produk 'Laptop XYZ'
Before
Pengguna sudah masuk ke dalam sistem
Pengguna membuka halaman daftar produk
Pengguna memilih produk 'Smartphone ABC'
Pengguna melihat pesan error 'Produk tidak tersedia'
```

Dari hasil yang kita peroleh, hook `@Before` akan dijalankan terlebih dahulu sebelum `Background`. `Background` pada file `Product.feature` dijalankan untuk setiap skenario di bawahnya.

Ingat, Background yang ada di dalam file **Product.feature** hanya berjalan untuk file feature itu saja. Jika kita memiliki Background lain di dalam file feature lain, scope-nya hanya berjalan di file feature yang berkaitan dengan Background tersebut.

Untuk membuktikannya, coba buka kembali **Checkout.feature** dan ubah kodenya menjadi seperti berikut:

```
Feature: Checkout Process

  Background:
    Given User has items in the cart

  @Checkout
```

```
  Scenario: Successful checkout with valid payment details
    When User proceeds to checkout
    And User enters valid payment details
    And User confirms the order
    Then User should see an order confirmation message
```

Kemudian, ubah method step01 yang ada di dalam class CheckoutDefinition menjadi seperti berikut:

```
@Given("User has items in the cart")
public void step01() {
  System.out.println("User has items in the cart");
}
```

Tambahkan file **Checkout.feature** ke dalam features di `RunnerTest`:

```
features = {
  "src/test/resources/features/Product.feature",
  "src/test/resources/features/Checkout.feature",
}
```

Jalankan test kembali. Hasilnya:

```
Before
User has items in the cart
CheckoutDefinition:step02()
CheckoutDefinition:step03()
CheckoutDefinition:step04()
CheckoutDefinition:step05()
Before
Pengguna sudah masuk ke dalam sistem
Pengguna membuka halaman daftar produk
Pengguna memilih produk 'Laptop XYZ'
Pengguna melihat detail produk 'Laptop XYZ'
Before
Pengguna sudah masuk ke dalam sistem
Pengguna membuka halaman daftar produk
Pengguna memilih produk 'Smartphone ABC'
Pengguna melihat pesan error 'Produk tidak tersedia'
```

Dari hasil yang kita peroleh, `Background` pada file **Chekout.feature** hanya berjalan untuk skenario yang ada di dalamnya. Begitu juga dengan `Background` yang ada di dalam file `Product.feature`, hanya berjalan untuk skenario dari feature product itu saja.

Data Driven Testing

Pada bab ini kita akan mempelajari cara melakukan pengujian di Cucumber berbasis data yang dikenal sebagai Data Driven Testing. Cucumber secara inheren mendukung pengujian berbasis data (Data Driven Testing) dengan menggunakan `Scenario Outline`.

Menerapkan Data Driven Testing

Untuk menerapkan Data Driven Testing, mari kita buka file **Persegi.feature** sebelumnya dari folder **features**. Ubah bagian `Scenario` menjadi `Scenario Testing` dan tambahkan juga beberapa daftar data yang mau kita uji:

```
@SmokeTest
Feature: Persegi dalam Matematika Bangun Datar

  @SanityTest
  Scenario Outline: Hitung luas persegi panjang
    Given saya punya meja dengan panjang <panjang> cm.
    And lebarnya <lebar> cm.
    When saya menghitung luas meja tersebut.
    Then hasilnya harus <hasil>.

    Examples:
      | panjang | lebar | hasil |
      |      12 |    10 |   120 |
      |       0 |    20 |     0 |
      |      15 |     0 |     0 |
      |      11 |    20 |   220 |
```

Di sini kita tidak perlu mengubah kode class definition `PersegiPanjangDefintion` yang sudah kita update di langkah sebelumnya pada bab yang membahas tentang Hook.

Setelah itu, ubah kode pada class RunnerTest menjadi seperti berikut:

```
@CucumberOptions(
  features = {
    "src/test/resources/features/Persegi.feature",
  },
  glue = {
    "com.belajar.cucumber.definitions",
  }
```

```
)
public class RunnerTest extends AbstractTestNGCucumberTests { }
```

Jalankan test dengan perintah `mvn test`, hasilnya akan tercetak seperti berikut:

```
[INFO] Tests run: 4, Failures: 0, Errors: 0, Skipped: 0
[INFO]
[INFO]
------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO]
------------------------------------------------------------------------
```

Dari hasil yang kita peroleh, meskipun kita hanya menjalankan satu class definition saja (`PersegiPanjangDefinition`), yang dijalankan oleh Cucumber adalah 4 kali test karena ada empat data contoh yang kita berikan di file **Persegi.feature**:

```
Examples:
  | panjang | lebar | hasil |
  |      12 |    10 |   120 |
  |       0 |    20 |     0 |
  |      15 |     0 |     0 |
  |      11 |    20 |   220 |
```

Data String

Jika kita ingin menggunakan `Scenario Outline` dengan data string di bagian `Examples`, format di Gherkin harus menggunakan tanda kutip `"` agar Cucumber mengenalinya sebagai string. Berikut contoh `Scenario Outline` dengan data string pada file feature:

```
Feature: Login ke Aplikasi

  Scenario Outline: Login dengan berbagai kombinasi
    email dan password

    Given pengguna memasukkan email "<email>"
    And pengguna memasukkan password "<password>"
    When pengguna menekan tombol login
    Then sistem harus menampilkan "<pesan>"
```

```
    Examples:
      | email           | password | pesan             |
      | "a@example.com" | "123"    | "berhasil"        |
      | "a@example.com" | ""       | "Password kosong" |
      | ""              | "123"    | "Email kosong"    |
      | "invalid.email" | "123"    | "Email invalid"   |
```

Untuk penerapannya di step definition terlihat seperti berikut:

```
public class LoginStepDefinitions {

  ...

  @Given("pengguna memasukkan email {string}")
  public void inputEmail(String email) {
    ...
  }

  @Given("pengguna memasukkan password {string}")
  public void inputPassword(String password) {
    ...
  }

  @When("pengguna menekan tombol login")
  public void clickLogin() {
    ...
  }

  @Then("sistem harus menampilkan {string}")
  public void show(String expectedMessage) {
    ...
  }
}
```

Bagaimana? Mudah bukan!

Laporan Pengujian

Cucumber menyediakan berbagai opsi untuk menghasilkan laporan hasil pengujian. Salah satu yang paling dasar dan sering digunakan adalah plugin `pretty`, yang memberikan output lebih rapi dan mudah dibaca di console.

Menggunakan Pretty

Cara menggunakan `pretty` cukup mudah. Tambahkan saja opsi atau parameter `plugin = "pretty"` dalam `@CucumberOptions` di class `TestRunner`:

```
@CucumberOptions(
  features = {
    "src/test/resources/features/Persegi.feature",
  },
  glue = {
    "com.belajar.cucumber.definitions",
  },
  plugin = { "pretty" }
)
public class RunnerTest extends AbstractTestNGCucumberTests { }
```

Jika kita jalankan test, hasil yang tercetak akan terlihat seperti berikut:

```
@SmokeTest @SanityTest
Scenario Outline: Hitung luas persegi panjang # src/test/resources/features/Persegi.feature:13
  Given saya punya meja dengan panjang 12 cm. # com.belajar.cucumber.definitions.PersegiPanjangDefintion.punyaPanjang(java.lang.Integer)
  And lebarnya 10 cm.                         # com.belajar.cucumber.definitions.PersegiPanjangDefintion.punyaLebar(java.lang.Integer)
  When saya menghitung luas meja tersebut.    # com.belajar.cucumber.definitions.PersegiPanjangDefintion.calculate()
  Then hasilnya harus 120.                    # com.belajar.cucumber.definitions.PersegiPanjangDefintion.resultMustBe(java.lang.Integer)

@SmokeTest @SanityTest
Scenario Outline: Hitung luas persegi panjang # src/test/resources/features/Persegi.feature:14
  Given saya punya meja dengan panjang 0 cm.  # com.belajar.cucumber.definitions.PersegiPanjangDefintion.punyaPanjang(java.lang.Integer)
  And lebarnya 20 cm.                         # com.belajar.cucumber.definitions.PersegiPanjangDefintion.punyaLebar(java.lang.Integer)
  When saya menghitung luas meja tersebut.    # com.belajar.cucumber.definitions.PersegiPanjangDefintion.calculate()
  Then hasilnya harus 0.                      # com.belajar.cucumber.definitions.PersegiPanjangDefintion.resultMustBe(java.lang.Integer)

@SmokeTest @SanityTest
Scenario Outline: Hitung luas persegi panjang # src/test/resources/features/Persegi.feature:15
  Given saya punya meja dengan panjang 15 cm. # com.belajar.cucumber.definitions.PersegiPanjangDefintion.punyaPanjang(java.lang.Integer)
  And lebarnya 0 cm.                          # com.belajar.cucumber.definitions.PersegiPanjangDefintion.punyaLebar(java.lang.Integer)
  When saya menghitung luas meja tersebut.    # com.belajar.cucumber.definitions.PersegiPanjangDefintion.calculate()
  Then hasilnya harus 0.                      # com.belajar.cucumber.definitions.PersegiPanjangDefintion.resultMustBe(java.lang.Integer)

@SmokeTest @SanityTest
Scenario Outline: Hitung luas persegi panjang # src/test/resources/features/Persegi.feature:16
  Given saya punya meja dengan panjang 11 cm. # com.belajar.cucumber.definitions.PersegiPanjangDefintion.punyaPanjang(java.lang.Integer)
  And lebarnya 20 cm.                         # com.belajar.cucumber.definitions.PersegiPanjangDefintion.punyaLebar(java.lang.Integer)
  When saya menghitung luas meja tersebut.    # com.belajar.cucumber.definitions.PersegiPanjangDefintion.calculate()
  Then hasilnya harus 220.                    # com.belajar.cucumber.definitions.PersegiPanjangDefintion.resultMustBe(java.lang.Integer)
[INFO] Tests run: 4, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.912 s -- in TestSuite
[INFO]
[INFO] Results:
```

Laporan dengan Format HTML

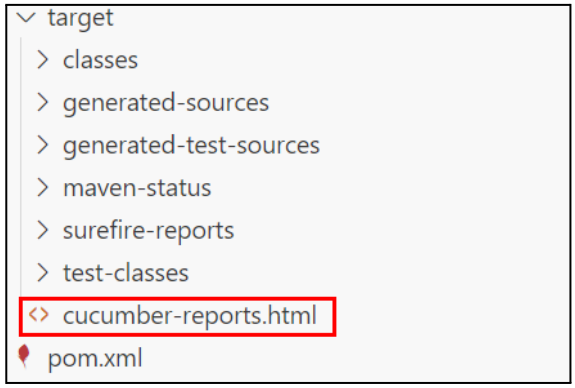
Di Cucumber kita juga bisa membuat laporan dasar dengan format HTML sehingga hasil pengujian dapat dilihat dalam tampilan yang lebih rapi dan mudah dipahami di browser.

Laporan HTML ini secara otomatis dihasilkan oleh Cucumber dengan menambahkan opsi `html:target/cucumber-reports.html` dalam `@CucumberOptions` pada class `TestRunner`:

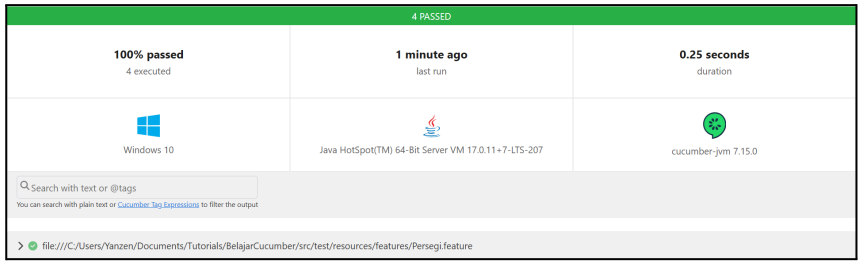
```
@CucumberOptions(
  ...
  plugin = { "pretty", "html:target/cucumber-reports.html" }
)
public class RunnerTest extends AbstractTestNGCucumberTests { }
```

`html` adalah format yang akan kita gunakan untuk membuat file laporan dan `target` adalah folder tempat Cucumber menyimpan file reporting bernama `cucumber-reports.html`.

Saat kita menjalankan test kembali, file **cucumber-reports.html** akan dihasilkan Cucumber di dalam folder **target**:



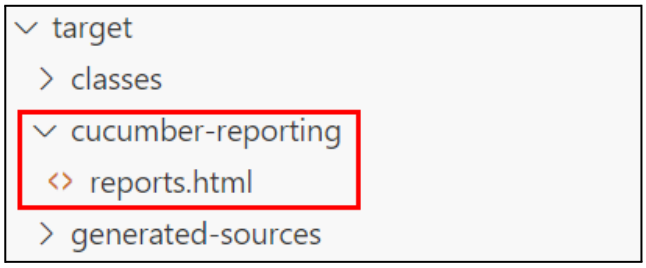
Buka file tersebut di browser, hasilnya akan terlihat seperti gambar berikut:



Kita juga bisa mengubah nama laporan dan menyimpannya di dalam folder yang kita inginkan pada folder target. Misalnya, jika kita ingin mengubah nama laporan menjadi **reports.html** dan menyimpan laporan tersebut di dalam folder **cucumber-reporting**, kita bisa menulisnya seperti ini:

```
plugin = {
  "pretty",
  "html:target/cucumber-reporting/reports.html"
}
```

Jalankan test kembali. Cucumber akan membuatkan kita folder **cucumber-reporting** di dalam **target**. Cucumber juga akan membuat file laporan bernama **reports.html** di dalam folder **cucumber-reporting** tersebut:

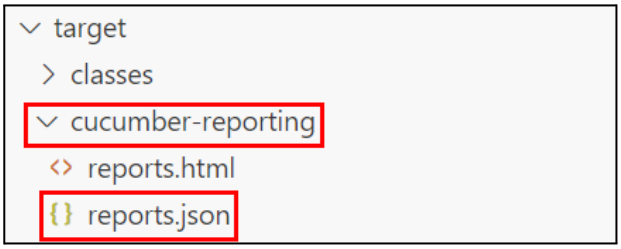


Laporan dengan Format JSON

Selain menggunakan format HTML, Cucumber juga mendukung laporan dengan format JSON. Caranya cukup mudah, kita bisa menambahkan kode laporan berikut di dalam `plugin`:

```
plugin = {
  "pretty",
  "html:target/cucumber-reporting/reports.html",
  "json:target/cucumber-reporting/reports.json",
}
```

Setelah test selesai dijalankan, Cucumber akan membuat file laporan **reports.json** di dalam folder **target/cucumber-reporting**:



Jika kita buka, hasilnya akan terlihat seperti berikut:

Laporan ini berisi data hasil eksekusi tes dalam format terstruktur yang bisa digunakan untuk berbagai keperluan, seperti analisis lebih lanjut, integrasi dengan tools lain atau pembuatan laporan yang lebih kaya fitur.

Integrasi ExtentReports

ExtentReports adalah library populer untuk membuat laporan pengujian yang sangat kaya dan interaktif. Dengan ExtentsReports, kita dapat menghasilkan laporan yang mencakup grafik, tabel dan informasi yang membantu untuk menganalisis hasil pengujian secara lebih rinci. Laporan yang dihasilkan sangat berguna, terutama dalam CI/CD pipeline untuk memberikan visualisasi yang jelas tentang status pengujian.

Menambahkan Dependensi

Untuk membuat laporan dengan ExtentReports, kita perlu menambahkan dua dependensi berikut di dalam pom.xml:

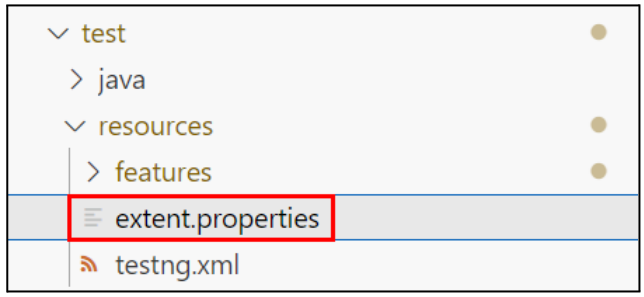
```
<dependency>
  <groupId>com.aventstack</groupId>
  <artifactId>extentreports</artifactId>
  <version>5.1.1</version>
  <scope>compile</scope>
</dependency>
<dependency>
  <groupId>tech.grasshopper</groupId>
  <artifactId>extentreports-cucumber7-adapter</artifactId>
  <version>1.14.0</version>
</dependency>
```

Pada dependensi di atas, kita menggunakan extent report adapter untuk Cucumber 7. Hal ini disebabkan karena kita menggunakan Cucumber di versi 7.

Setelah selesai menambahkan dua dependensi di atas, sinkronisasi perubahan yang baru saja kita lakukan di **pom.xml**.

Membuat extent.properties

Langkah selanjutnya, kita perlu membuat file **extent.properties** di folder **resources** agar adapter extend report yang sudah kita instal bisa mengenalinya:



Dengan menggunakan file properti untuk laporan pengujian, kita dapat menentukan beberapa properti-properti berbeda.

Selanjutnya, aktifkan laporan spark, html dan pdf di dalam file **extent.properties** seperti berikut:

```
extent.reporter.spark.start=true
extent.reporter.spark.out=Reports/Spark.html

#FolderName
basefolder.name=target/ExtentReports/LaporanPengujian
basefolder.datetimepattern=DD-MMM-YYYY HH-mm-ss

#PDF Report
extent.reporter.pdf.start=true
extent.reporter.pdf.out=PdfReport/ExtentPdf.pdf

#HTML Report
extent.reporter.html.start=true
extent.reporter.html.out=HtmlReport/ExtentHtml.html

#System Info
systeminfo.os=windows
systeminfo.version=10
```

Menambahkan Plugin

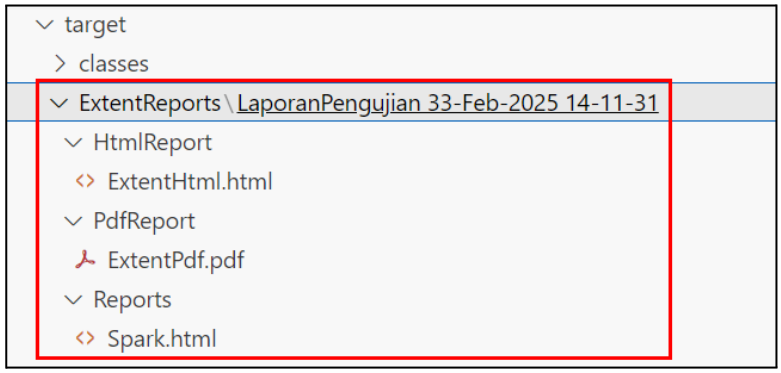
Setelah selesai menambahkan properti-properti untuk membuat laporan di dalam file extent.properties, tambahkan laporan **ExtentReports** di dalam `plugin` pada `@CucumberOptions` di class `RunnerTest` seperti berikut:

```
@CucumberOptions(
  features = {
```

```
    "src/test/resources/features/Persegi.feature",
  },
  glue = {
    "com.belajar.cucumber.definitions",
  },
  plugin = {
    "pretty",
    "html:target/cucumber-reporting/reports.html",
    "json:target/cucumber-reporting/reports.json",

"com.aventstack.extentreports.cucumber.adapter.ExtentCucumberAdapt
er:",
  }
)
public class RunnerTest extends AbstractTestNGCucumberTests { }
```

Menjalankan Test

Jalankan test dengan perintah `mvn test`. Saat test selesai dijalankan, laporan dari ExtentReports akan muncul di dalam folder **target**:



Hal ini disebabkan karena kita mengatur laporan yang dihasilkan berada di dalam folder **target** pada konfigurasi properti berikut:

```
basefolder.name=target/ExtentReports/LaporanPengujian
```

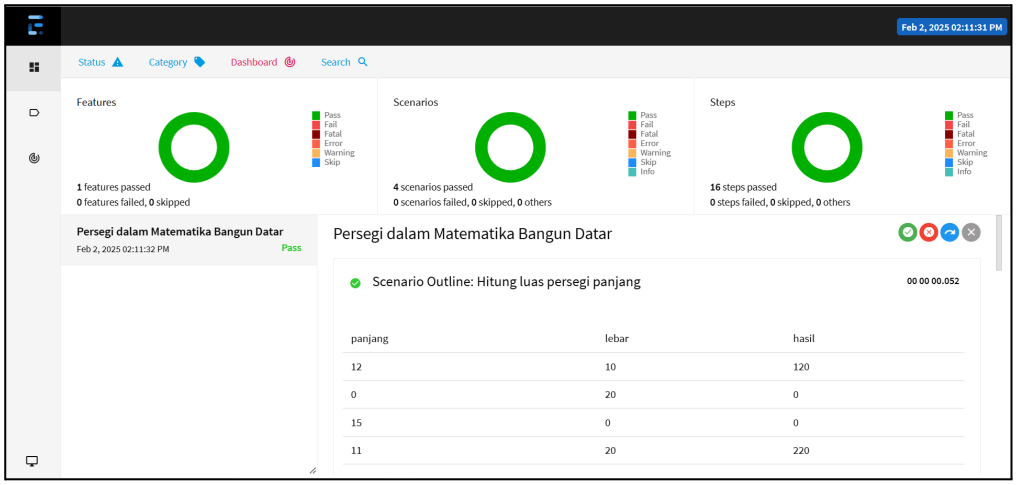
Bentuk nama folder yang dihasilkan oleh ExtentReports mengikuti konfigurasi yang juga kita tambahkan di dalam file **extent.properties**:

```
basefolder.datetimepattern=DD-MMM-YYYY HH-mm-ss
```

Di sini kita membuat tiga jenis laporan sekaligus. Laporan HTML yang lebih lengkap ada di folder **HtmlReport**:

```
extent.reporter.html.out=HtmlReport/ExtentHtml.html
```

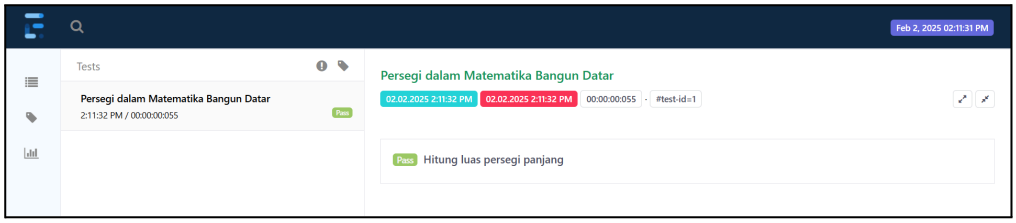
Jika file **ExtentHtml.html** dibuka, tampilannya akan terlihat seperti gambar berikut:



Laporan HTML yang sederhana ada di dalam folder **Reports**:

```
extent.reporter.spark.out=Reports/Spark.html
```

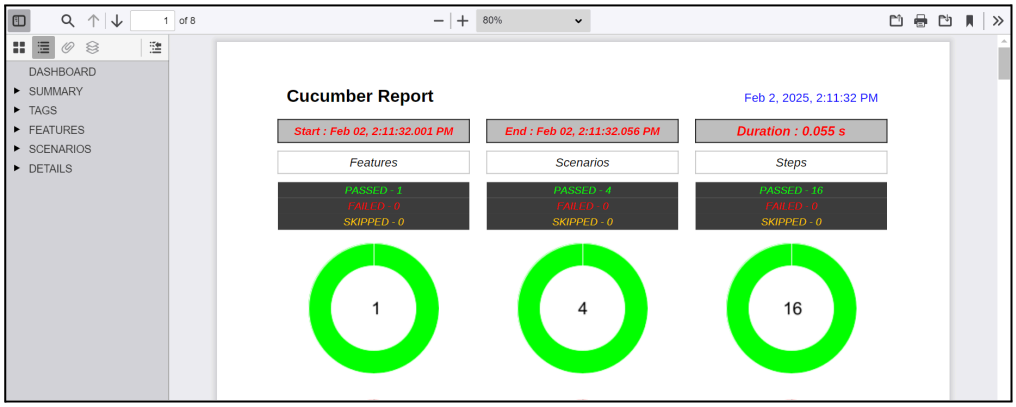
Jika file Spark.html dibuka, hasilnya akan terlihat seperti gambar berikut:



Laporan jenis PDF ada di folder **PdfReport**:

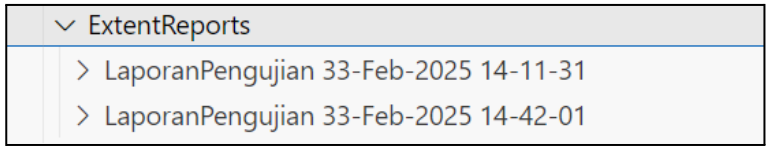
```
extent.reporter.pdf.out=PdfReport/ExtentPdf.pdf
```

Jika file **ExtentPdf.pdf** dibuka, hasilnya akan terlihat seperti gambar berikut:

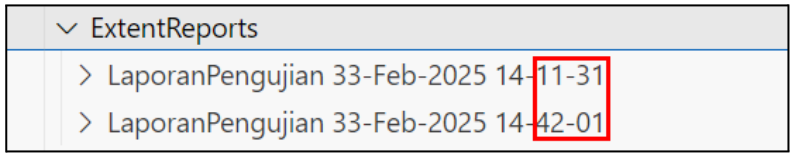


Laporan-laporan yang dihasilkan dicatat dengan baik oleh ExtentReports. Dengan begitu kita bisa menggunakannya untuk menganalisis hasil pengujian lebih mendalam.

Jika kita menjalankan test dua kali dengan perintah `mvn test`, ExtentReports akan menambahkan hasil laporan di folder kedua dengan tanggal atau waktu berbeda:



Dikarenakan kita menjalankan test di tanggal yang sama, ExtentReports menghasilkan folder laporan dengan waktu yang berbeda:



Jika kita ingin mengulang laporan dari awal, kita bisa menjalankan perintah `mvn clean test`. Perintah ini akan menghapus isi folder target sebelumnya dan menghasilkan satu folder laporan yang baru.

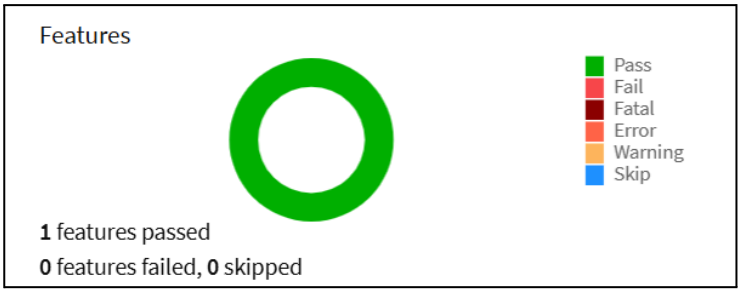
Cara Membaca Laporan HTML

Di bab sebelumnya kita sudah belajar bagaimana membuat laporan yang bagus dengan bantuan ExtentReports. Laporan yang dihasilkan ada beberapa jenis. Di bab ini kita akan belajar bagaimana membaca laporan hasil pengujian yang dihasilkan oleh ExtentReports khususnya laporan HTML yang lengkap, bukan laporan HTML sederhana (spark).

Sebelum lebih jauh, pertama-tama, buka terlebih dahulu laporan **ExtentHtml.html** dari folder **HtmlReport** di browser.

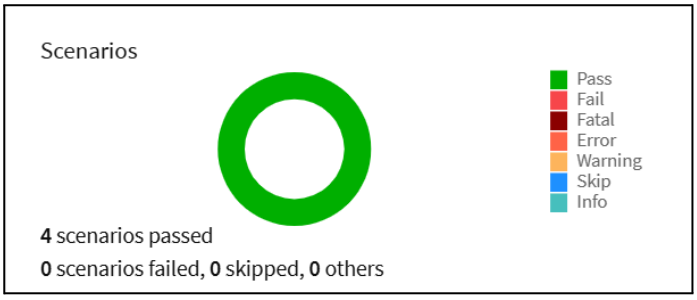
Laporan di Halaman Utama

Di halaman utama, terdapat 3 buah chart. Chart pertama di paling kiri mencantumkan fitur-fitur yang diuji. Dalam contoh ini, ada satu fitur yang diuji, yaitu **"Persegi dalam Matematika Bangun Datar"**:

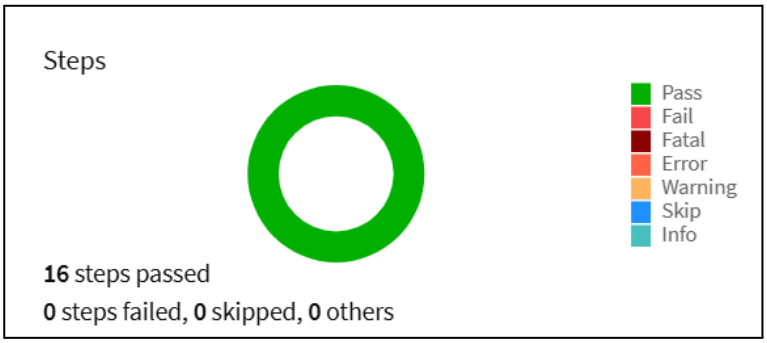


Warna hijau menunjukkan lulus, merah pertama menunjukkan fail, merah tua menunjukkan fatal, merah ketiga menunjukkan error, kuning menunjukkan peringatan dan biru menunjukkan dilewati (skip). Dalam contoh ini, fiturnya **"Pass"** (Lulus) berwarna hijau.

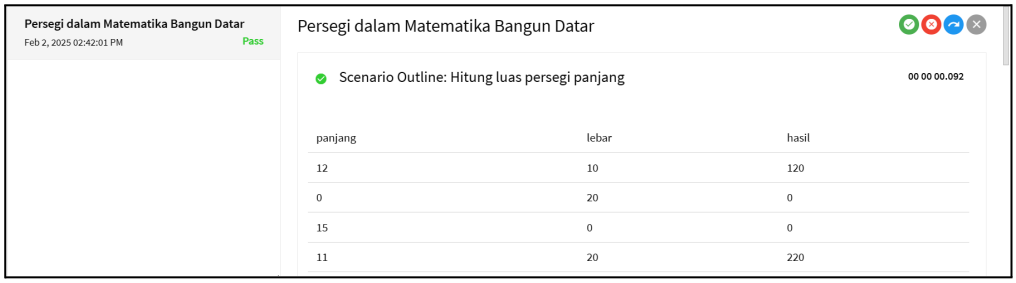
Pada chart tengah mencantumkan skenario-skenario pengujian yang dilakukan dalam setiap fitur. Dalam contoh ini, ada empat skenario yang diuji. Sama seperti bagian Features, chart ini menampilkan persentase skenario yang lulus, gagal (fail, fatal dan error), peringatan dan dilewati (skip):



Selanjutnya di bagian chart sebelah kanan merinci langkah-langkah pengujian yang dilakukan dalam setiap skenario. Dalam contoh ini, ada **16** langkah yang diuji. Sama seperti sebelumnya, grafik ini menampilkan persentase langkah yang lulus, gagal, peringatan dan dilewati.



Di bawah dari ketiga chart terdapat bagian yang berisi rincian skenario. Bagian ini memberikan rincian lebih lanjut mengenai setiap skenario pengujian.



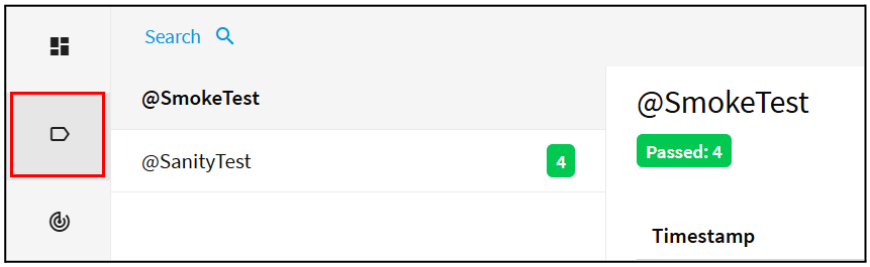
"Scenario Outline: Hitung luas persegi panjang" adalah nama skenario yang diuji. Di sebalah kanan ada sebuah tabel. Tabel ini menunjukkan data masukan (**panjang** dan **lebar**) dan hasil yang diharapkan (**hasil**) untuk setiap skenario. **"00 00 00.052"** menunjukkan waktu yang dibutuhkan untuk menjalankan skenario ini.

Dari laporan ini, dapat disimpulkan bahwa:

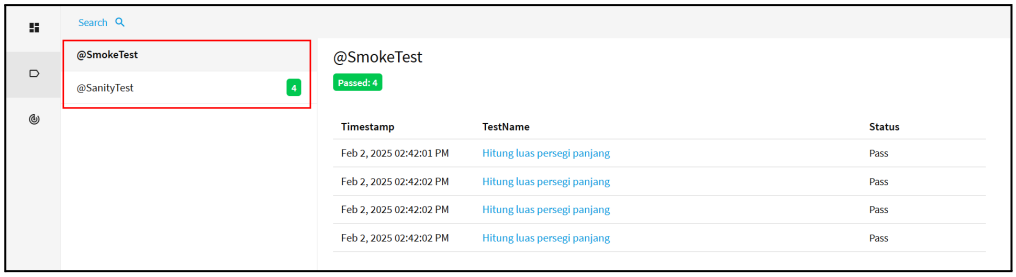
- Satu feature, yaitu "Persegi dalam Matematika Bangun Datar", telah diuji dan lulus.
- Empat skenario pengujian telah dilakukan, dan semuanya lulus.
- Enam belas langkah pengujian telah dilakukan, dan semuanya lulus.
- Rincian data masukan dan hasil untuk setiap skenario dapat dilihat dalam tabel.

Laporan Berdasarkan Tag

Di sebelah kiri terdapa menu hasil pengujian berdasarkan tag-tag yang kita tambahkan di proyek pengujian kita:



Di panel sebelah kiri berisi daftar kategori pengujian (`@SmokeTest` dan `@SanityTest`):

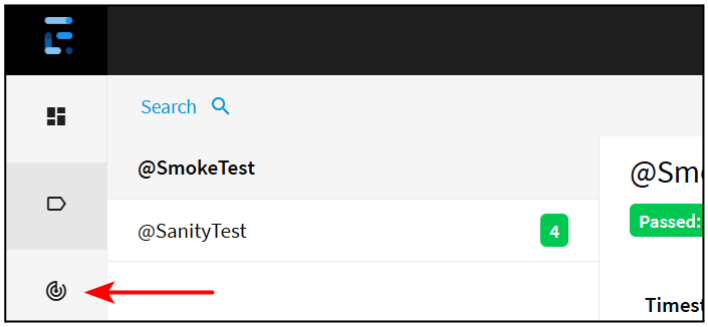


Di sini, tag `@SmokeTesting` sedang dibuka sehingga panel sebelah kanan akan memberikan informasi terkait tag tersebut berupa jumlah pengujian yang lusus (**Passed: 4**), **Timestamp** (waktu pelasanaan pengujian), **TestName** (nama test) dan **status**:

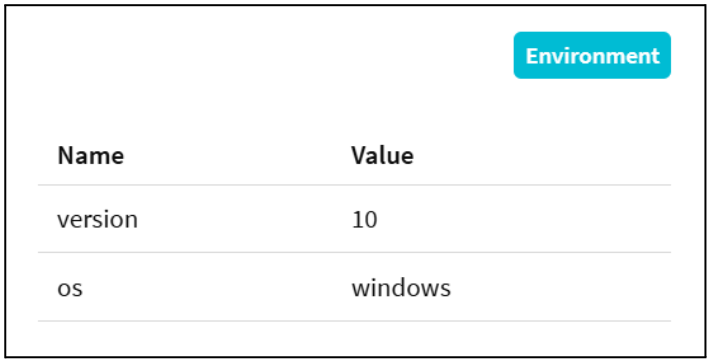
Kamu bisa membuka bagian `@SanityTest` untuk melihat informasi hasil pengujiannya.

Laporan Summary di Dashboard

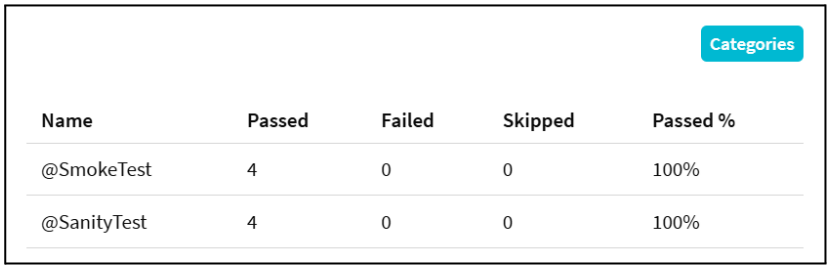
Coba klik menu di sebelah kiri di paling bawah:



Di sini kita akan di bawa ke halaman Dashboard yang berisi summary pengujian kita. Di halaman ini terdapat Environment yang isinya sesuai dengan yang kita atur di dalam extent.properties sebelumnya:



Selain itu terdapat juga informasi berupa tag-tag yang dianggap sebagai kategori:



Dari informasi kategori tersebut, terdapat dua tag yang lulus 100% dalam pengujian.

Di halaman Dashboard juga berisi informasi tentang waktu mulai, waktu selesai dan durasi yang diperlukan untuk suatu proses atau pengujian:

Start	End	Time Taken
Feb 2, 2025 02:42:00 PM	Feb 2, 2025 02:42:02 PM	00 00 01.226

Kesimpulan dari informasi waktu pengujian tersebut menunjukkan bahwa pengujian yang kita lakukan **menghabiskan waktu 1 detik 226 milidetik**. Informasi waktu pengujian tersebut memberikan kita gambaran tentang dua hal penting, yaitu:

- **Efisiensi Waktu:** menunjukkan berapa lama waktu yang dibutuhkan untuk menjalankan pengujian atau proses tertentu.
- **Kinerja Sistem:** memberikan indikasi tentang sebarapa cepat sistem (aplikasi) merespons dan menyelesaikan tugas.