**BDD Automation Framework**

**Content**

1. **Framework - overview**
   1. Automation testing
   2. Selenium WebDriver automation testing tool
   3. Page Object Model (POM) design pattern principles
   4. Cucumber testing - Behavior Driven Development approach
2. **Environment setup**
   1. Java
   2. Maven
   3. IntelliJ IDEA
   4. Selenium WebDriver
   5. Cucumber
   6. TestNG
   7. Git
   8. Log4J
   9. Rest Assured
   10. Postman
   11. Jenkins
3. **User Acceptance Testing Module**
   1. Framework development
      1. Framework structure
         1. Base package
         2. Config package
         3. Features package
         4. Step definitions package
         5. Pages package
         6. Utils package
         7. Runner package
      2. External libraries
   2. Test execution
      1. Sequential test execution
      2. Parallel test execution
   3. Reporting
      1. Default report
      2. Extent report
      3. Log file
4. **API testing Module**
5. **Continuous Integration, Continuous Delivery, Continuous Deployment** 
   1. Jenkins integration
   2. GitLab/GitHub integration
   3. Jira integration
   4. Slack integration
6. **Framework overview**

**1.1 Automation testing**

Software tests have to be repeated often during development cycles to ensure quality. Every time source code is modified software tests should be repeated. For each release of the software it may be tested on all supported operating systems and hardware configurations. Manually repeating these tests is costly and time consuming. Once created, automated tests can be run over and over again at no additional cost and they are much faster than manual tests.

Automation testing uses the specialized tools to automate the execution of manually designed test cases without any human intervention. Automation testing tools can access the test data, controls the execution of tests and compares the actual result against the expected result. Consequently, generating detailed test reports of the system under test. Automation testing is an automatic technique where the tester writes scripts by own and uses suitable software to test the application.

**Benefits of Automation Testing**

* **Fast:** Runs tests significantly faster than human users.
* **Repeatable:** Testers can test how the website or software reacts after repeated execution of the same operation.
* **Reusable:** Tests can be reused on different versions of the software.
* **Reliable:** Tests perform precisely the same operation each time they are run thereby eliminating human error.
* **Comprehensive:** Testers can build test suites of tests that covers every feature in software software application.
* **Programmable:** Testers can program sophisticated tests that bring hidden information.

You can get the most benefit out of your automated testing efforts by automating:

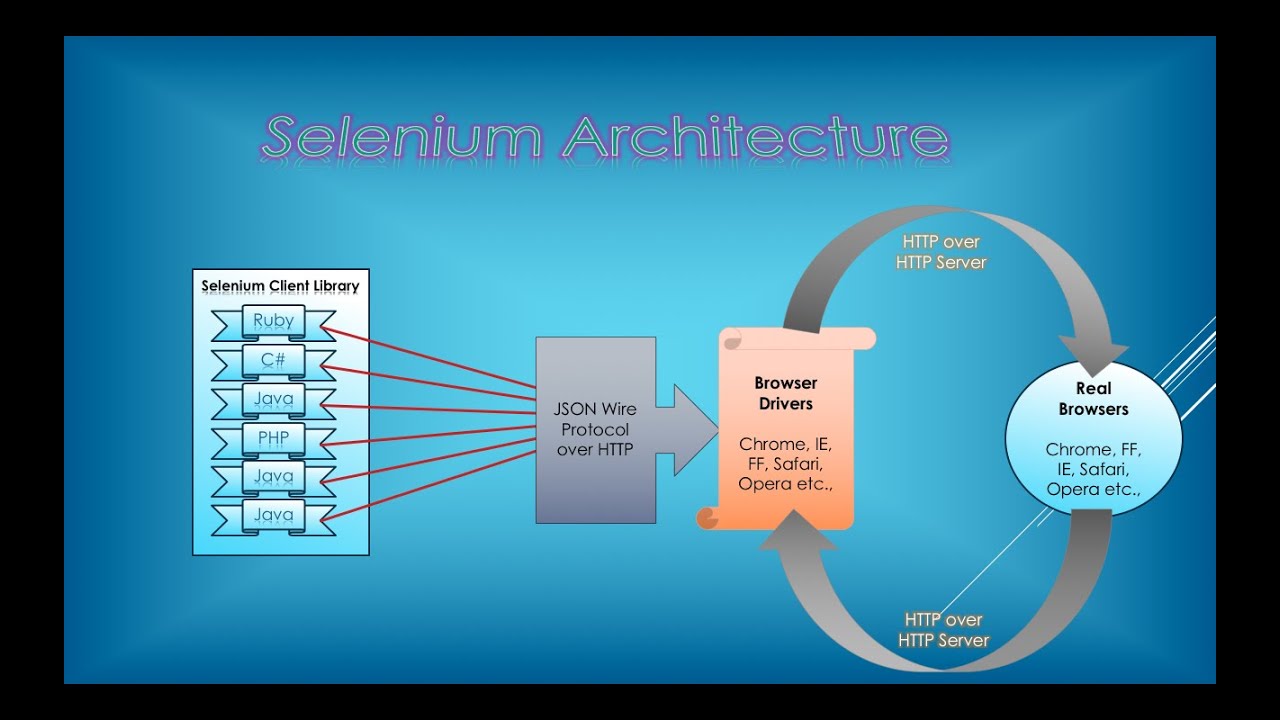
* Repetitive tests that run for multiple builds.
* Tests that tend to cause human error.
* Tests that require multiple data sets.
* Frequently used functionality that introduces high risk conditions.
* Tests that are impossible to perform manually.
* Tests that run on several different hardware or software platforms and configurations.
* Tests that take a lot of effort and time when manual testing.

**1.2 Selenium WebDriver automation testing tool**

Selenium WebDriver is an open source tool which is used for automating the tests carried out on web browsers. Since Selenium is open-source, there is no licensing cost involved, which is a major advantage over other testing tools. Other reasons behind Selenium’s ever growing popularity are:

* Test scripts can be written in any of these programming languages: **Java**, **Python**, **C#**, **PHP**, **Ruby**, **Perl**
* Tests can be carried out in any of these OS**:** **Windows**, **Mac** or **Linux**
* Tests can be carried out using any browser: **Mozilla Firefox**, **Internet Explorer**, **Google Chrome**, **Safari** or **Opera**
* It can be integrated with tools such as **TestNG** & **JUnit** for managing test cases and generating reports
* It can be integrated with **Maven**, **Jenkins** & **Docker** to achieve Continuous Testing

Selenium has a **client-server architecture**, and includes both client and server components.



### **Selenium Language Bindings / Selenium Client Libraries**

Selenium developers have built language bindings/Selenium Client Libraries in order to support multiple languages. For instance, if you want to use the browser driver in java, use the java bindings. All the supported language bindings can be downloaded from the official website [(https://www.seleniumhq.org/download/#client-drivers)](https://www.seleniumhq.org/download/#client-drivers) of Selenium.

### **JSON Wire Protocol**

JSON (JavaScript Object Notation) is an open standard for exchanging data on web. It supports data structures like object and array. So, it is easy to write and read data from JSON. To learn more about JSON, visit <https://www.javatpoint.com/json-tutorial>

JSON Wire Protocol provides a transport mechanism to transfer data between a server and a client. JSON Wire Protocol serves as an industry standard for various REST web services. To learn more about Web Services, visit <https://www.javatpoint.com/web-services-tutorial>

### **Browser Drivers**

Selenium uses drivers, specific to each browser in order to establish a secure connection with the browser without revealing the internal logic of browser's functionality. The browser driver is also specific to the language used for automation such as Java, C#, etc.

When we execute a test script using WebDriver, the following operations are performed internally.

* HTTP request is generated and sent to the browser driver for each Selenium command.
* The driver receives the HTTP request through HTTP server.
* HTTP Server decides all the steps to perform instructions which are executed on browser.
* Execution status is sent back to HTTP Server which is subsequently sent back to automation script.

### **Browsers**

Browsers supported by Selenium WebDriver:

* Internet Explorer
* Mozilla Firefox
* Google Chrome
* Safari

For a better understanding on how Selenium WebDriver works please read [this explanation.](https://www.quora.com/How-does-the-Selenium-WebDriver-work)

**Selenium WebDriver methods**

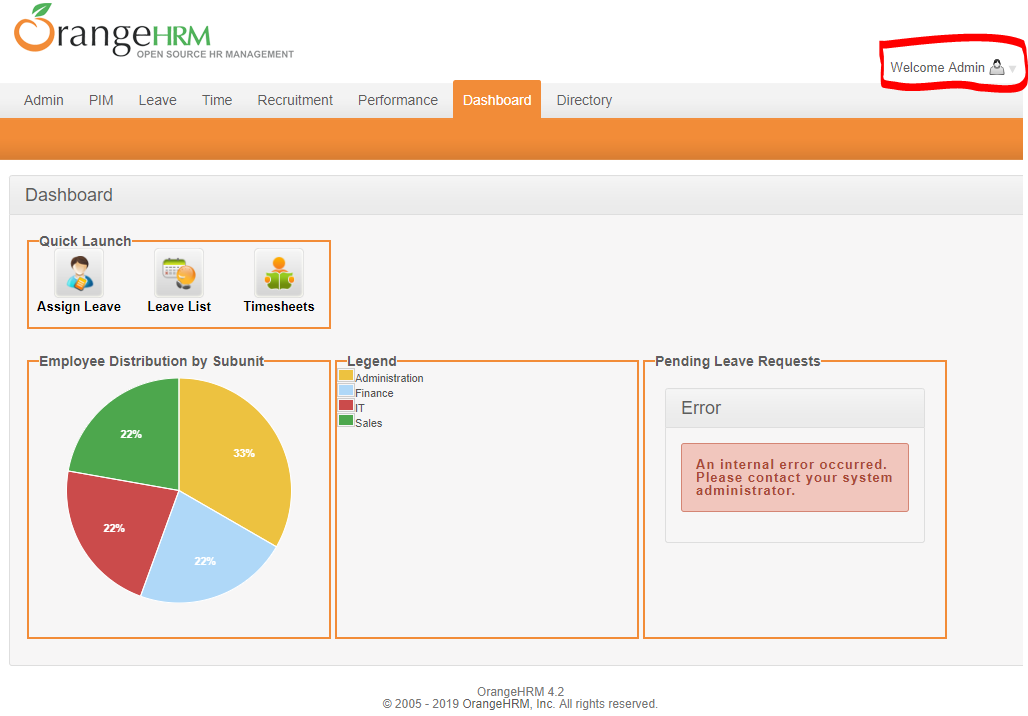
|  |
| --- |
| **1.get ().** It is used to open specified url browser in windows. Syntax: //to launch the browser driver. get(http://google.com); **2.getCurrentUrl().** Its Returns title of the Browser Syntax: //to launch the browser Driver.get(http://google.com); String url=driver.getCurrentUrl(); System.out.println(url); **3.getTitle().** It is used to get the title of current web page Syntax: //to launch the browser driver.get("http://www.google.com"); String title=driver.getTitle(); **4.getPageSource().** It is used to get the source of current load page Syntax: //to launch the browser driver.get("http://www.google.com"); String pagesource=driver.getPageSource(); System.out.println(pagesource); **5.findElement().** It is used to find the first WebElement using the given method. Syntax: //to launch the browser driver.get("http://www.gmail.com"); WebElement gmaillink=driver.findElement(By.id()); System.out.println(gmaillink.getText()); **6.findElements().** It is used to find all elements within the current page Syntax: //to launch the browser driver.get("http://www.facebook.com"); //to findelements List links=driver.findElements(By.TagName("a")); //Counting no of links in result page System.out.println(links.size()); **7.close().** Close the current window, if there are multiple windows, it will close the current window which is active and quits the browser if it is the last window opened currently. Syntax: driver.get("http://www.etestinghub.com"); driver.close(); **8.quit().** It is used to close every associated window which is opened. Syntax: driver.get("http://www.etestinghub.com"); driver.quit(); **9.getWindowHandle().** Whenever the web driver launches the browser it assigns the unique id to that browser which is called as window handler. This can be captured through the method. Syntax: driver.getWindowhandle(). **10.getWindowHandles().** Whenever multiple windows are opened by webdriver and we want to capture all their ids. We use this method. Syntax: getWindowHandles(). **11.switchTo().** Used to switch from one window to another window (or) window to a frame (or) frame to a window (or) window to an alert Syntax: driver.switchTo().window(); driver.switchTo().frame(); driver.switchTo().alert(); **12.navigate().** The driver to access the browser’s history and to navigate to a given URL&Refresh page. Syntax: driver.get("http://gmail.com"); //navigate to page driver.navigate().to("http://estestinghub.com "); //navigate to back driver.navigate().back(); //navigate to forward driver.navigate().forward(); //navigate to refresh page driver.navigate().refresh(); **13. manage().** This is used to perform maximize the size of the window. driver.get("http://gmail.com"); driver.manage().window().maximize(); **14.click()** This is used to click on webelements like link, button, radio group, checkbox, images...etc. **15.sendKeys()** Purpose: This is used to sending inputs into text fields and text areas, and also used to select value from the drop down box. **16.clear()** Purpose: This is used to clear the input from existing data. **17.getText()** Purpose: This is used to capture text of the webElement. **18.getTagName()** Purpose: This is used to capture html tag of the webElement. **19.getLocation()** This is used to capture X and Y coordinates of webelement in the application. **20.isSelected()** This is used to check, is the check-box is currently checked or unchecked to checked Radio buttons are selected or not. **21.isDisplayed()** This is a Boolean condition. It is used to either an element is visible or not. If an element is displayed it gives true and an element is not displayed it gives false. **22.IsEnabled()** This is a Boolean condition. It is used to either an element is enable or not. If an element is enable it gives true and an element is disable it gives false. **23.getAttribute ()** This is used to capture the attributes which are present in web applications |

There are seven basic steps in creating a Selenium test script, which apply to any test case and any application under test (AUT).

1. Create a WebDriver instance.
2. Navigate to a Web page.
3. Locate an HTML element on the Web page.
4. Perform an action on an HTML element.
5. Anticipate the browser response to the action.
6. Run tests and record test results using a test framework.
7. Conclude the test.

Below you can see a basic example of a automated login test.





|  |
| --- |
| import io.github.bonigarcia.wdm.WebDriverManager; import org.junit.Assert; import org.openqa.selenium.By; import org.openqa.selenium.WebDriver; import org.openqa.selenium.chrome.ChromeDriver;  import java.util.concurrent.TimeUnit;  public class Login {   public static void main(String args[]) {   *//setup Chrome driver to interact with real browser*  WebDriverManager.chromedriver().setup();   *//create an instance of ChromeDriver class*  WebDriver driver = new ChromeDriver();   *//wait for a certain amount of time before throwing an exception that it cannot find the element on the page*  driver.manage().timeouts().implicitlyWait(10, TimeUnit.SECONDS);   *//maximize the window*  driver.manage().window().maximize();   *//navigate to url*  driver.get("https://opensource-demo.orangehrmlive.com/");   *//find username text box on the page using id locator and then type "Admin"*  driver.findElement(By.id("txtUsername")).sendKeys("Admin");   *//find password text box on the page using id locator and then type "admin123"*  driver.findElement(By.id("txtPassword")).sendKeys("admin123");   *//find the login button and click on it*  driver.findElement(By.id("btnLogin")).click();  */\*after clicking login button a new page is displayed, "Welcome page". To make sure the login process was successfully accomplished we have to verify it by extracting a piece of text from the new page\*/*  String actualValue = driver.findElement(By.id("welcome")).getText();   *//compare actual outcome with the expected outcome*  Assert.assertEquals("Welcome Admin", actualValue);   *//close the browser*  driver.close();  } } |

**1.3 Page Object Model (POM) design pattern principles**

Page Object Model is a design pattern which has become popular in test automation for enhancing test maintenance and reducing code duplication. Page Object Model helps to create an Object Repository for web UI elements. Under this model, for each web page in the application, there should be corresponding page class. This Page class will find the WebElements of that web page and also contains Page methods which perform operations on those WebElements. The tests then use the methods of this page object class whenever they need to interact with the UI of that page, the benefit is that if the UI changes for the page, the tests themselves don’t need to be changed, only the code within the page object needs to change. Subsequently all changes to support that new UI are located in one place.

**Advantages of Page Object Model:**

* According to Page Object Model, we should keep our tests and element locators separately, this will keep code clean and easy to understand and maintain.
* The Page Object approach makes test automation framework programmer friendly, more durable and comprehensive.
* Another important advantage is our Page Object Repository is Independent of Automation Tests. Keeping separate repository for page objects helps us to use this repository for different purposes with different frameworks like, we are able to integrate this repository with other tools like [JUnit](https://junit.org/)/[NUnit](http://nunit.org/)/[PhpUnit](https://phpunit.de/) as well as with [TestNG](http://testng.org/)/[Cucumber](https://cucumber.io/)/etc.
* Test cases become short and optimized as we are able to reuse page object methods in the POM classes.
* Any change in UI can easily be implemented, updated and maintained into the Page Objects and Classes.

More details about Page Object Model and an implementation can be found at this [link.](https://www.guru99.com/page-object-model-pom-page-factory-in-selenium-ultimate-guide.html)

**1.4 Cucumber testing - Behavior Driven Development approach**

Cucumber is a tool that supports Behavior Driven Development (BDD), allowing users to define application operations in plain text. It works based on the [Gherkin](https://github.com/cucumber/cucumber/wiki/Gherkin) language. This simple but powerful syntax of Gherkin lets developers and testers write complex tests while keeping it comprehensible to even non-technical users. Cucumber reads executable specifications written in plain text and validates that the software does what those specifications say. The specifications consists of multiple examples, or scenarios. For example:

|  |
| --- |
| Feature: Login Action   Scenario: Successful Login with Valid Credentials Given User is on Home Page When User navigates to LogIn Page And User enters UserName and Password And User clicks on Login Button Then A Welcome message is displayed |

Behavior Driven Development - BDD

Behavior Driven Development is a software development approach that has evolved from TDD (Test Driven Development). It differs by being written in a shared language, which improves communication between tech and non-tech teams and stakeholders. In both development approaches, tests are written ahead of the code, but in BDD, tests are more user-focused and based on the system’s behavior.

Gherkin language

In order for Cucumber to understand the scenarios, they must follow some basic syntax rules, called Gherkin. Gherkin is a line-oriented language using line endings, indentations and keywords to define documents. Each non-blank line usually starts with a Gherkin keyword, followed by an arbitrary text, which is usually a description of the keyword. Gherkin is the language that Cucumber uses to define test cases. It is designed to be non-technical and human readable, and collectively describes use cases relating to a software system. The purpose behind Gherkin's syntax is to promote behavior-driven development practices across an entire development team, including business analysts and managers. The main keywords in Gherkin are:

* *Feature* - the purpose of the Feature keyword is to provide a high-level description of a software feature, and to group related scenarios.
* *Scenario* - each Feature will contain some number of tests to test the feature. Each test is called a Scenario and is described using the Scenario keyword.
* *Given* - defines a precondition to the test.
* *When* - defines the test action that will be executed.
* *Then* - defines the outcome of previous steps
* *And* - keyword is used to add conditions to your steps.
* *But* - keyword is used to add negative type comments
* *Background* - is used to define steps which are common to all the tests in the feature file.
* *Scenario outline* - used together with *Examples* keyword to run the same test with multiple sets of data
* *Examples* - used together with *Scenario Outline* keyword to run the same test with multiple sets of data

More about Cucumber, BDD or Gherkin you can find [here.](https://docs.google.com/document/d/1WFhplQ2BT56gxUu1YsriiwygvfDvkKNOrIxmwVVH7CA/edit?usp=sharing)

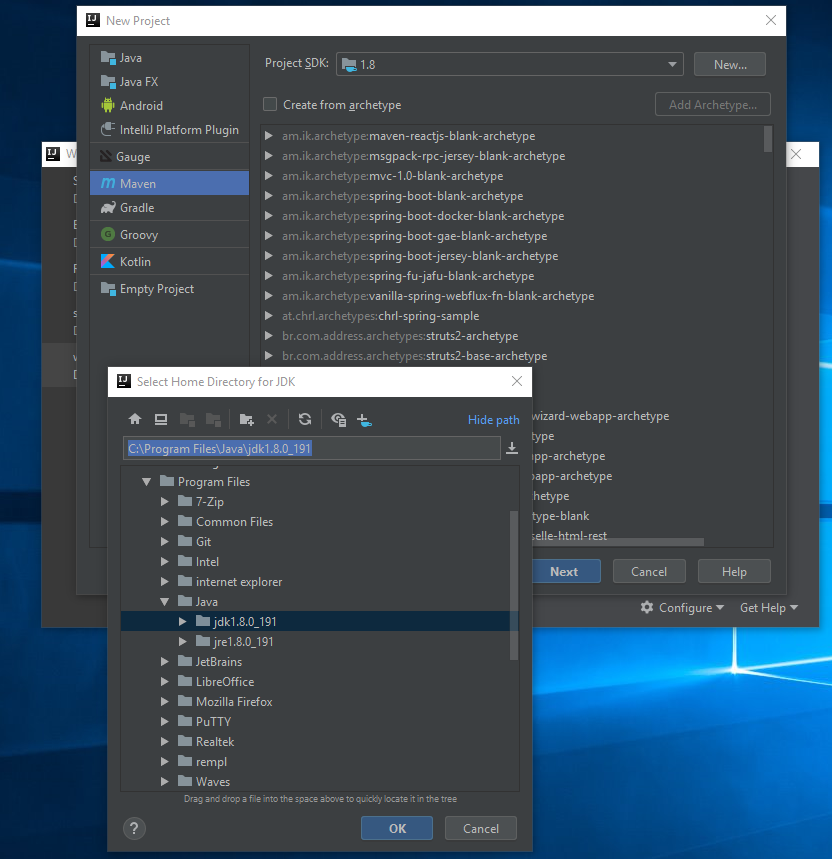
1. **Environment setup**

1. **User Acceptance Testing Module**

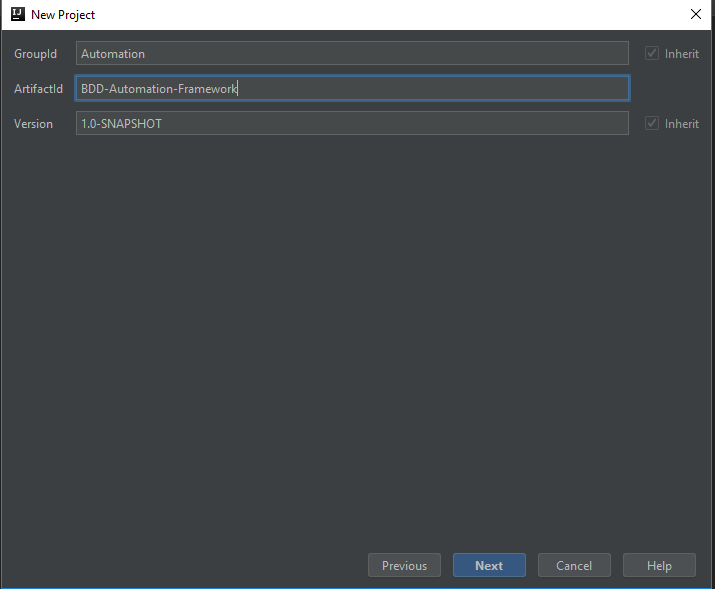
**3.1 Framework development**

In this project we will use an e-commerce site, [www.emag.ro](http://www.emag.ro) and we will automate the login and search tests. So, in order to perform the login test it is required to create an account on [www.emag.ro](http://www.emag.ro) .

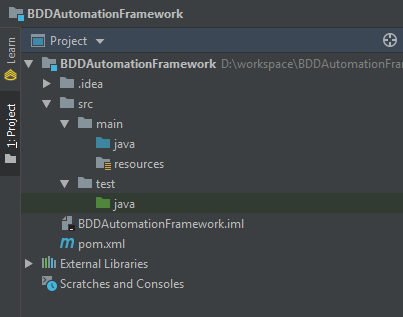
Start IntelliJ IDEA and click on *Create New Project* . On the left bar select *Maven* and then link “*Project SDK*” to your JDK, e.g.: on Windows “C:\Program Files\Java\jdkxxx” by clicking on *New…* .



Then specify a *GroupId* and *ArtifactId* (you can give any name), click *Next*  and then *Finish.*

**

You should now have a project with the following structure:

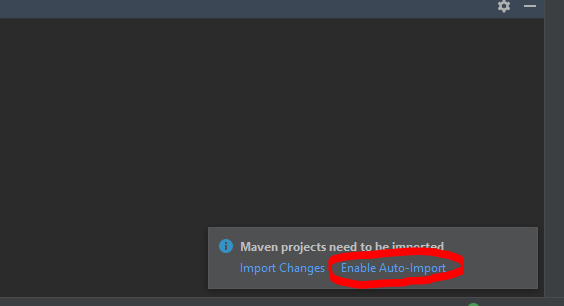


For simplicity, delete the folder called “*test*” because we won’t need it and open *pom.xml* . Using *pom.xml*(Maven) you can configure dependencies needed for building the project.

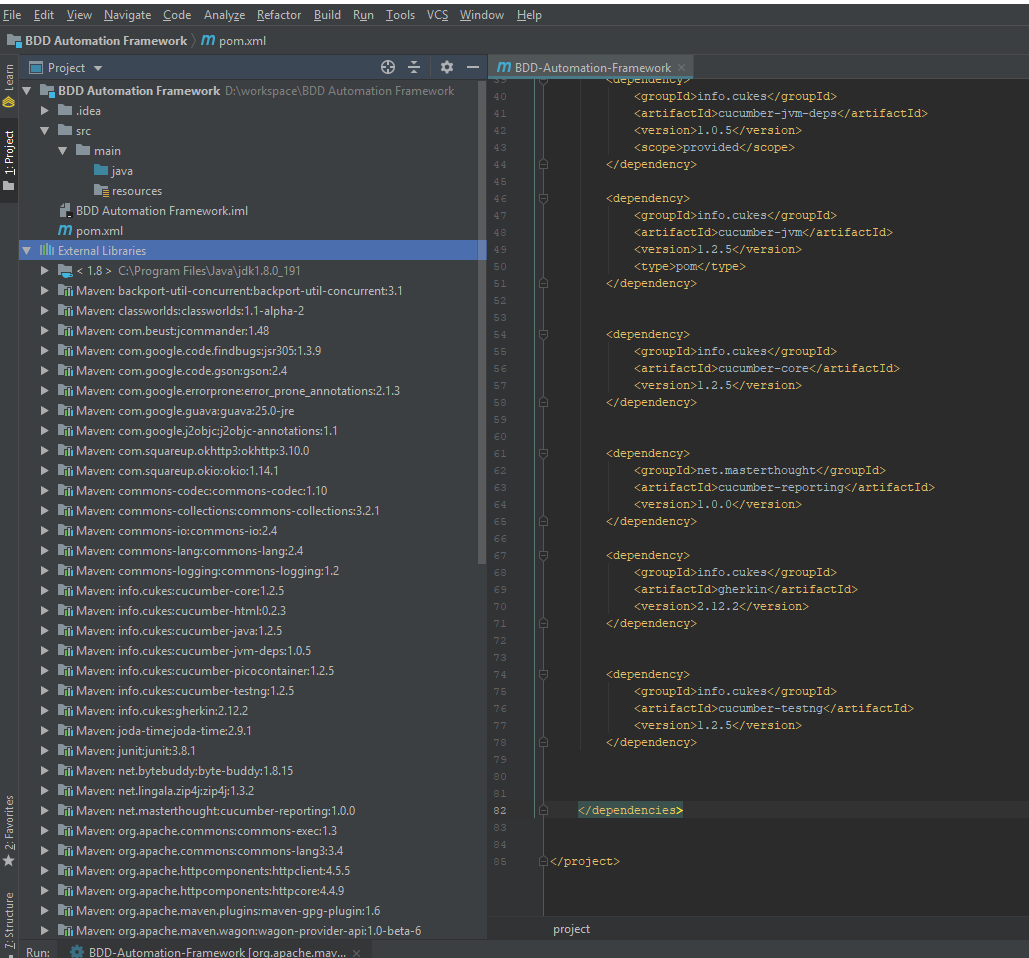
As we are managing project using Maven, we will be adding all the dependencies using maven *pom* only. Copy the following dependencies into *pom.xml* .

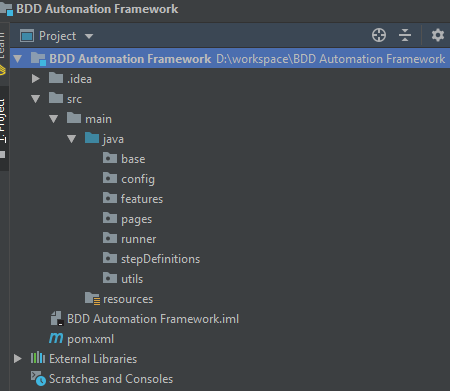
|  |
| --- |
| <dependencies>   *<!--Selenium-->*  <dependency>  <groupId>org.seleniumhq.selenium</groupId>  <artifactId>selenium-java</artifactId>  <version>3.14.0</version>  </dependency>    *<!--Cucumber-->*   <dependency>  <groupId>info.cukes</groupId>  <artifactId>cucumber-java</artifactId>  <version>1.2.5</version>  <scope>compile</scope>  </dependency>    <dependency>  <groupId>info.cukes</groupId>  <artifactId>cucumber-picocontainer</artifactId>  <version>1.2.5</version>  <scope>test</scope>  </dependency>    <dependency>  <groupId>info.cukes</groupId>  <artifactId>cucumber-jvm-deps</artifactId>  <version>1.0.5</version>  <scope>provided</scope>  </dependency>   <dependency>  <groupId>info.cukes</groupId>  <artifactId>cucumber-jvm</artifactId>  <version>1.2.5</version>  <type>pom</type>  </dependency>    <dependency>  <groupId>info.cukes</groupId>  <artifactId>cucumber-core</artifactId>  <version>1.2.5</version>  </dependency>    <dependency>  <groupId>net.masterthought</groupId>  <artifactId>cucumber-reporting</artifactId>  <version>1.0.0</version>  </dependency>   <dependency>  <groupId>info.cukes</groupId>  <artifactId>gherkin</artifactId>  <version>2.12.2</version>  </dependency>    <dependency>  <groupId>info.cukes</groupId>  <artifactId>cucumber-testng</artifactId>  <version>1.2.5</version>  </dependency>   *<!-- TestNG -->*    <dependency>  <groupId>org.testng</groupId>  <artifactId>testng</artifactId>  <version>6.14.3</version>  <scope>compile</scope>  </dependency>    *<!--WebDriver Manager-->*   <dependency>  <groupId>io.github.bonigarcia</groupId>  <artifactId>webdrivermanager</artifactId>  <version>3.1.1</version>  <scope>compile</scope>  </dependency>   </dependencies> |

If you have IntelliJ configured to auto-import dependencies, it will automatically import them for you. Otherwise, you can manually import them by opening the *Maven* menu on the right and clicking the *Reimport all Maven Projects* icon on the top left of that menu.



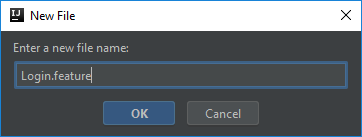
To check if your dependencies have been downloaded, you can open the *External Libraries* in the left Project menu in IntelliJ.



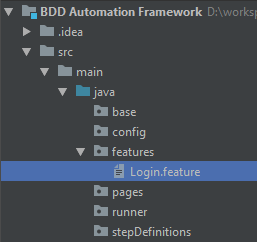
Under src/main/java create the following packages(*base, config, features, pages, runner, stepDefinitions, utils*) by right-clicking on *java* folder, then *New→ Package*. The folder structure should look like in the below image.

**Creating feature files**

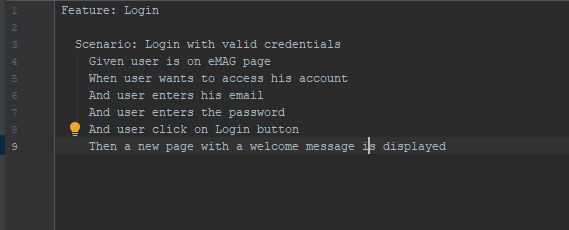
To create a feature file navigate to *src/main/java* and right click on *features* package, *New→ File* . Since we will test the login feature, the file name of the first feature will be “*Login.feature*”.



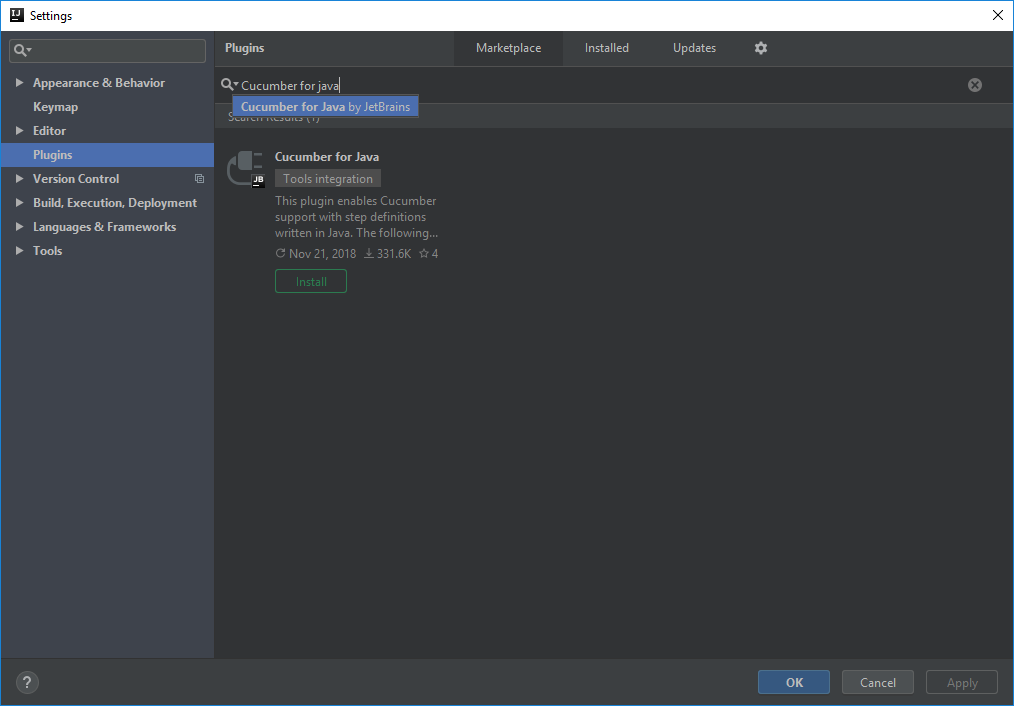
After clicking *OK*, it should look like this.



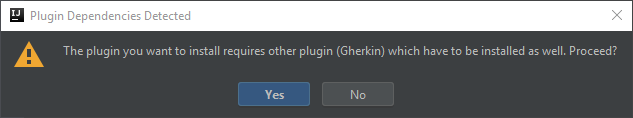
Open the file we have just created and start writing the test scenario.



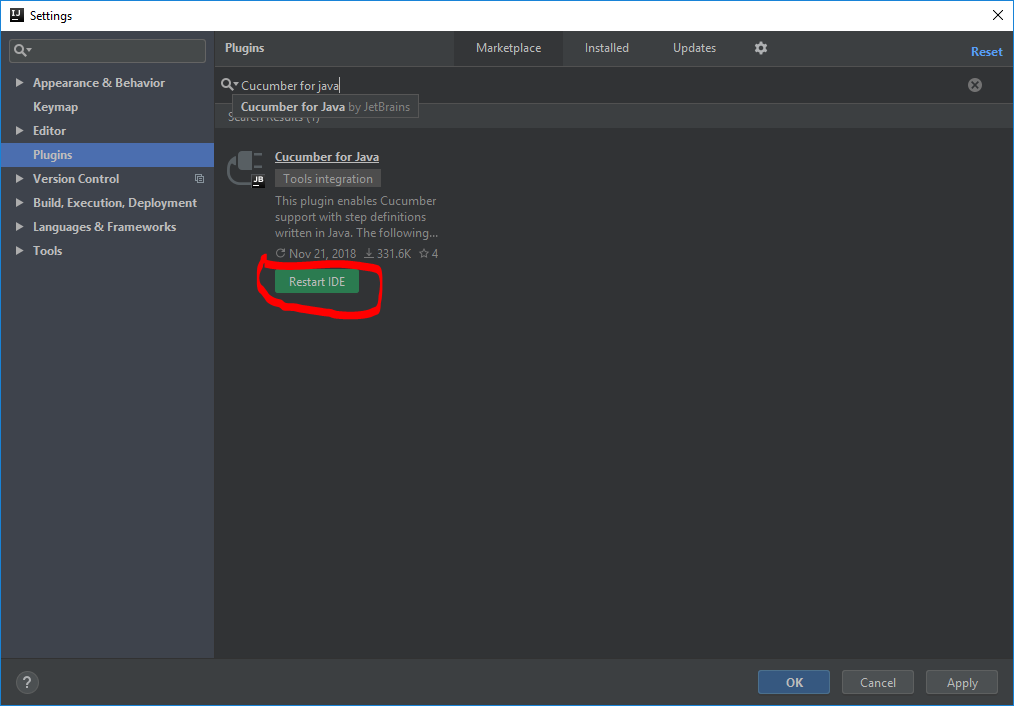
To highlight the Gherkin keywords and for a better visibility of the steps, we have to install Cucumber and Gherkin plugins. This can be done in IntelliJ going to *File→ Settings*. On the left menu select *Plugins.* On the top menu select *Marketplace*  and search for “*Cucumber for Java*” .



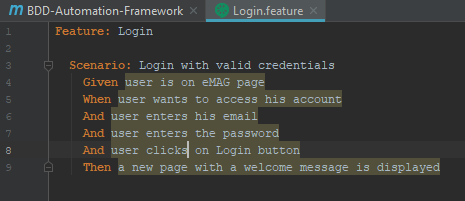
Click *Install,* and then choose *Yes* for installing also the Gherkin plugin.

**

After installation, *Restart IDE*

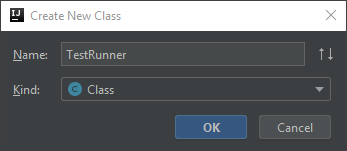
*.*

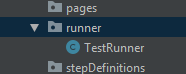
After this step, your feature file should look like this



**Creating Runner class**

The runner class will be located in the *runner* package. Right click on *runner*, *New→ Java Class*. Name it as *TestRunner*  and press *OK.*

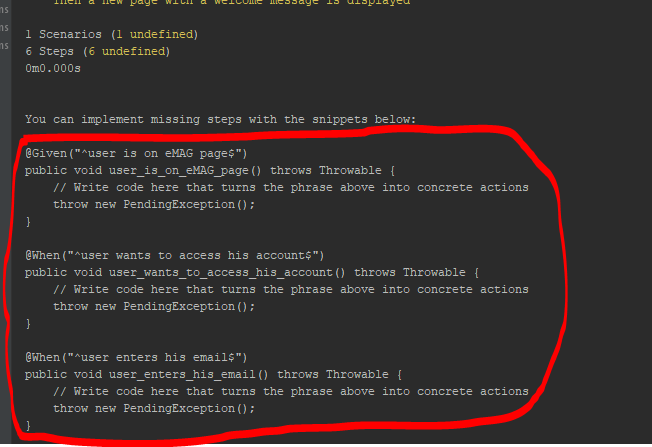
**

**

Write these lines of code in this class

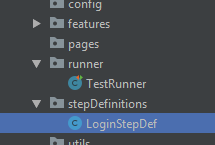
|  |
| --- |
| package runner;  import cucumber.api.CucumberOptions; import cucumber.api.testng.AbstractTestNGCucumberTests;   @CucumberOptions(  features = "src/main/java/features", *//path of the features files*  glue = {"stepDefinitions"}, *//path of the step definitions files*  dryRun = false, *//to check the mapping is proper between features files and step definitions files*  plugin = {"pretty", "html:target/cucumber-reports", "json:json\_output/cucumber.json"} *//plugin for generating different types of reporting: HTML, JSON, etc*  )   public class TestRunner extends AbstractTestNGCucumberTests { } |

To get the steps automatically generated, we need to execute TestRunner class. Right click, and choose *Run ‘TestRunner’*. You would get the below result in the console.



**Creating Step Definitions class**

Under stepDefinitions package create a Java class called *LoginStepDef*



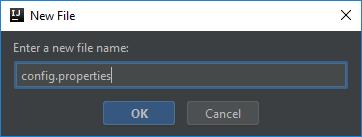
In this class we have to add the methods provided earlier in the console.

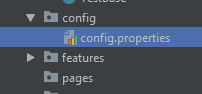
|  |
| --- |
| package stepDefinitions;  import cucumber.api.PendingException; import cucumber.api.java.en.Given; import cucumber.api.java.en.Then; import cucumber.api.java.en.When;  public class LoginStepDef {   @Given("^user is on eMAG page$")  public void user\_is\_on\_eMAG\_page() throws Throwable {  *// Write code here that turns the phrase above into concrete actions*  throw new PendingException();  }   @When("^user wants to access his account$")  public void user\_wants\_to\_access\_his\_account() throws Throwable {  *// Write code here that turns the phrase above into concrete actions*  throw new PendingException();  }   @When("^user enters his email$")  public void user\_enters\_his\_email() throws Throwable {  *// Write code here that turns the phrase above into concrete actions*  throw new PendingException();  }   @When("^user enters the password$")  public void user\_enters\_the\_password() throws Throwable {  *// Write code here that turns the phrase above into concrete actions*  throw new PendingException();  }   @When("^user clicks on Login button$")  public void user\_clicks\_on\_Login\_button() throws Throwable {  *// Write code here that turns the phrase above into concrete actions*  throw new PendingException();  }   @Then("^a new page with a welcome message is displayed$")  public void a\_new\_page\_with\_a\_welcome\_message\_is\_displayed() throws Throwable {  *// Write code here that turns the phrase above into concrete actions*  throw new PendingException();  }   } |

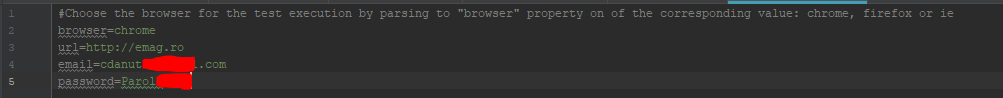
Later, in the method body we will write our Selenium code to interact with the browser.

**Creating config file and Constants class**

In *config* package create a properties file and save with .properties extension.



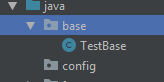


This file is used to keep the credentials, the URL and browser in which the tests will be executed. 

Also in config package create the *Constants* class

|  |
| --- |
| package config;  public class Constants {   public static final String PROPERTIES\_PATH = System.getProperty("user.dir") + "\\src\\main\\java\\config\\config.properties";  public static final long IMPLICIT\_WAIT = 10;  public static final long PAGE\_LOAD\_TIMEOUT = 15;   } |

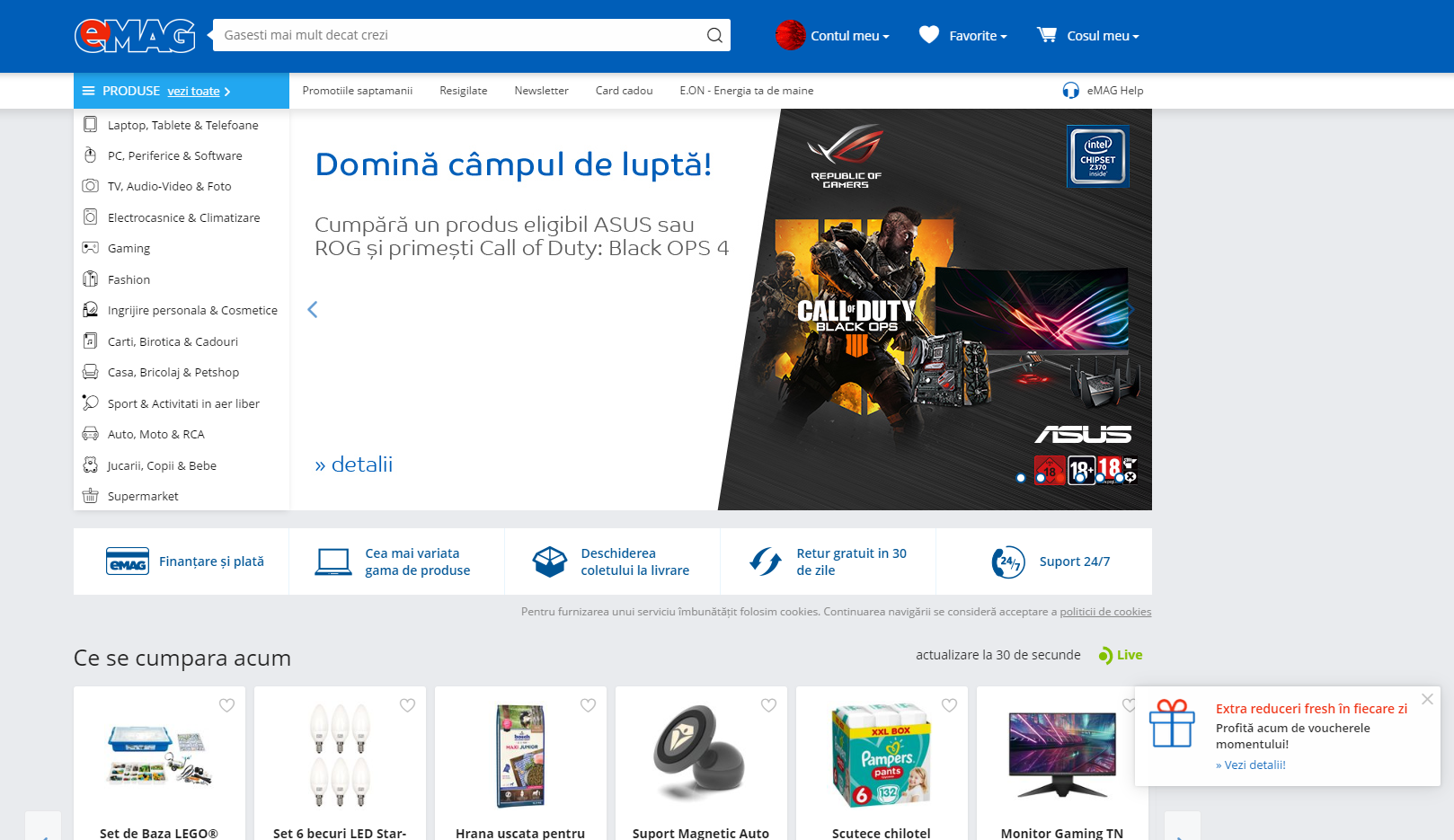
**Creating TestBase Class**

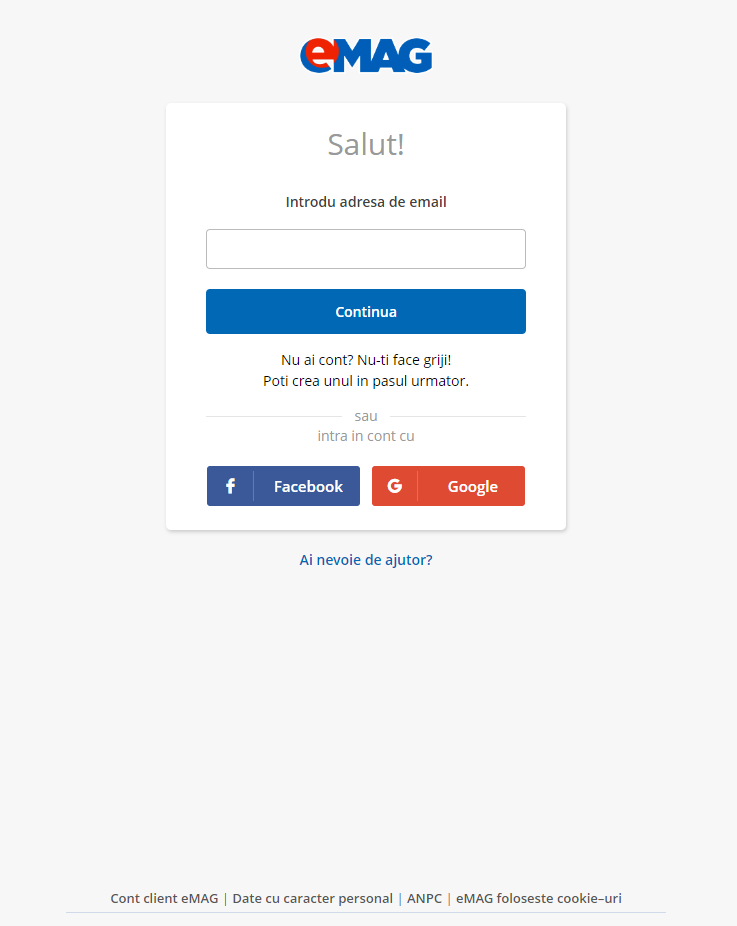
****

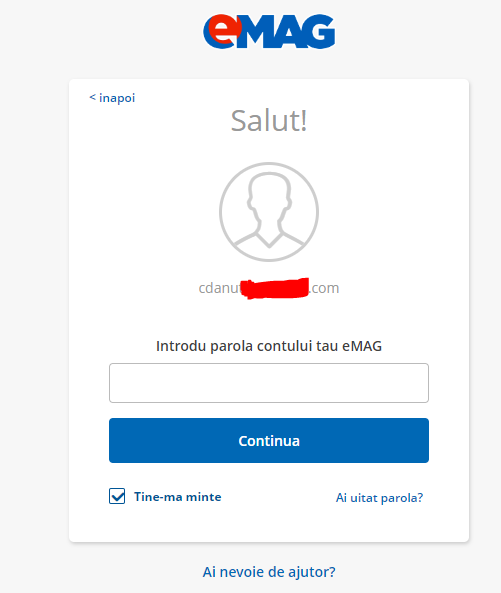
Here we create an object of WebDriver, maximize browser, implementing waits, launching URL.

|  |
| --- |
| package base;  import config.Constants; import io.github.bonigarcia.wdm.WebDriverManager; import org.openqa.selenium.WebDriver; import org.openqa.selenium.chrome.ChromeDriver; import org.openqa.selenium.firefox.FirefoxDriver; import org.openqa.selenium.ie.InternetExplorerDriver; import org.openqa.selenium.support.ui.WebDriverWait;  import java.io.FileInputStream; import java.io.FileNotFoundException; import java.io.IOException; import java.util.Properties; import java.util.concurrent.TimeUnit;  public class TestBase {   *//Objects declaration*  public static WebDriver driver;  public static Properties prop;  public static WebDriverWait wait;   *//Constructor of the class. The properties file is loaded*  public TestBase() {  try {  prop = new Properties();  FileInputStream fis = new FileInputStream(Constants.PROPERTIES\_PATH);  prop.load(fis);  } catch (  FileNotFoundException e) {  System.out.println("The properties file couldn't be found in the specified location ");  } catch (  IOException e) {  System.out.println("Couldn't read information from specified properties file");  }  }    public static void intialization() {  String browserName = prop.getProperty("browser");    *//choose the browser for tests execution*  try {  switch (browserName) {  case "chrome":  if (driver == null) {  WebDriverManager.chromedriver().setup();  driver = new ChromeDriver();  break;  }  case "firefox":  if (driver == null) {  WebDriverManager.firefoxdriver().setup();  driver = new FirefoxDriver();  break;  }  case "ie":  if (driver == null) {  WebDriverManager.iedriver().setup();  driver = new InternetExplorerDriver();  break;  }  }  } catch (Exception e) {  System.out.println("Unable to load the browser");  }  *//instantiate an object of explicit wait type*  wait = new WebDriverWait(driver, 10);    driver.manage().window().maximize();  driver.manage().deleteAllCookies();   driver.manage().timeouts().pageLoadTimeout(Constants.PAGE\_LOAD\_TIMEOUT, TimeUnit.SECONDS);  driver.manage().timeouts().implicitlyWait(Constants.IMPLICIT\_WAIT, TimeUnit.SECONDS);   *//launch the browser and navigate to the address specified in properties file*  driver.get(prop.getProperty("url"));  } } |

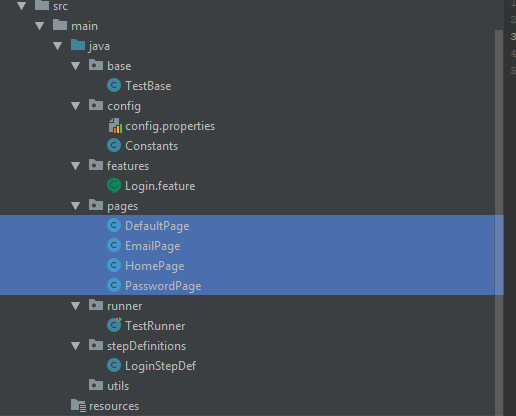
**Creating page objects**

For each web page of the application a Java class will be created (e.g. Default Page, Email Page, Password Page, etc) to hold element locators and their methods. 





To create the Java classes go to *pages* package and right click on it, *New→ Java class* and name each class with the corresponding name.



DefaultPage class

|  |
| --- |
| package pages;  import base.TestBase; import org.openqa.selenium.WebElement; import org.openqa.selenium.interactions.Actions; import org.openqa.selenium.support.FindBy; import org.openqa.selenium.support.PageFactory; import org.openqa.selenium.support.ui.ExpectedConditions;  *//extends TestBase in order to inherit all variables and methods from it* public class DefaultPage extends TestBase {    *//Using FindBy for locating elements*  @FindBy(xpath = "//a[contains(@href,'homepage')]")  private WebElement inapoiInSite\_link;  @FindBy(xpath = "//span[text()='Contul meu ']")  private WebElement contulMeu\_button;  @FindBy(xpath = "//a[text()='Intra in cont']")  private WebElement intraInCont\_button;  @FindBy(xpath = "//span[contains(text(),'Favorite')]")  private WebElement favorite\_button;  @FindBy(xpath = "//span[contains(text(),'Cosul meu')]")  private WebElement cosulMeu\_button;    public DefaultPage() {  *//initialize the elements of the Page Object*  PageFactory.initElements(driver, this);  }   *// Defining the user actions (Methods) that can be performed on the Default page*   *//This method is used to mouse hover and then to click on a web element*  public EmailPage clickOnSignInButton() {   *//mouse hover*  Actions action = new Actions(driver);  wait.until(ExpectedConditions.elementToBeClickable(contulMeu\_button));  action.moveToElement(contulMeu\_button).build().perform();   *//wait until element became clickable*  wait.until(ExpectedConditions.elementToBeClickable(intraInCont\_button));  *//click*  intraInCont\_button.click();  *//return the new page(EmailPage)*  return new EmailPage();  } } |

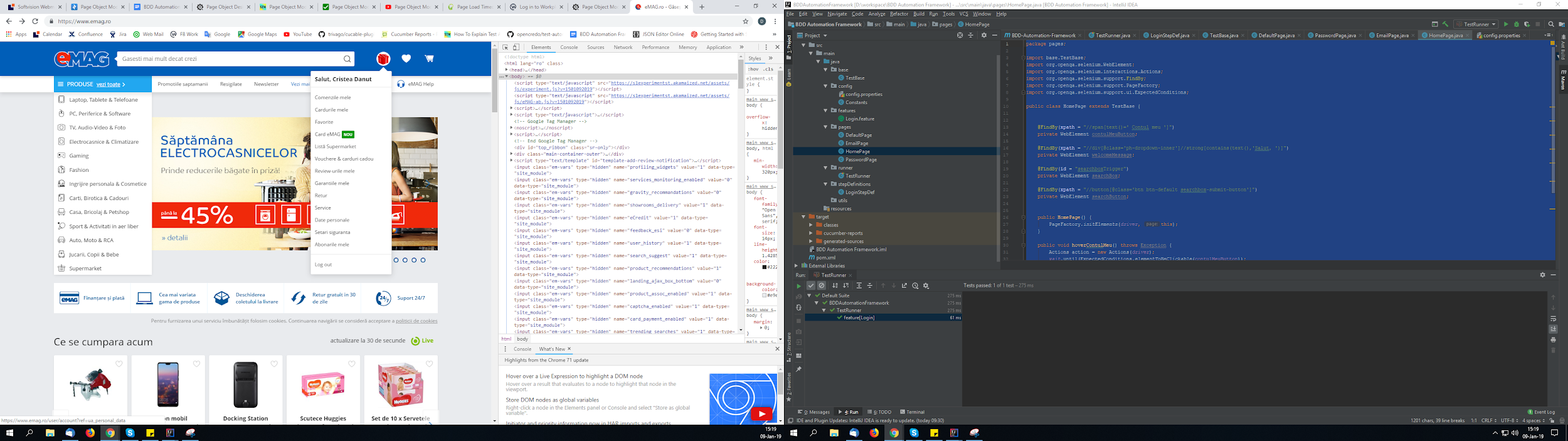
EmailPage class

|  |
| --- |
| package pages;  import base.TestBase; import org.openqa.selenium.WebElement; import org.openqa.selenium.support.FindBy; import org.openqa.selenium.support.PageFactory;   public class EmailPage extends TestBase {  @FindBy(id = "email")  private WebElement email\_input;  @FindBy(xpath = "//button[contains(text(),'Continua')]")  private WebElement continua\_button;   public EmailPage() {  PageFactory.initElements(driver, this);  }   *//write in the email text box the email address provided in config.properties file*  public void insertEmail() {   email\_input.sendKeys(prop.getProperty("email"));  }   public PasswordPage clickOnContinueButton() {  continua\_button.click();    *//after clicking the button "Continua", a new page will be displayed --> PasswordPage*  return new PasswordPage();  } } |

PasswordPage class

|  |
| --- |
| package pages;  import base.TestBase; import org.openqa.selenium.WebElement; import org.openqa.selenium.support.FindBy; import org.openqa.selenium.support.PageFactory;  public class PasswordPage extends TestBase {  @FindBy(id = "password")  private WebElement password\_input;  @FindBy(xpath = "//button[contains(text(),'Continua')]")  private WebElement continua\_button;   public PasswordPage() {  PageFactory.initElements(driver, this);  }   public void insertPassword() {  password\_input.sendKeys(prop.getProperty("password"));  }   public HomePage clickOnContinuaButton() {  continua\_button.click();  return new HomePage();  } } |

After entering the valid credentials, a new page will be displayed, HomePage.



HomePage class

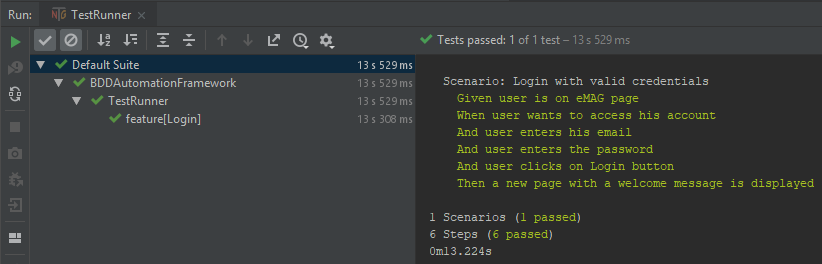
|  |
| --- |
| package pages;  import base.TestBase; import org.openqa.selenium.WebElement; import org.openqa.selenium.interactions.Actions; import org.openqa.selenium.support.FindBy; import org.openqa.selenium.support.PageFactory; import org.openqa.selenium.support.ui.ExpectedConditions;  public class HomePage extends TestBase {    @FindBy(xpath = "//span[text()=' Contul meu ']")  private WebElement contulMeuButton;   @FindBy(xpath = "//div[@class='ph-dropdown-inner']//strong[contains(text(),'Salut, ')]")  private WebElement welcomeMessage;   @FindBy(id = "searchboxTrigger")  private WebElement searchBox;   @FindBy(xpath = "//button[@class='btn btn-default searchbox-submit-button']")  private WebElement searchButton;    public HomePage() {  PageFactory.initElements(driver, this);  }   public void hoverContulMeu() {  Actions action = new Actions(driver);  wait.until(ExpectedConditions.elementToBeClickable(contulMeuButton));  action.moveToElement(contulMeuButton).build().perform();  }   public String getWelcomeMessageText() {  hoverContulMeu();  return welcomeMessage.getText();  } } |

**Usage of Page Objects in Step Definition file of Cucumber**

|  |
| --- |
| package stepDefinitions;  import base.TestBase; import cucumber.api.java.en.And; import cucumber.api.java.en.Given; import cucumber.api.java.en.Then; import cucumber.api.java.en.When; import org.testng.Assert; import pages.DefaultPage; import pages.EmailPage; import pages.HomePage; import pages.PasswordPage;  *//extends TestBase class to have access on its methods* public class LoginStepDef extends TestBase{   *//declare objects for each page object class*  DefaultPage defaultPage;  EmailPage emailPage;  PasswordPage passwordPage;  HomePage homePage;    @Given("^user is on eMAG page$")  public void user\_is\_on\_eMAG\_page() {  *//call the static method from parent class(TestBase) to open the browser and to navigate to application*  TestBase.intialization();  *//instantiate an object of type DefaultPage*  defaultPage = new DefaultPage();  }   @When("^user wants to access his account$")  public void user\_wants\_to\_access\_his\_account() {  emailPage = defaultPage.clickOnSignInButton();  }   @When("^user enters his email$")  public void user\_enters\_his\_credentials() {  emailPage.insertEmail();  passwordPage = emailPage.clickOnContinueButton();  }   @And("user enters the password")  public void userEntersThePassword() {  passwordPage.insertPassword();  }   @When("^user clicks on Login button$")  public void user\_clicks\_on\_Login\_button() {  homePage = passwordPage.clickOnContinuaButton();  }   @Then("^a new page with a welcome message is displayed$")  public void a\_new\_page\_with\_a\_welcome\_message\_is\_displayed() {  *//Asserts helps us to verify the conditions of the test and decide whether test has failed or passed*  Assert.assertEquals(homePage.getWelcomeMessageText(), "Salut, Cristea Danut");  } } |

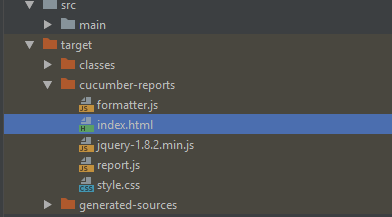
**Test execution**

After these all steps, our first test is ready for execution. Open *TestRunner* class, right click → *Run ‘TestRunner’.* After test execution finishes, we can see the result in console.

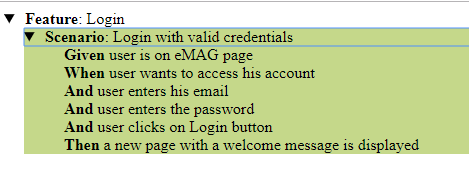


**Reporting**

Also, after test execution finishes, a HTML report will be generated. To open it, go to *target/cucumber-reports* and right click on *index.html*.

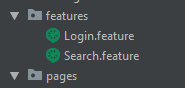


Then go to *Open in Browser* and choose the any browser you want to open the file. The result will be displayed like in the image



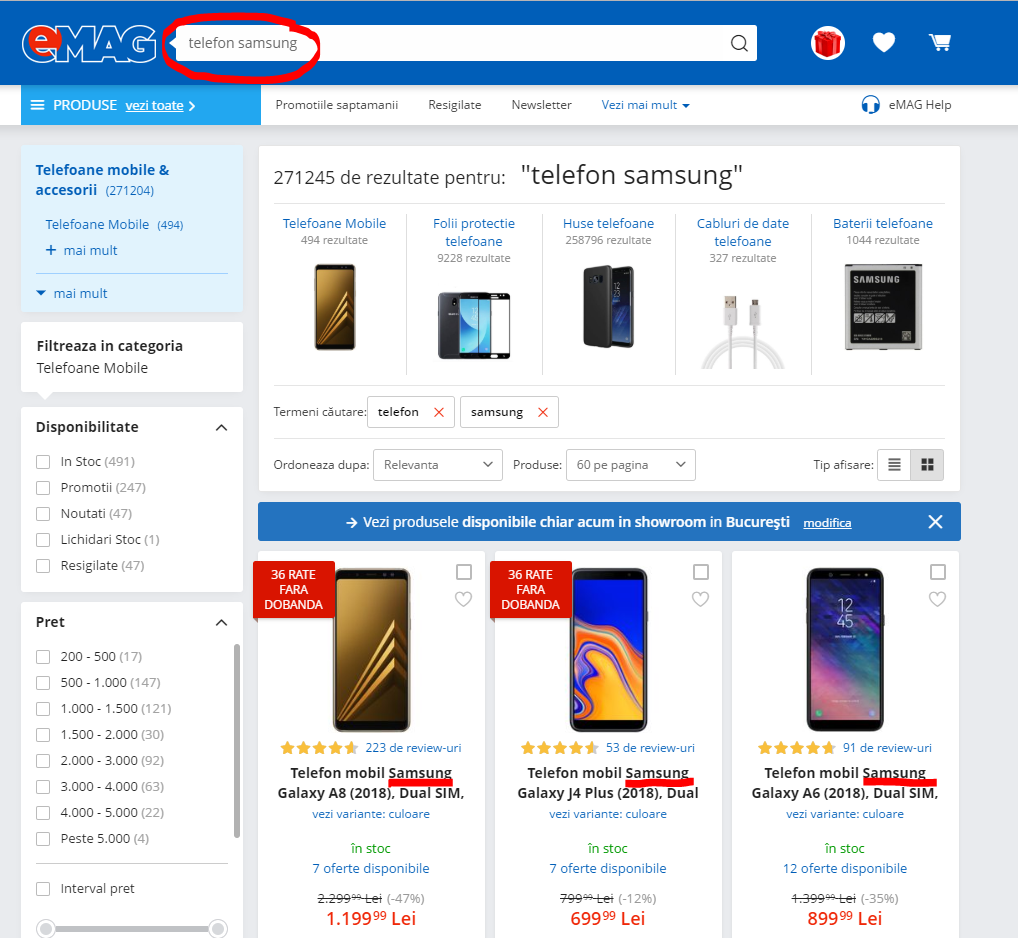
As we can see throughout the green color, all the step have passed, so our login test passed.

Now let’s add a new feature in our project, searching feature.



|  |
| --- |
| **Feature: Search functionality   Scenario Outline: Search a valid product  Given user is logged in  When user search for "<product>"  Then a page with "<results>" is displayed   Examples:  | product | results |  | telefon samsung | Samsung |  | laptop dell | Dell |** |

In this scenario we want to verify if each product listed on the result page contains the expected name.



Since a new page will be displayed when searching for a product, the page with results, we have to create a new page object, *ResultsPage .*

|  |
| --- |
| *package pages;  import base.TestBase; import org.openqa.selenium.WebElement; import org.openqa.selenium.support.FindBy; import org.openqa.selenium.support.FindBys; import org.openqa.selenium.support.PageFactory;  import java.util.ArrayList; import java.util.List;  public class ResultsPage extends TestBase {   //find multiple web elements  @FindBys({@FindBy(xpath = "//h2/a")})  private List<WebElement> resultsList;   public ResultsPage() {  PageFactory.initElements(driver, this);  }   //method to get the name of each product  public List<Boolean> containsProductsName(String productName) {  //create a list of boolean  List<Boolean> titlesList = new ArrayList<>();  for (WebElement element : resultsList) {  //add in list true or false depending on searched product name existence  titlesList.add(element.getText().contains(productName));  }  return titlesList;  } }* |

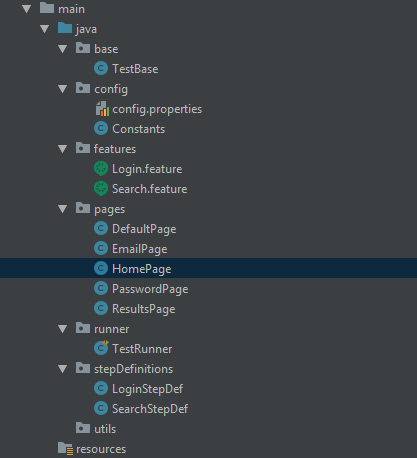
The next step is to add the step definitions file. So we will create a new class called *SearchStepDef .*

|  |
| --- |
| package stepDefinitions;  import base.TestBase; import cucumber.api.java.en.Given; import cucumber.api.java.en.Then; import cucumber.api.java.en.When; import org.testng.Assert; import pages.\*;   public class SearchStepDef extends TestBase {   DefaultPage defaultPage;  EmailPage emailPage;  PasswordPage passwordPage;  HomePage homePage;  ResultsPage resultsPage;   *//preconditions of the test*  @Given("^user is logged in$")  public void user\_is\_logged\_in() {  TestBase.intialization();  defaultPage = new DefaultPage();  emailPage = defaultPage.clickOnSignInButton();  emailPage.insertEmail();  passwordPage = emailPage.clickOnContinueButton();  passwordPage.insertPassword();  homePage = passwordPage.clickOnContinuaButton();  }    @When("^user search for \"(.\*)\"$")  public void user\_search\_for\_product(String productName) {  resultsPage = homePage.searchProduct(productName);   }   @Then("^a page with \"(.\*)\" is displayed$")  public void a\_page\_with\_expected\_product\_is\_displayed(String resultName) {  *//check if all displayed products contain the expected result*  for (boolean flag : resultsPage.containsProductsName(resultName)) {  Assert.assertTrue(flag);  }  } } |

A new modification will be required before running the tests, the *searchProduct* method addition in the *HomePage* class.

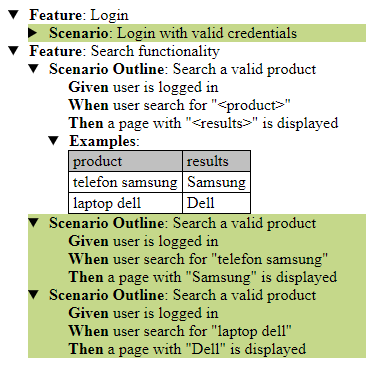
|  |
| --- |
| public ResultsPage searchProduct(String productName) {  *//write the name of the product in the search box*  searchBox.sendKeys(productName);  searchButton.click();  *//return the landing page with results*  return new ResultsPage(); } |

The structure of the project so far will look like this



To run the tests execution, we will follow the same steps as presented earlier. Right click inside *TestRunner* class, and then click on *Run ‘TestRunner’.*

A new HTML report will be generated

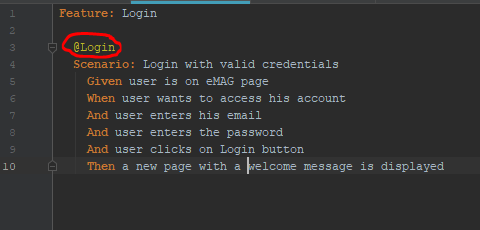


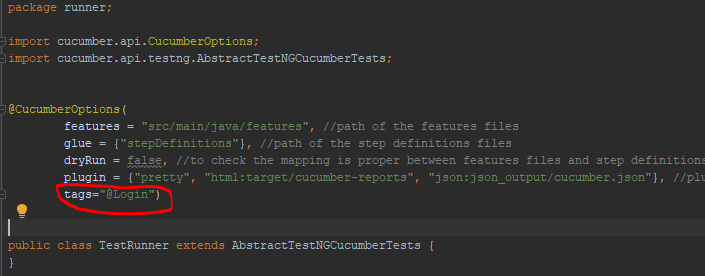
Both Login and Search tests have been executed. Since we have used two sets of data, given in *Examples* , the Search test was executed two times, each time with different set of data.

**Usage of the Cucumber tags**

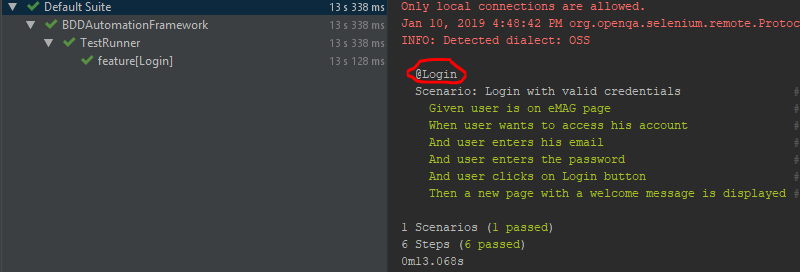
Let’s suppose our project has a lot of features and we want to execute just the login test for ensuring that the other tests won’t be executed unless we are able to enter our account and not wasting time exectuted them. For that we will make use of [Cucumber tags](https://www.toolsqa.com/cucumber/cucumber-tags/).

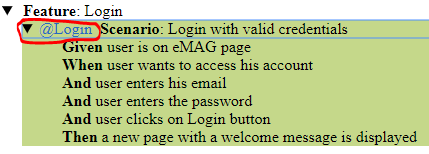
Two small modifications we have to do. First, add *@Login* tag or whatever meaningful name you want in the *Login.feature* file and the second step is to add the option *tags* in the *TestRunner* file and specify which tag or tags we want to execute.





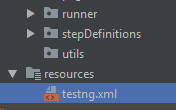
Now run the *TestRunner* class. Only the login test will be executed.





**Running tests from Maven**

It is also possible to run our tests directly from command prompt using a simple command. First, we have to configure our project. Under *src/main* there is a folder called *resources.* Right click on it *New→ File.* Enter the name as *testng.xml.*

**

Inside this file add these lines of xml code.

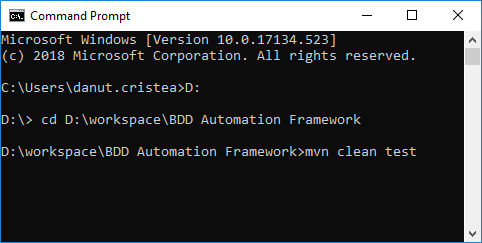
|  |
| --- |
| <?xml version="1.0" encoding="UTF-8"?> <!DOCTYPE suite SYSTEM "http://testng.org/testng-1.0.dtd"> <suite name="Default Suite" >  <test name="Cucumber tests">  <classes>  *<!--set the relative path of the runner class-->*  <class name="runner.TestRunner"></class>  </classes>  </test> </suite> |

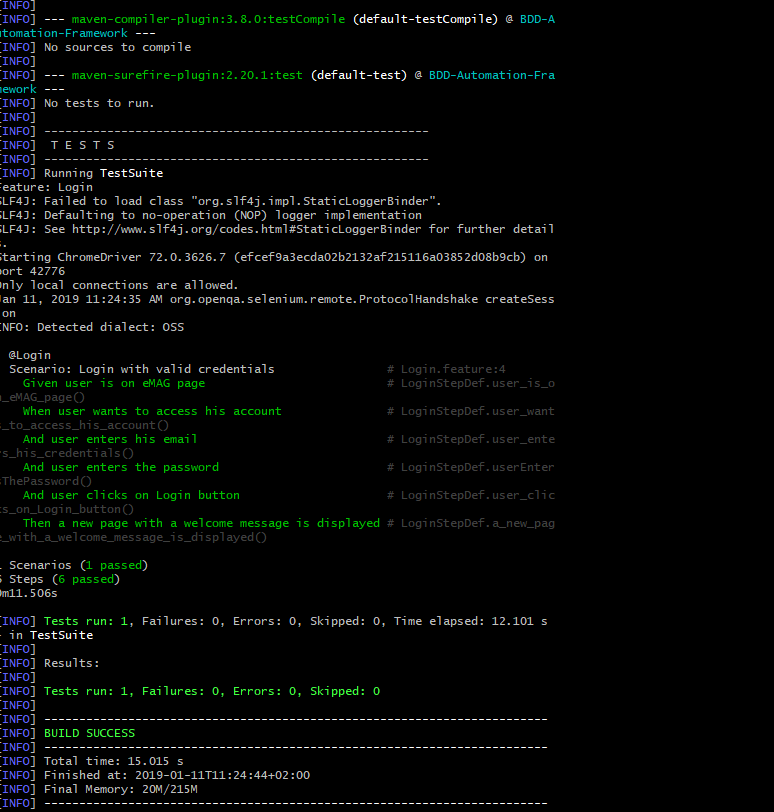
The most important thing to notice here is that we have to provide the path of the runner class as name attribute inside <class> tag.

Then open *pom.xml* and above <dependencies> tag add these lines

|  |
| --- |
| <build>  <plugins>  <plugin>  <groupId>org.apache.maven.plugins</groupId>  <artifactId>maven-compiler-plugin</artifactId>  <version>3.8.0</version>  <configuration>  <source>1.8</source>  <target>1.8</target>  </configuration>  </plugin>  <plugin>  <groupId>org.apache.maven.plugins</groupId>  <artifactId>maven-surefire-plugin</artifactId>  <version>2.20.1</version>  <configuration>  <suiteXmlFiles>  <suiteXmlFile>src/main/resources/testng.xml</suiteXmlFile>  </suiteXmlFiles>  </configuration>  </plugin>  </plugins> </build> |

Make sure inside <suiteXmlFile> tag the path to the *testng.xml* file is correct(*src/main/resources/testng.xml*). Now we are all set. The following step is to open the command prompt and to navigate to the folder address of the project. Here you have to write to following command: *mvn clean test*  and press Enter.





As shown in the above image, only the login test has executed. That’s because in the *TestRunner* class, at Cucumber options section, *tags=”@Login”*.

A folder called *target* will be generated within the project and here(/target/cucumber-reports/index.html) you can find the HTML report.

**Parallel test execution**

Parallel testing is defined as a software testing type, which checks multiple applications or subcomponents of one application concurrently to reduce the test time.

Practically, what we propose is to start at the same time both login and search tests. For that we must add in *pom.xml* two plugins, one for paralllel testing(cucumber-jvm-parallel-plugin) and the other one(cluecumber-report-plugin) for creating a report for all tests run in parallel. We also have to make some modifications for the maven-surefire-plugin. The entire *pom.xml*  file is presented below.

|  |
| --- |
| <?xml version="1.0" encoding="UTF-8"?> <project xmlns="http://maven.apache.org/POM/4.0.0"  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">  <modelVersion>4.0.0</modelVersion>   <groupId>Automation</groupId>  <artifactId>BDD-Automation-Framework</artifactId>  <version>1.0-SNAPSHOT</version>  <build>  <plugins>  <plugin>  <groupId>org.apache.maven.plugins</groupId>  <artifactId>maven-compiler-plugin</artifactId>  <version>3.8.0</version>  <configuration>  <source>1.8</source>  <target>1.8</target>  </configuration>  </plugin>  *<!--<plugin>-->*  *<!--<groupId>org.apache.maven.plugins</groupId>-->*  *<!--<artifactId>maven-surefire-plugin</artifactId>-->*  *<!--<version>2.20.1</version>-->*  *<!--<configuration>-->*  *<!--<suiteXmlFiles>-->*  *<!--<suiteXmlFile>src/main/resources/testng.xml</suiteXmlFile>-->*  *<!--</suiteXmlFiles>-->*  *<!--</configuration>-->*  *<!--</plugin>-->*    <plugin>  <groupId>org.apache.maven.plugins</groupId>  <artifactId>maven-surefire-plugin</artifactId>  <version>2.20.1</version>  <configuration>  *<!--The parameter forkCount defines the maximum number of JVM processes that maven-surefire-plugin will spawn concurrently to execute the tests-->*  <forkCount>2</forkCount>  <reuseForks>true</reuseForks>  <includes>  <include>\*\*/\*IT.class</include>  </includes>  <testFailureIgnore>true</testFailureIgnore>  </configuration>  </plugin>   <plugin>  <groupId>com.github.temyers</groupId>  <artifactId>cucumber-jvm-parallel-plugin</artifactId>  <version>2.1.0</version>  <executions>  <execution>  <id>generateRunners</id>  <phase>generate-test-sources</phase>  <goals>  <goal>generateRunners</goal>  </goals>  <configuration>  *<!-- Mandatory -->*  *<!-- comma separated list of package names to scan for glue code -->*  <glue>stepDefinitions</glue>  <outputDirectory>${project.build.directory}/generated-test-sources/cucumber  </outputDirectory>  *<!-- The directory, which must be in the root of the runtime classpath, containing your feature files. -->*  <featuresDirectory>src/main/java/features/</featuresDirectory>  *<!-- Directory where the cucumber report files shall be written -->*  <cucumberOutputDir>target/cucumber-parallel</cucumberOutputDir>  *<!-- comma separated list of output formats json,html,rerun.txt -->*  <format>json,html</format>  *<!-- CucumberOptions.strict property -->*  <strict>true</strict>  *<!-- CucumberOptions.monochrome property -->*  <monochrome>true</monochrome>  *<!-- The tags to run, maps to CucumberOptions.tags property you can pass ANDed tags like "@tag1","@tag2" and ORed tags like "@tag1,@tag2,@tag3" -->*  *<!--<tags>"@SortByAvailability"</tags>-->*  *<!-- If set to true, only feature files containing the required tags shall be generated. -->*  <filterFeaturesByTags>true</filterFeaturesByTags>  *<!-- Generate TestNG runners instead of default JUnit ones. -->*  <useTestNG>true</useTestNG>  *<!-- The naming scheme to use for the generated test classes. One of 'simple' or 'feature-title' -->*  <namingScheme>feature-title</namingScheme>  *<!-- The class naming pattern to use. Only required/used if naming scheme is 'pattern'.-->*  *<!--<namingPattern>Parallel{c}IT</namingPattern>-->*  *<!-- One of [SCENARIO, FEATURE]. SCENARIO generates one runner per scenario. FEATURE generates a runner per feature. -->*  <parallelScheme>FEATURE</parallelScheme>  *<!-- This is optional, required only if you want to specify a custom template for the generated sources (this is a relative path) -->*  </configuration>  </execution>  </executions>  </plugin>   <plugin>  <groupId>com.trivago.rta</groupId>  <artifactId>cluecumber-report-plugin</artifactId>  <version>1.5.0</version>  <executions>  <execution>  <id>report</id>  <phase>post-integration-test</phase>  <goals>  <goal>reporting</goal>  </goals>  </execution>  </executions>  <configuration>  <sourceJsonReportDirectory>${project.build.directory}\cucumber-parallel</sourceJsonReportDirectory>  <generatedHtmlReportDirectory>${project.build.directory}\test-report</generatedHtmlReportDirectory>  </configuration>  </plugin>   </plugins>  </build>   <dependencies>   *<!--Selenium-->*  <dependency>  <groupId>org.seleniumhq.selenium</groupId>  <artifactId>selenium-java</artifactId>  <version>3.14.0</version>  </dependency>    *<!--Cucumber-->*   <dependency>  <groupId>info.cukes</groupId>  <artifactId>cucumber-java</artifactId>  <version>1.2.5</version>  <scope>compile</scope>  </dependency>    <dependency>  <groupId>info.cukes</groupId>  <artifactId>cucumber-picocontainer</artifactId>  <version>1.2.5</version>  <scope>test</scope>  </dependency>    <dependency>  <groupId>info.cukes</groupId>  <artifactId>cucumber-jvm-deps</artifactId>  <version>1.0.5</version>  <scope>provided</scope>  </dependency>   <dependency>  <groupId>info.cukes</groupId>  <artifactId>cucumber-jvm</artifactId>  <version>1.2.5</version>  <type>pom</type>  </dependency>    <dependency>  <groupId>info.cukes</groupId>  <artifactId>cucumber-core</artifactId>  <version>1.2.5</version>  </dependency>    <dependency>  <groupId>net.masterthought</groupId>  <artifactId>cucumber-reporting</artifactId>  <version>1.0.0</version>  </dependency>   <dependency>  <groupId>info.cukes</groupId>  <artifactId>gherkin</artifactId>  <version>2.12.2</version>  </dependency>    <dependency>  <groupId>info.cukes</groupId>  <artifactId>cucumber-testng</artifactId>  <version>1.2.5</version>  </dependency>   *<!-- TestNG -->*    <dependency>  <groupId>org.testng</groupId>  <artifactId>testng</artifactId>  <version>6.14.3</version>  <scope>compile</scope>  </dependency>    *<!--WebDriver Manager-->*    <dependency>  <groupId>io.github.bonigarcia</groupId>  <artifactId>webdrivermanager</artifactId>  <version>3.1.1</version>  <scope>compile</scope>  </dependency>  </dependencies> </project> |

We will run the tests from command prompt following the same procedure excepting the Maven command, which , in this case is *mvn clean verify*. To learn more about Maven lifecycle please visit this [address](https://maven.apache.org/guides/introduction/introduction-to-the-lifecycle.html).

After executing this command, two web browsers should open at the same time. The test report can be found in */target/test-report/index.html*. The results are presented in a user friendly report with pie charts, graphs and elapsed time for each step.

