* Este un framework pentru testing care se bazeaza pe a crea teste pe baza la BDD
* BDD - Behaviour Driven Development
* BDD- strategie de a implementa scenarii de teste acceptabile cat produsul este inca in dezvoltare dezvoltand scripturi scrise din perspectiva la customer si developer, asa incat poate fi inteles usor de oricine.
* Features files sunt punctul de pornire cand lucram cu Cucumber tests
* Un feature file este un fisier care contine difertie scenarii listate in diferiti pasi
* Fiecare pas e executat unul dupa altul in Cucumber testing pentru a mima cum end-user interactioneaza cu sistemul.



* Scenariile in Cucumber le putem scrie cu Gherkin, un line oriented language

**Gherkin**

* Gherkin este un line oriented language.
* Orice linie noua incepe deobicei cu un Gherkin keyword, urmat de arbitrary text, care este adesea o descriere la keyword
* Tot acest continut se scrie intr-un fisier cu extensia .feature, ca sa fie asa recunoscut de Cucumber
* El se bazeaza pe asa structura:

**Feature**: A short description of the desired functionality

**Scenario**: A business situation

**Given** a precondition

**And** another precondition

**When** an event happens

**And** another event happens too

**Then** a testable outcome is achieved

**And** something else is also completed

* **Feature** - descrie ce face anume acest fisier. Este pentru documentatie
* **Scenario** - un scenario este un test ce permite validarea unei capabilitati a aplicatiei. El descrie un context initial, eventuri care se pot intampla si expectd results create de aceste evenimente
* **Given** - este o preconditie pentru use case. Este pentru a pune sistemul intr-un state definit inainte ca userul sa interactioneze cu aplicatia
* **When** - descrie un event care se intampla in aplicatie. Poate fi o actiune facuta de user sau un eveniment triggered de un system
* **Then** - specificam un expected outcome a testului.
* **And si But** - le folosim cand sunt mai multi pasi de acelasi tip

**Dependente**

<dependency>

<groupId>io.cucumber</groupId>

<artifactId>cucumber-java</artifactId>

<version>6.8.0</version>

<scope>test</scope>

</dependency>

<dependency>

<groupId>io.cucumber</groupId>

<artifactId>cucumber-junit</artifactId>

<version>6.8.0</version>

</dependency>

<dependency>

<groupId>io.cucumber</groupId>

<artifactId>cucumber-spring</artifactId>

<version>6.8.0</version>

</dependency>

**Sinonime**

Scenario = Example

Scenarios = Examples

Scenario Outline = Scenario Template

**Background**

* Background: are rolul de a executa una sau mai multe chestii inainte de orice Feature:
* Cucumber creaza un object nou pentru fiecare Scenario: ,deci ar fi cazuri multe cand ar trebui sa tot scriem unul si acelasi Given: de exemplu, dar background ar face ca acest Given: sa fie executat pentru fiecare Scenario

**Rule:**

* Rule: e pentru a grupa mai multe scenarii.
* Rule scrie o regula la care se supun toate scenariile din el
* In IDE, in raport, apare si acest rule si scenariile lui. Nu exista anotatie pentru el

Feature: Highlander

Rule: There can be only One

Example: Only One -- More than one alive

Given there are 3 ninjas

And there are more than one ninja alive

When 2 ninjas meet, they will fight

Then one ninja dies (but not me)

And there is one ninja less alive

Example: Only One -- One alive

Given there is only 1 ninja alive

Then he (or she) will live forever ;-)

Rule: There can be Two (in some cases)

Example: Two -- Dead and Reborn as Phoenix

...

**Given**

* Are rolul de a oferi niste date deja cunoscute din timp, adica o stare cunoscuta a sistemului
* Are anotatie @Given

**When**

* Arata o actiune, eveniment concret
* E recomandat sa avem un When per scenario
* Nu ia :

**Then**

* Descrie rezultatul la actiune
* Are in el, in Java, un assert, deobicei

**And**

* Ne ajuta sa evitam repetarea de cuvinte
* Gen:

Given

Given

Given

Este echivalent cu:

Given

And

And

* Are anotatie @And

**But**

* Este pentru o negatie, pentru a arata ca rezultatul nu trebuie sa fie anume asta
* Ex:

Scenario: Find user by name

When user use /users/eduard

Then it should get 5 users

But it should not get 0 users

* Are anotatie @But

**\***

* Este o buna inlocuire a lui And
* Asa, totul arata ca o lista si e mai elegant:

Scenario: All done

Given I am out shopping

\* I have eggs

\* I have milk

\* I have butter

When I check my list

Then I don't need anything

Dar s-ar putea de pus si And in loc de \*

**Step Definitios**

* Gherkin scenarios ar fi useless daca nu ar fi traduse in actiuni si anume aici step definitions ne ajuta
* Un step definition este o metoda cu anotatie Java cu un patern atasat al carui scop este de a converti Gherkin steps din plain text in executabil code.
* Cucumber va cauta step definiots care corespund la predefinitele Gherkin steps
* De exemplu, fie un **step**:

Given I have registered a course in Baeldung

* Si un step definition:

@Given("I have registered a course in Baeldung")

public void verifyAccount() {

// method implementation

}

* Cand cucumber citeste pasii dati, el se uita la step definition care au la anotatie un pattrn ce coresunde cu Gherkin text

**Feature File**

* Feature file contine scenarii listate cu diferiti pasi

Feature: Testing a REST API

Users should be able to submit GET and POST requests to a web service,

represented by WireMock

Scenario: Data Upload to a web service

When users upload data on a project

Then the server should handle it and return a success status

Scenario: Data retrieval from a web service

When users want to get information on the 'Cucumber' project

Then the requested data is returned

* Salvam acest fisier cu extensia .feature in /resources apoi cream clasa in package principal:

**@RunWith(Cucumber.class)**

**@CucumberOptions(features = "classpath:Feature",**

**glue = "package")**

public class CucumberIntegrationTest {

}

* @CucumberOptions(features="fisier") - punem adresa la folderul cu fisierele .feature
* glue = package la clasee glue
* Ne mai trebuie o clasa, pusa in glue package, ce sa porneasca testele pentru cucumber:

@SpringBootTest  
@CucumberContextConfiguration  
public class SpringBootIntegrationTest {  
}

anume @CucumberContextConfiguration face ca Cucumber sa ruleze testele pe baza la toate anotatiile sale gasite in glue package setat

**Scrierea la step definitions**

@When("users upload data on a project")

public void usersUploadDataOnAProject() throws IOException {

}

* Cand Cucumber se ocupa de steps, el va cauta metodele anotate cu gherkin keywords, ca When -> @When. In aceste anotatii se gasesc step definition, ce pot fi simplu text sau regular expression

@When("users want to get information on the {string} project")

public void usersGetInformationOnAProject(String projectName) throws IOException {

}

* Vedem mai sus ca avem {string}, care se va pune in parametrul metodei. Trebuie pus intre {} si reprezinta tipul la argument.
* Pentru int, vom pune {int}, si {int} va pune automat /d+
* Putem folosi si regular expression:

@When("^users want to get information on the '(.+)' project$")

public void usersGetInformationOnAProject(String projectName) throws IOException {

}

* Ordinea lui when(), then() etc. nu conteaza de loc. Oricum se aleg cele care corespund la expresie.
* **Atentie! Pentru fiecare scenariu, se creaza un obiect nou al clasei, ca si cu @Test in JUnit!**

**Exemplu project**

Avem RestController:

@RestController  
public class Rest {  
 @GetMapping("/name")  
 public String getName(){  
 return "Hello";  
 }  
}

Structura la teste:



DemoApplicationTests:

@RunWith(Cucumber.class)  
@CucumberOptions(features = "classpath:feature",  
glue = "com.example.demo.myGlue")  
class DemoApplicationTests {  
   
}

feature1.feature:

Feature: A test feature  
  
 Scenario: A user should use /name endpoint  
  
 When: User use endpoint /name  
 Then: he should receive Hello

Putem pune si mai multe Scenario: in acelasi fisier

Glue1:

public class Glue1 {  
 String response;  
 RestTemplate restTemplate = new RestTemplate();  
  
 @When("User use endpoint {string}")  
 public void when1(String endpoint){  
 response = restTemplate.getForObject("localhost:3306"+endpoint,String.class);  
 }  
 @Then("he should receive {string}")  
 public void then1(String message){  
 *assertThat*(response).isEqualTo(message);  
 }  
}

SpringBootIntegrationTest

@SpringBootTest(webEnvironment = SpringBootTest.WebEnvironment.*DEFINED\_PORT*)  
@CucumberContextConfiguration  
public class SpringBootIntegrationTest {  
}

**Atentie la (webEnvironment = SpringBootTest.WebEnvironment.*DEFINED\_PORT*)**

**de altfel, serverul nu se va porni. E suficient sa il punem doar in clasa unde avem @CucumberContextConfiguration, si gata.**

**Obtinerea datelor**

* Cand folosim Given, adesea utilizam niste date ce sunt deja cunoscute, ce exista deja

Feature: A test file  
 Scenario: File test  
 Given user1 has password test123  
 And user2 has password test123  
 When A user use endpoint /test  
 Then It should receive user1 and password test123  
 And It should receive user2 and password test123

Acum, daca vom vrea sa le obtinem, vom folosi:

@Given("^(.\*) has password (.\*)$")  
public void getList(String username, String password){  
 System.*out*.println(username);  
 System.*out*.println(password);  
}

Metoda data va fi executata de mai multe ori, de fiecare data cand se gaseste o expresie ce corespunde ei, si va returna de fiecare data username+password din Given.

* Avem si o metoda mai simpla de a reprezenta datele, fara sa repetam de fiecare data AND userX has password passwordXYZ

Feature: A test file  
 Scenario: File test  
 Given We have these users  
 | user1 | test123 |  
 | user2 | test123 |  
 When A user use endpoint /test  
 Then It should receive user1 and password test123  
 And It should receive user2 and password test123

@Given("We have these users")  
public void getList(Map<String,String> users){  
 users.forEach((k,v)->System.*out*.println(k +" "+v));  
}

Aici va merge doar un Map, caci e doar un Given cu mai multe date.

* Daca vrem sa avem List<String>

Given We have these users  
| user1 |  
| user2 |

@Given("We have these users")  
public void getList(List<String> strings){  
 System.*out*.println(strings);  
}

* Daca am folosi de exemplu: | name1 | name2 | name3 | name4 |, am avea problema. In asa caz, va fi nevoie sa speciificam numele la coloane, si acestea vor fi key iar valorile lor vor fi stocate intr-un List<Map<String,String>>, si fiecare map va avea 4 elemente in el, per rand deci, si fiecare nume de coloana va fi un key. Prima linie, cand avem mai mult de 2 elemente, adica mai mult de | e1 | e2 |, mereu are numele coloanelor:

Given We have these users  
| firstName | lastName | password |  
| Mititiuc | Eduard | test123 |  
| Goncear | Alexandru | test123 |

@Given("We have these users")  
public void getList(List<Map<String,String>> users){  
 for(Map<String,String> map : users){  
 System.*out*.println("FirstName: "+map.get("firstName"));  
 System.*out*.println("LastName: "+map.get("lastName"));  
 System.*out*.println("password: "+map.get("password"));  
 }  
}

**Sau am putea sa o facem ca List<List<String>> ca fiecare linie sa fie un List<String>**

**Maparea automata**

* Nu e prea comod sa luam mereu datele ca HashMap si sa cream apoi manual obiecte cu ele
* Ar fi mai usor ca sa punem un parametru deodata de tipul obiectului dorit in metoda cu @Given
* Fie obiectul:

@Getter  
@Setter  
@ToString  
@NoArgsConstructor  
@AllArgsConstructor  
@Builder  
public class User {  
 private String firstName;  
 private String lastName;  
 private Integer age;  
   
}

1. E nevoie sa cream un mapper ce sa faca convertirea din datele Cucucmber in Object, si o facem asa:

@SpringBootTest  
@CucumberContextConfiguration  
public class SpringBootIntegrationTest {  
 @DataTableType  
 public User transformToUser(Map<String,String> map){  
 return new User(  
 map.get("firstName"),  
 map.get("lastName"),  
 Integer.*valueOf*(map.get("age")));  
 }  
}

Nu conteaza numele, important sa fie in **@CucumberContextConfiguration** class si sa aiba anotatia **@DataTableType**

**Atentie! Nu e neaparat String. Putem pune si int. Cucumber va avea grija sa faca conversia.**

2. Asta e tot. Deja putem lua direct User object ca argument:

@Given("We have these users")  
public void getList(List<User> users){  
 for(User user : users){  
 System.*out*.println(user);  
 }  
}

Given We have these users  
| firstName | lastName | age |  
| Mititiuc | Eduard | 21 |  
| Goncear | Alexandru | 20 |

* Daca vom lasa | | gol, vom primi null. Daca vrem un empty string, folosim:

| [anonymous] |

**Maparea la un obiect**

Daca avem o singura linie cu date, putem pur si simplu returna acel Object mapat si gata, fara List<> sau HashMap<>, caci deja se face maparea si asa in metoa definita de mapare in @DataTableType

Scenario:  
 When I receive a user  
 Then it's like that  
 | firstName | lastName | age |  
 | Mititiuc | Eduard | 21 |

**Maparea automata a obiectelor**

Adaugand urmatoarele anotatii,cu un objectMapper, Cucumber va face maparea automata, exact asa cum o face Jackson cu @RequestBody

@DefaultParameterTransformer

@DefaultDataTableEntryTransformer

@DefaultDataTableCellTransformer

@SpringBootTest(webEnvironment = SpringBootTest.WebEnvironment.*DEFINED\_PORT*)  
@CucumberContextConfiguration  
public class SpringBootRunCucumberTests {  
 private final ObjectMapper objectMapper = new ObjectMapper();  
  
 @DefaultParameterTransformer  
 @DefaultDataTableEntryTransformer  
 @DefaultDataTableCellTransformer  
 public Object transformer(Object fromValue, Type toValueType) {  
 return objectMapper.convertValue(fromValue, objectMapper.constructType(toValueType));  
 }  
}

**Scenario Outline si Examples**

* Scenario Outline are rolul de a rula acelasi Scenarios/Examples de mai multe ori
* Intr-un scenario se poate intampla sa avem mai multe date identice pe la When, Then, And etc.
* Nu e prea comod sa dam mereu copy paste la tabelul cu date in .feature
* Scenario Outline permite definirea unui Scenariu ce are un tabel ce poate fi folosit in mai multe keyword.
* Acest tabel e scris dupa **Exemples**:

Scenario Outline: A user accesses the /users/21 endpoint  
 When A user uses /users/21 endpoint  
 Then It should receive this data  
 | firstName | lastName | age |  
 | <firstName> | <lastName> | <age> |  
  
 Examples:  
 | firstName | lastName | age |  
 | Mititiuc | Eduard | 21 |  
 | Goncear | Alex | 21 |

Vedem ca folosim <NumeColoana> si vor fi puse deodata toate valorile din tabelul Examples.

**Background**

Se foloseste pentru a efectua o anumita operatie inainte de fiecare scenario, ca asa sa nu repetam acelasi cod pentru fiecare scenario

Feature: Multiple site support

Only blog owners can post to a blog, except administrators,

who can post to all blogs.

Background:

Given a global administrator named "Greg"

And a blog named "Greg's anti-tax rants"

And a customer named "Dr. Bill"

And a blog named "Expensive Therapy" owned by "Dr. Bill"

Scenario: Dr. Bill posts to his own blog

Given I am logged in as Dr. Bill

When I try to post to "Expensive Therapy"

Then I should see "Your article was published."

Scenario: Dr. Bill tries to post to somebody else's blog, and fails

Given I am logged in as Dr. Bill

When I try to post to "Greg's anti-tax rants"

Then I should see "Hey! That's not your blog!"

Scenario: Greg posts to a client's blog

Given I am logged in as Greg

When I try to post to "Expensive Therapy"

Then I should see "Your article was published."

**Comentarii in multiples lines**

* Folosim """ """"

"""  
It's a comment  
on multiple lines  
"""

**@Before si @After**

* @Before si @After se executa pana la si dupa fiecare scenariu, exact ca in JUnit
* Putem specifica si un parametru order, pentru a specifica ordinea de executie:

**@Before(order = 5)**

* Putem folosi si taguri. Gen, daca folosim vre-un tag la un Scenario: si un tag in @Before sa u@After, ele se vor executa doar pentru scenario ce au acele taguri:

Background: Initialize lists  
 Given Our Lists are not null  
Rule: This is for user  
 @OnlyThis  
Scenario: A user accesses the /users endpoint  
 When A user uses /users endpoint

@Before("@OnlyThis")  
public void before(){  
 System.*out*.println("A statement executed before a tagged scenario");  
}

* Putem folosi si @Before("not @OnlyThis") pentru a excluce features cu asa taguri

**@BeforeAll si @AfterAll**

* Le folosim exact ca in Junit, cu metode statice

**Variabile in Cucumber**

Variabilele defapt si sunt acel tabel creat. Cand cream tabelul in Examples, acolo defapt definim variabile. Dar, daca le folosim doar intre < > se va lua cate una, dar daca intre | < > | se vor lua toate. Gen:

Feature: A second test file  
 Scenario Outline: Test week's days  
 When An user enter the day's number <nr>  
 Then It should receive this day <name>  
  
 Examples:  
 | nr| name |  
 | 1 | Luni |  
 | 2 | Marti |  
 | 3 | Miercuri |  
 | 4 | Joi |  
 | 5 | Vineri |  
 | 6 | Sambata |  
 | 7 | Duminica |

Atunci, pentru a verifica fiecare zi, facem asa:

public class WeekGlue {  
 private Integer number;  
 private String day;  
 @When("^An user enter the day's number (.\*)$")  
 public void setNumber(int number){  
 this.number = number;  
 }  
  
  
 @Then("^It should receive this day (.\*)$")  
 public void setDay(String day){  
 this.day = day;  
 System.*out*.println(number + " " + day);  
 }  
}



si vedem ca deorece variabilele nu au fost puse intre | <> |, ci doar intre <>, s-a luat cate una rand pe rand pentru fiecare scenario.

**Cucumber Expressions**

Cucumber expressions se scriu cu { }

**Atentie! Nu putem folosi CucumberExpression impreuna cu RegEx, gen:**

@When("^I write number and {string}$")

Nu e permis!!!!

* Intre { } se pune tipul de date si putem pune oricate intr-o expresie, gen:

Scenario:  
 When I write 'number' and 'letter'  
 Then Return 'Hello'

@When("I write {string} and {string}")  
public void test(String number, String letter){  
 System.*out*.println(number + " " + letter);  
}  
@Then("Return {string}")  
public void print(String string){  
 System.*out*.println(string);  
}

Atentie! Daca folosim {string}, va trebui sa punem in fisierul .feature stringul intre ' ' sau intre " "

* {} - e pentru orice
* {bigdecimal} - exact ca float, dar returneaza BigDecimal

**Cum verificam { sau (**

Pentru a verifica { sau ( propriu zis ca string, folosim /{ si /(