Кукумбер был создан для реализации BDD (behavior driven development) Назначение данного инструмента- служить некой понятной, общей частью между всеми отделами в компании. Так как главный документ .**Feature**  представляет собой больше классический английский текст, где описывается сценарий тестирования с предопределёнными, зарезервированными словами(они называются steps) в тексте за которыми и располагается реализация.

Для работы с cucumber надо зависимость cucumber.java + plagin (чтобы Intelleje IDEA распозновала синтаксис в feature файле).

Вся работа происходит в .feature файле, Создаётся файл с расширением .feature, здесь описывается сценарий выполнения тестов с помощью предопределённых/зарезервированных слов (standard structure), они определяют шаги/steps выполнения сценария. It helps to separate business logic from the behavior, and its written in Gherkin. Gherkin - serves as project documentation and also as an executable specification that can be automated.

**~ Gherkin needs to be declarative ! ! !**

**WHAT IS FEATURE IN GHERKIN SYNTAX**

One Gherkin document basically = one feature document. (one can not have multiple features in one .feature file)

A feature is a keyword that provides a high level description about the software featurethat needs to be developed. So basically every Gherkin document has to be started with keyword **Feature.**

The name is basically - the title of the feature. For example “Add to cart”.

And the description can be the user story (feature to be developed) that is being represented by this feature. For example:

“As a customer of an app //The first statement represent the actor

I want to add a product to the cart //The action that the actor wants to perform

So that I can purchase this product”.//The business goal that is going to be derived from this user story.

So this is the typical **user story** that we write in agile, and all this staff is ignored by cucumber during run time.

But this information shows up in the report, so it's very important from the perspective of documentation.

One feature can have multiple scenarios, multiple rules, but one background for all the scenarios. And if there are rules each rule can have its own background.

**What is the relation between user story and feature?**

User story is nothing but a task that we take up for a particular sprint. Once the user story is implemented the sprint is over it does not have much significance. It may be representing requirement (требование, feature) or a part of it.

The feature is the requirement(требование, другими словами это функция) that we are developing. feature can be a user story, or a feature can be represented by multiple user stories.

So one user story CANNOT HAVE MULTIPLE FEATURES. And a feature can have multiple user stories.

Что тут написано - один user story это бой, а feature это как большая война в которой много боёв. В одном бое не может быть несколько войн, но в одной войне может быть много боёв (точно также как и один бой). Как бы в одной функции много более мелких опций.

**Scenario**

Scenario is a concrete example that illustrates a specific business situation. It is similar to the test case that is written for user story but the only difference is the scenario illustrates the behavior of the system and not the behavior of the test, and the behavior is directly derived (получено) from the requirement. The scenario can have the title just like Feature and the description.

The scenario consists of different steps and these steps can be: Given, When, Then, And

We can have multiple Given, When, Then, And statements. But an important point to note is that the number of these statements should be minimum. The general rule is the number of steps should be 5 - 8. And the primary reason for that is that the scenario is supposed to be a specification and executable specification that would be read by business stakeholders (заинтересованные стороны). So if you put a lot of steps over there it will lose its expressive power as a specification and documentation.

All these steps are optional, for example the Given statement is not really mandatory, the important thing over here is the **sequence** (последовательность) of the steps and that is what matters. If you put the When statement before the Given it would not make sense. The given statement should be the first statement.

Feature can have multiple scenarios, and if there's a rule defined or several rules, you can put the scenario under the specific rule. And we can tag the scenario (use annotation in .feature fail).

It is good practice to keep each scenario independent.

**Steps**

The cucumber executes the steps in sequence, one step at a time. When cucumber comes to the step it basically finds the matching step definition to execute.(речь идёт о java файле в котором находится реализация соответствующего step-a)

**!!!ВАЖНО!!!** - раньше я думал что коомпилятор знает какой шаг выполнить благодаря названию шагов в .feature файле и названию аннотаций в java файле, они ж одинаковы, я думал что они связаны благодаря именно этому механизму. Оказывается нет связывание происходит благодаря тексту написанному в отдельно взятом шаге в feature файле и тексту который указан в аннотации в .java файле, при этом название аннотации не обязательно должно иметь одинаковое название с шагом (key word) из .feature файла, но текст должен быть одинаков и там и там. Видимо это обусловлено тем что в .feature файле может быть много одинаковых (step) шагов, но логика (описание) у них будет ведь разной. и ещо - если описание шагов одинаково, а key words разные то в .java файле будет только одна реализация.

**Feature**: описание, в принципе назначение данного документа

**Background**: то что происходит до выполнения основного сценария

**Given**: то что у нас дано ( my account balance is 100$), describe the initial state of the system, the precondition. Something that has happened in the past. So in the corresponding step definition we need to write the code that will put the system to the state described in the Given key word. It is normal to have several Given.

**When:** it is used to describe an invent or an action that user is performing on the system, this is the action that I am performing. For example: I withdraw 50 $.For a better understanding try to have only one When statement, if its not possible, use And statement but not more than 2. If scenario looks like it needs more When statements, probably this scenario is too complex and its trying to do a lot of things, it could be multiple scenarios, may be better solution is to split the scenario in 2.

**And:**

**Then:** This is used to describe the expected outcome, what should happen when the action is performed. For example:

Given: my account balance is 100$

When: I withdraw 50 $

Then: the account balance should be 50$

So in step definition method most of the time it's going to do assertion to compare actual outcome with the expected outcome. And the outcome should be observable. For example: if you put something like “the record should be created in data base” than it may not be observable if you writing the scenario for front end.

**But:** typically it is used for representing negative action.

For example:

Given: I have a bank account

But: I forgot to bring my debit card.

or

Given: my account balance is 100$

When: I withdraw 50 $

Then: the account balance should be 50$

But: I should not get an email

**\* -** use it instead of multiple And, But statements

For example:

Given: I am at grocery shop

And: I have bananas

\*: I have grapes

\*: I have watermelon

So it's basically to simplify the sintex. When we use asterisk its like we use a list of things

**Rule:**

**Anti patterns to avoid**

The main idea it is to describe the behavior and not how it's done, how part is done in a step definition.

1)-For a Given step it should describe something that happened in the past, avoid talking about user interaction. Don't write user action.

For example:

Given: I am navigating to the user page. ←This is wrong

This represents some actions that you are performing. Instead of this should be:

Given: I am on the login page.

2)-step When. It is strongly recommended to have only 1. If you think that you need more of this statement, may be you dealing with complex scenario. If you write more of this statement it might lose the expressive nature.

For example:

When: I provide my credentials ← This is correct as long as one step clearly defines the action the user is performing

When:I provide userName ← This is not good

And: I provide password ← Avoid doing this

And: I press login button

3)-step Then - here we typically write an assertion to compare the actual outcome with expected outcome. So it should be an observable output.

For example:

Then: I logged in. That's it, correct, nothing more needed, don't put any technical information

How are you providing credentials doesn't matter, the how part should be covered in step definition.

4)-For the test automation one of the most important factors is the **maintenance**! You can write as many code as you want, but maintaining the code is a real challenge. Practically the application will undergo (подвергаться) a lot of changes in some projects it can happen within a day. If you write a scenario that changes only when requirement changes and not when implementation changes than that scenario will have very less maintenance, but if you write a long scenario with technical description and interacting with UI that can change for every minor change, than it will be maintenance nightmare.

5)-The steps should be reusable, if the steps are not reusable there will be a lot of duplication. The steps should be kept under very minimum.

6)-Readability of the scenario. The scenarios should represent these specifications and not test cases. The scenario should be short.

**Scenario outline**

for example I have scenario

Scenario: Scenario with 100 $

Given: my account balance is 100$

When: I withdraw 50 $

Then: the account balance should be 50$

And now I want to execute this scenario with different values. So I created new scenario with different values.

Now I have 2 scenarios in my .feature fail.

Given: my account balance is 100$

When: I withdraw 50 $

Then: the account balance should be 50$

Scenario: Scenario with 50 $

Given: my account balance is 50$

When: I withdraw 50 $

Then: the account balance should be 50$

So for those 2 scenarios I will have the same step definition methods. 6 steps but 3 step definitions methods.The step definitions methods would be parameterized, The parameter takes the amount and uses it as argument for method.

But instead of using 2 scenarios we can use different combinations of values and we can pass it to the same scenario. Then in that case the scenario will be executed multiple times (depending on how many values we're gonna give to it).

Scenario outline: is the key word for that.

For example:

Scenari outline: Multiple value scenario.

Given: my account balance is <opening balance>

When: I withdraw <withdrawal amount>

Then: the account balance should be <closing balance>

Examples:

| opening balance | withdrawal amount | closing balance |

| 100 | 50 | 50 |

| 50 | 50 | 50 |

All the steps are parameterized. подставляются значения из таблицы где перечислены значения которые прим-

меняются в scenario, для этого в scenario указываются названия столбцов таблицы и коомпилятор подставляет соответствующие значения при выполнении. Scenario будет отработано столько раз сколько строк в таблице. If you want to add more values you simply add one more row.

@Given("my account balance is {int}")

public void myAccountBalanceIs$(int arg0) {

System.out.println(arg0);

}

@When("I withdraw {int}")

public void iWithdrawWithdrawalAmount$(int sda) {

}

@Then("the account balance should be {int}")

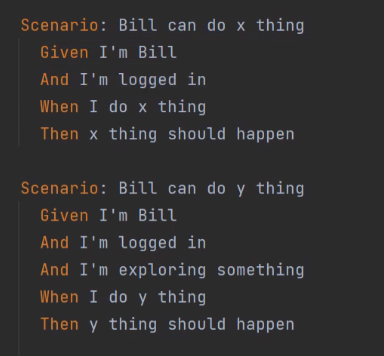
public void theAccountBalanceShouldBeClosingBalance$() {

}

**Background**

In case in a feature fail, if all the scenarios are sharing the same context, that is the same: Given step, And step… then instead of duplicating context in each of the scenarios you can use a Background keyword and reuse the steps. And if in every scenario you use the same steps it could be the indication that those steps are not essential to describe the scenarios. They are incidental details that can be moved out of the scenarios. So you can put all the repeated steps to the Background.

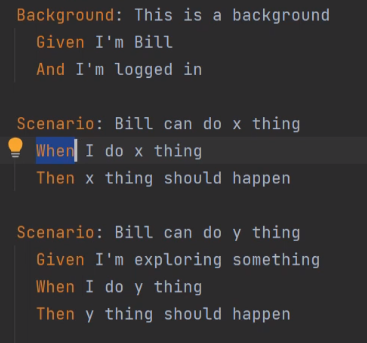
For example: we have 2 scenarios with the same steps (Given, And).



So we remove, these steps and put it in Background step

When cucumber executes any of these scenarios it will first go to the background section, it will execute these steps, then it will execute the scenario. And it will do it for all the scenarios. The steps indicated in the background should be common for all the scenarios, and the steps specific to particular scenarios should be defined in according scenario.

For a feature fail you can have only one background, if feature fail have a rule then for every rule you can have background and this background will execute for every scenario in that rule.



To use background wisely there are few rules that need to be followed.

1)- Do not complicate the background section, it means write only those steps which the user (or the customer) is going to perform. (these steps are of interest for the business), don't add technical steps like: “initializing webdriver” or “inserting record in the database” the business people wouldn't understand. All these technical steps can go to the higher level steps, which can be the hooks.

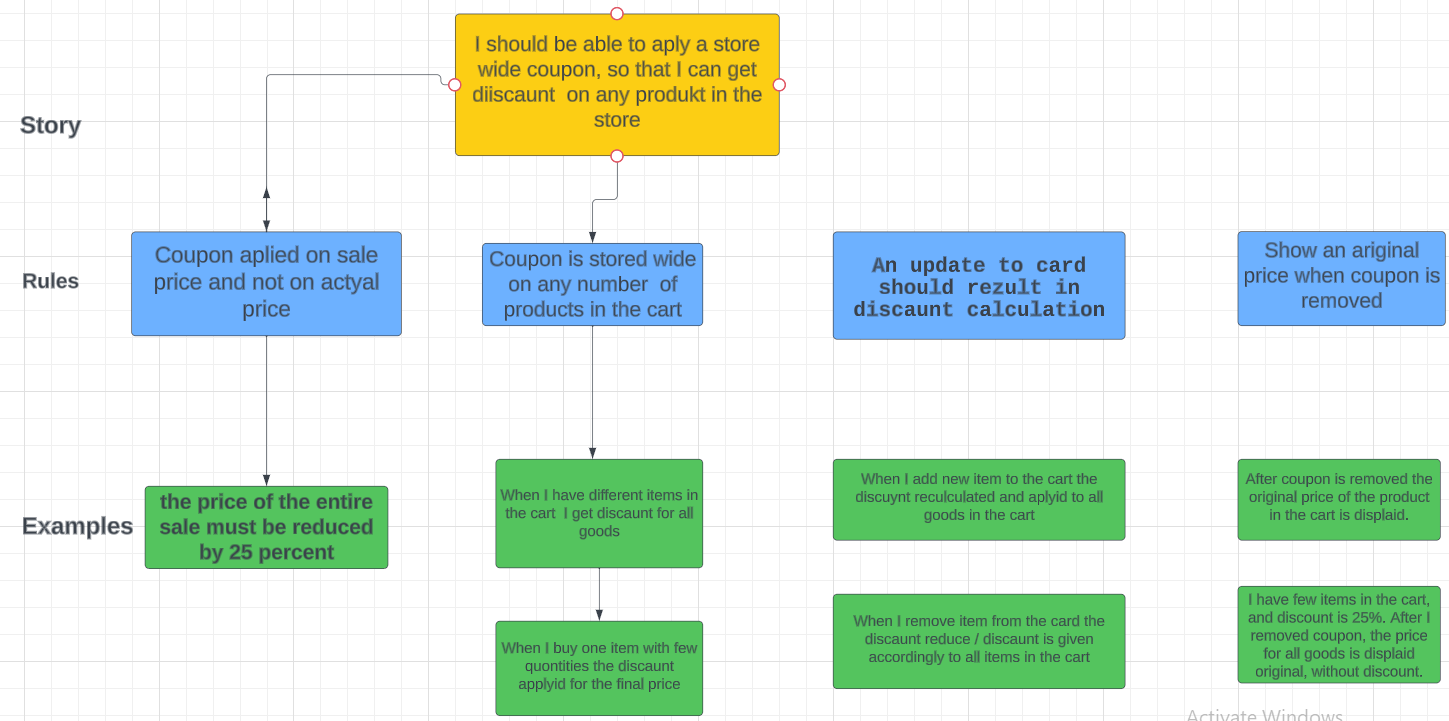
2)- Should be short, no more than 4 steps.

3)-Should be expressive(the business people can read it) and vivid

4)- keep the scenarios short.(so business people can read background section without scrolling)

**Rule**

The purpose is to represent one business rule that should be implemented. The rule provides additional information to the feature, a feature can have multiple rules. A rule can be used to group several scenarios that belong to the particular business rule. So the rule can have multiple scenarios.



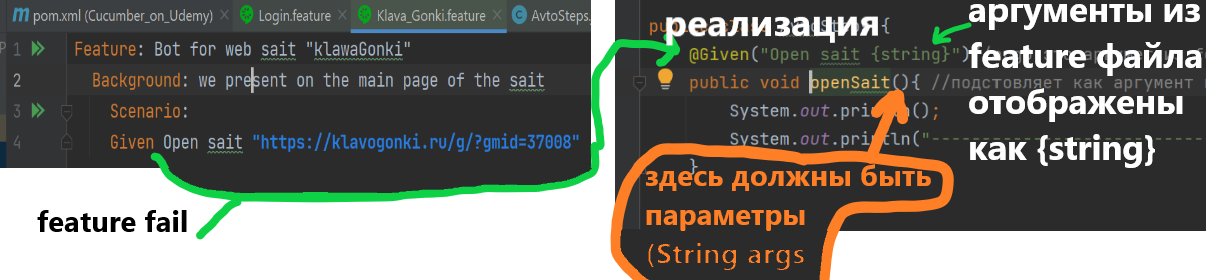
Ryle can have Background step, in case the scenarios in the rule have the same steps the Background is good place to put these steps in that case we can reuse these steps (как это пока не понятно) we don't have to write this steps in each scenario. And you can use this Background for another rule as well. And we can have Background at the feature level that is valid for all the scenarios in all the rules. Rul can have a description.

**аргументы в feature файле**

Также в самом feature файле в отдельно взятом шаге можно вставлять

(которые являются валидными - например URL сайта) которые в дальнейшем будут задействованы в коде.

После написания feature файла надо реализовать каждый шаг в коде(шагом/step называется каждое зарезервированное слово оно обязательное для реализации в коде). Для этого создаётся java класс где под одноимёнными аннотациями ( @Given, @And, @Then…) реализуется логика каждого из шагов. Данные аннотации в скобках имеют описание точно такое же как и в feature файле, благодаря этому происходит как бы склеивания feature файла и метода реализации и если в feature файле на данном шаге есть аргументы они указываются в фигурных скобках реализующей аннотации. Далее в сигнатуре метода данные аргументы должны быть приняты как параметры **иначе ошибка (**io.cucumber.core.exception.CucumberException: Step [Open sait {string}] is defined with 0 parameters**)**



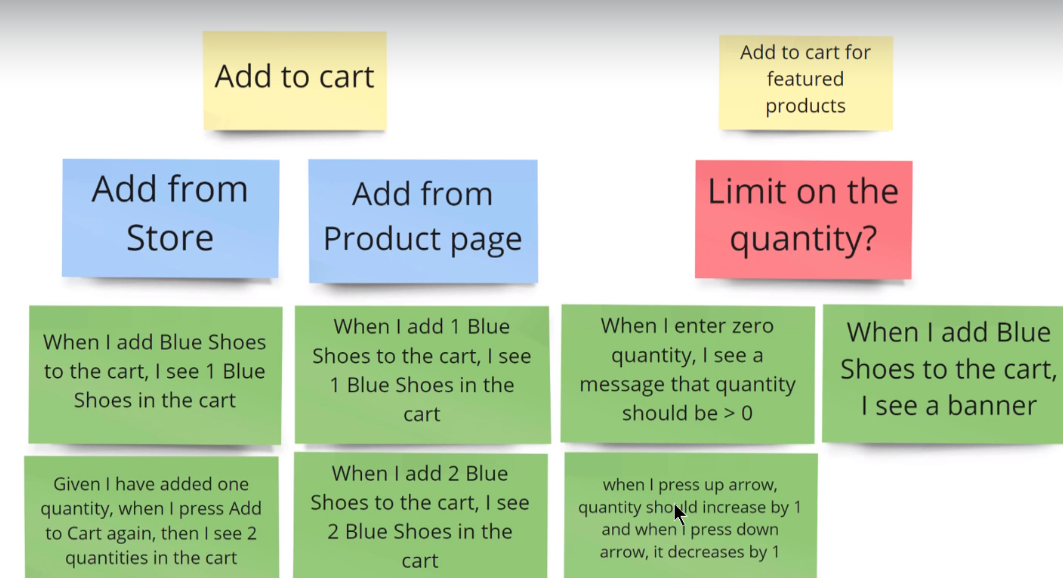
| username | possword | **<-- Alt+ctrl+L красиво редактирует таблицу**

| admin | passwoed\_admin |

**How cucumber knows where is step definition methods**

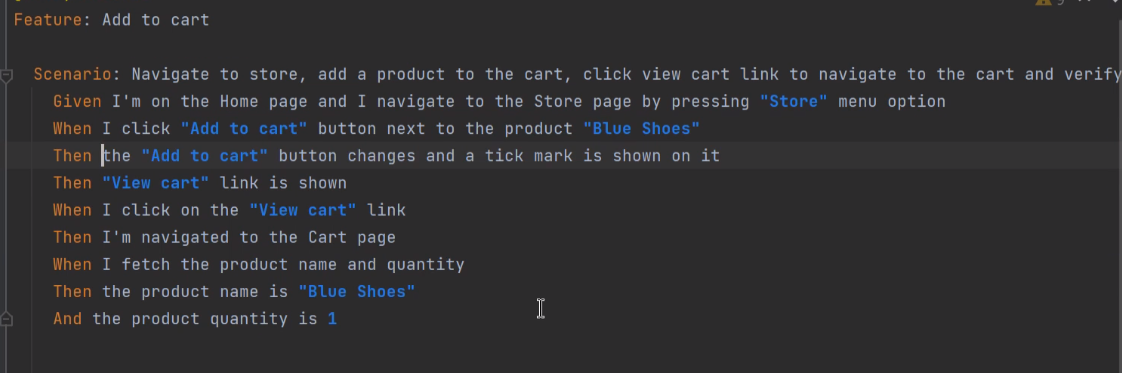
The .java fail, that is step definition of .feature fail, should be in the same root package. If you want to keep it under the different package-try to create package in different directory but with the same name.

**Creating gherkin, practice!**



Let's write gherkin for the first example,(When I add Blue Shoes to the cart, I see 1 blue Shoes in the cart) the context is: I am on the store page.

And that what we have got



**That is not correct**, being from technical people you spend a lot of time reading it, and imagine non technical people! They will find that scenario overwhelming ( слишком большой). So what is wrong with this scenario? 1)-We are putting a lot of implementation details in it, what if some of the implementations change in the future(we are talking about the realization of the app itself)? Then you are going to update this scenario, but ideally, this scenario should be updated only when there is a change in the requirement, not when the implementation changes. We are navigating through a UI: we are pressing on the “store” menu option, we are clicking on the button, we are clicking on the “view cart” link - so we are interacting with the UI. We are just writing the steps that we are performing on the UI. The business would not be necessarily interested in this. They would be interested in requirements, are we representing the requirement in the form of gherkin and whether we are validating it right.

This scenario has got a lot of incidental details (it is something that is not relevant to this scenario)

For example: I am on the home page.

This is not relevant to this scenario. The scenario should start from the store page, because on the rule we have written “Add from store”, that is the context (the context is not that of the user on the home page).

So this scenario is trying to do too many things. We have 4 then steps, basically we are doing 4 different assertions here

1)-we are checking tick mark

2)-we are checking view cart

3)-we are checking that we are navigating to the cart page

4)-we are checking the product name and the quantity.

If the tick mark is not showing on the button then all the scenarios are not going to be executed. But the main intention of these scenario was to validate that the product is added in the cart and we are not validating that because the intermediate step failed. So the scenario doesn't have a single intention. It is doing a lot of things.

The scenario should have one starting point and one ending point. The purpose of this scenario is to validate the functionality that the product should be added to the cart, that's it, and not validate the UI or something else. So it is good practice to avoid multiple Then statements.

So this scenario can be termed as a length scenario. We have 9 steps, and good practice is to have not more than 3-5 steps. If you have more steps than it might be indication that there are multiple scenarios at a play.

Conjunction (соединённые) steps in the Given step actually 2 steps: 1)- I am on the home page 2)-I navigate to the Store page. Both clubbed as a single step. We need to avoid doing such things and by doing so we can reuse steps.

The scenario got a bad name; it is very long and hard to read. For scenario name we can write “add from store” that's it. If you have a description you can use the description below. The name should represent the message what exactly we are validating and not how we are validating.

**Let's write correct scenario**

Feature: Add to a cart

Rule: Add from store

Scenario:Add one quantity to the cart

Given: I am on the store page

When: I add 1 blue shoes to the cart

Then: I see 1 blue shoes in the cart

I write this scenario only in 3 steps,

Given-this is the context that I am on the store page. I don't care how I reach the store page, I can go to the home page, click on the store menu option and reach the store page - but I don't care about that. I can even directly access the store page using a direct URL, that is also fine, but I don't care about that. THIS IS DECLARATION THAT I AM ON THE STORE PAGE - THAT'S IT. How I get there its doesnt matter.

When-this is the action that I am performing, adding the product to the cart.

Then-this is validating the outcome. How I am validating this outcome is not important for this step. What is important - the behavior that I should see -> one quantity of the product. For this step this is happening in the cart page. Cart page can be reused for different scenarios. So this particular step should be written in the way that it can be reused for different scenarios. In different scenarios we might be making the same assertion about the product.