





# Behavior-Driven Development : Cucumber et Gherkin

---

David THIBAU – 2020

david.thibau@gmail.com



# Agenda

---

## **Introduction**

- Tests d'acceptation, BDD et agilité
- Eco-système Cucumber
- Installation et démarrage

## **Fondations Cucumber**

- Syntaxe Gherkin
- Définition des étapes
- Réutilisation, lisibilité
- Tags
- Hooks
- Reporting

## **Cas d'usage**

- Recommandations et pratiques
- API Rest
- Application Web
- Intégration CI/CD



# Introduction

---

**Tests d'acceptation BDD et agilité**

Cucumber  
Installation



# Tests d'acceptation

---

A l'origine : *eXtreme Programming (XP)* et le *Test Driven Development (TDD)*

Avec les tests d'acceptation, le développeur **et** l'expert métier collaborent pour écrire des tests automatisés qui expriment le résultat souhaité par le métier

- Les tests expriment ce que le logiciel doit faire pour que la partie prenante le trouve ***acceptable***
- Les tests commencent par échouer au moment de leur rédaction mais permettent de capter le besoin



# *Behaviour Driven Development*

---

Le **BDD** consiste à développer un langage partagé que toute l'équipe agile s'approprie

=> Améliore la communication et minimise les fausses interprétations

Le BDD s'appuie sur le TDD

=> Écrits avant l'implémentation

Les tests interagissent directement avec le code des développeurs



# Exemple

---

**Fonctionnalité:** Inscription

L'inscription doit être rapide et facile

**Scénario:** Inscription réussie

Les nouveaux utilisateurs doivent recevoir un e-mail de confirmation et être accueillis personnellement par le site une fois connecté

**Etant donné que** J'ai choisi de m'inscrire

**Lorsque** Je m'inscris avec des détails valides

**Alors** Je devrais recevoir un e-mail de confirmation

**Et** Je devrais voir un message d'accueil personnalisé



# Spécifications par l'exemple, exécutable

---

Les tests d'acceptation sont donc des  
**spécifications exécutables**

- Comme les documents de spécifications traditionnels, ils peuvent être rédigés par l'équipe métier
- La spécification se décrit par des exemples concrets qui illustre le besoin
- Ils évoluent avec l'avancée du projet et sont versionnés et committés dans le dépôt de sources (SCM)
- Ils sont intégrés dans les pipelines de CI/CD





# BDD et agilité

---

La BDD assume l'utilisation d'une méthodologie agile.

Le travail est planifié par des petits incréments de fonctionnalité, les ***User Stories***

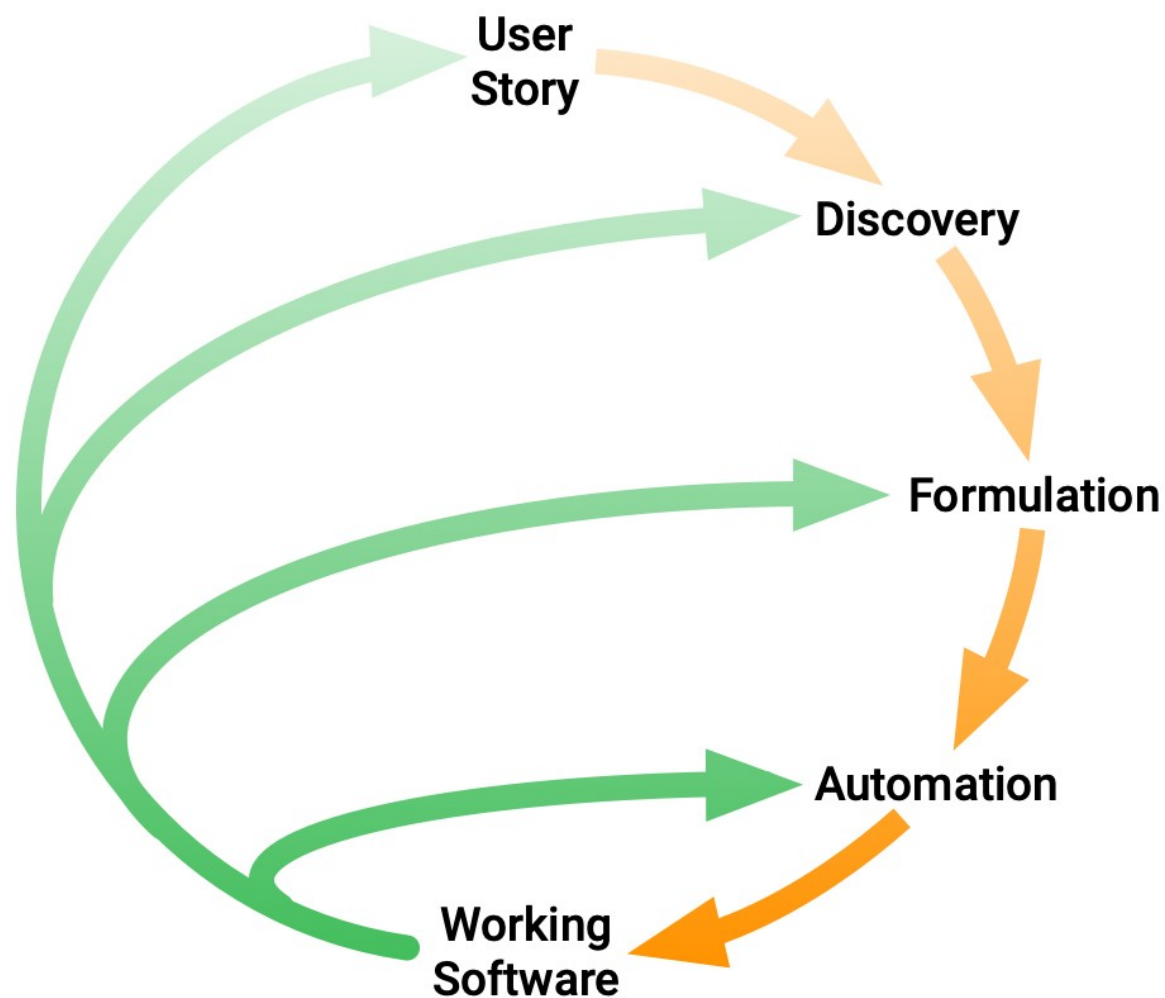
L'activité de BDD consiste à :

1. A partir d'une User Story, parlez d'exemples concrets de la nouvelle fonctionnalité pour explorer, découvrir et convenir des détails de ce qui devrait être fait.
2. Documentez ces exemples de manière automatisable dans la syntaxe Gherkin
3. Implémentez le comportement décrit par chaque exemple



# BDD et Agilité

---





# Ateliers de Découverte

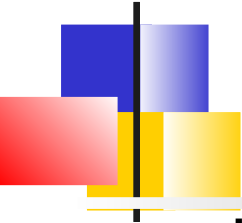
---

La **phase de découverte** consiste en la mise en place d'ateliers se concentrant sur des exemples concrets du système du point de vue des utilisateurs.

Ces ateliers permettent à l'équipe agile de mieux comprendre les besoins des utilisateurs, les règles qui régissent le fonctionnement du système et la portée de ce qui doit être fait.

Cela permet généralement de révéler :

- des lacunes dans notre compréhension nécessitant un approfondissement avant de savoir quoi faire.
- Des prioriser les fonctionnalités relatives à un user story



# Rôle des 3 amigos

---

La transcription des User Stories en spécifications exécutables s'effectue entre :

- Le **Product Owner** : Personne est la plus concernée par la portée de l'application. Traduit les user stories en une série de fonctionnalités et décide de ce qui est dans le champ d'application.
- Les **Testeur** : Génère de nombreux scénarios et de nombreux cas marginaux. Quand l'application part en erreur ? Quelles sont les user story non prises en compte dans les fonctionnalités ?
- Le **Développeur** : Ajoute de nombreuses étapes aux scénarios et réfléchit aux détails de chaque exigence.



# Modèles d'atelier

---

Les ateliers de découverte entre les **3 amigos** (Product Owner, le Testeur et le développeur) sont en général assez courts mais doivent être inclus dans la planification agile

Plusieurs modèles d'ateliers existent :

- L'Example Mapping
- OOPSI Mapping
- Feature Mapping
- ...



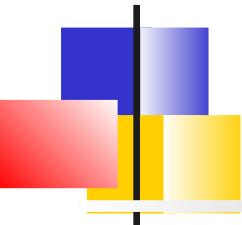
# Boucle itérative

---

Il est préférable que le premier jet d'une fonctionnalité soit rédigé par ou avec un «expert du domaine», non-développeur.

Ensuite, le ou les développeurs passent en revue les scénarios, affinant les étapes de clarification et de testabilité accrue.

- Le résultat est revu par l'expert du domaine pour s'assurer que l'intention n'a pas été compromise.
- Ce cycle est répété jusqu'à ce que tout le monde soient convaincus que les scénarios décrivent avec précision ce qui est souhaité d'une manière testable.



# Atelier : *Example Mapping*

---

- Écrivez la story sur un post-it **jaune**.
- Rédiger les règles métier ou des critères d'acceptation sur des post-its **bleus**.
- Pour chaque règle métier, écrivez des exemples de comportements souhaités et non souhaités sur des post-its **verts**.  
=> Ils deviendront la spécification exécutable
- Des questions vont surgir auxquelles personne dans la salle ne peut répondre en ce moment : écrivez-les sur des post-its **rouges**.

# Example Mapping

## Example

Schedule a team meeting

Meetings with more than 2 NYC attendees need a meeting room.

Remote attendees must have a video meeting link.

Meetings are within normal work hours for all attendees.

A meeting with less than 3 people can be in the Green Sauce meeting room.

A meeting with 2 NYC and one Ukraine people has a zoom meeting number.

NYC and Ukraine team members meet at 9 am EST.

A meeting with 4 people must be in the Bacon Cheeseburger meeting room.

NYC and West Coast team members meet at 3 pm EST.

What if there are more than 12 NYC attendees?

What if we have more than 5 concurrent distributed meetings?

@testacious

Copyright 2016 Lisa Crispin, JoEllen Carter

@lisacrispin





# Atelier : *OOPSI Mapping*

---

*OOPSI Mapping (Outcomes, Outputs, Processes, Scenarios, Inputs)* est une méthodologie convergente basée sur la valeur métier.

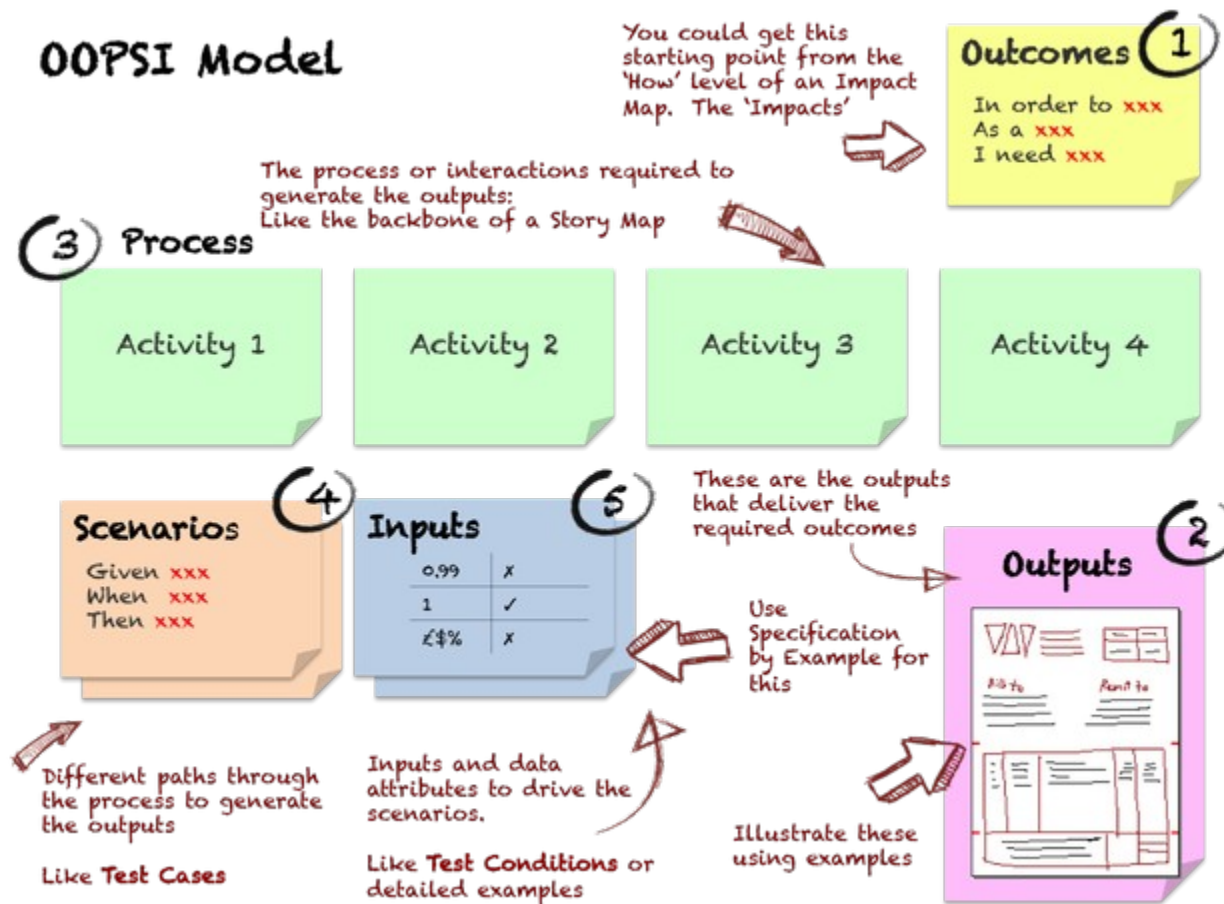
Elle se concentre sur les choses les +importantes

Elle utilise des post-its de différentes couleurs pour associer les processus et les relations entre les sorties du système et les scénarios.

- 1) On sélectionne les user story à la plus haute valeur ajoutée
- 2) Puis les sorties correspondantes les plus significatives
- 3) Les principaux processus pour y arriver
- 4) Les scénarios à plus grande valeur ajoutée
- 5) Les exemples à plus grande valeur ajoutée

# Modèle

## OOPSI Model



# Exemple

## Outcomes (1)

As a customer  
I want to withdraw  
cash  
So that I don't have to  
wait in line at the  
bank

## (3) Process



## Scenarios (4)

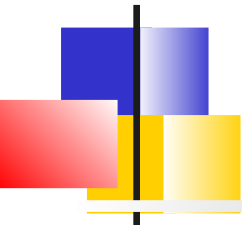
Customer has  
sufficient funds

Customer has  
insufficient funds

Insufficient  
Funds in ATM

## Outputs (2)





# Atelier : *Feature Mapping*

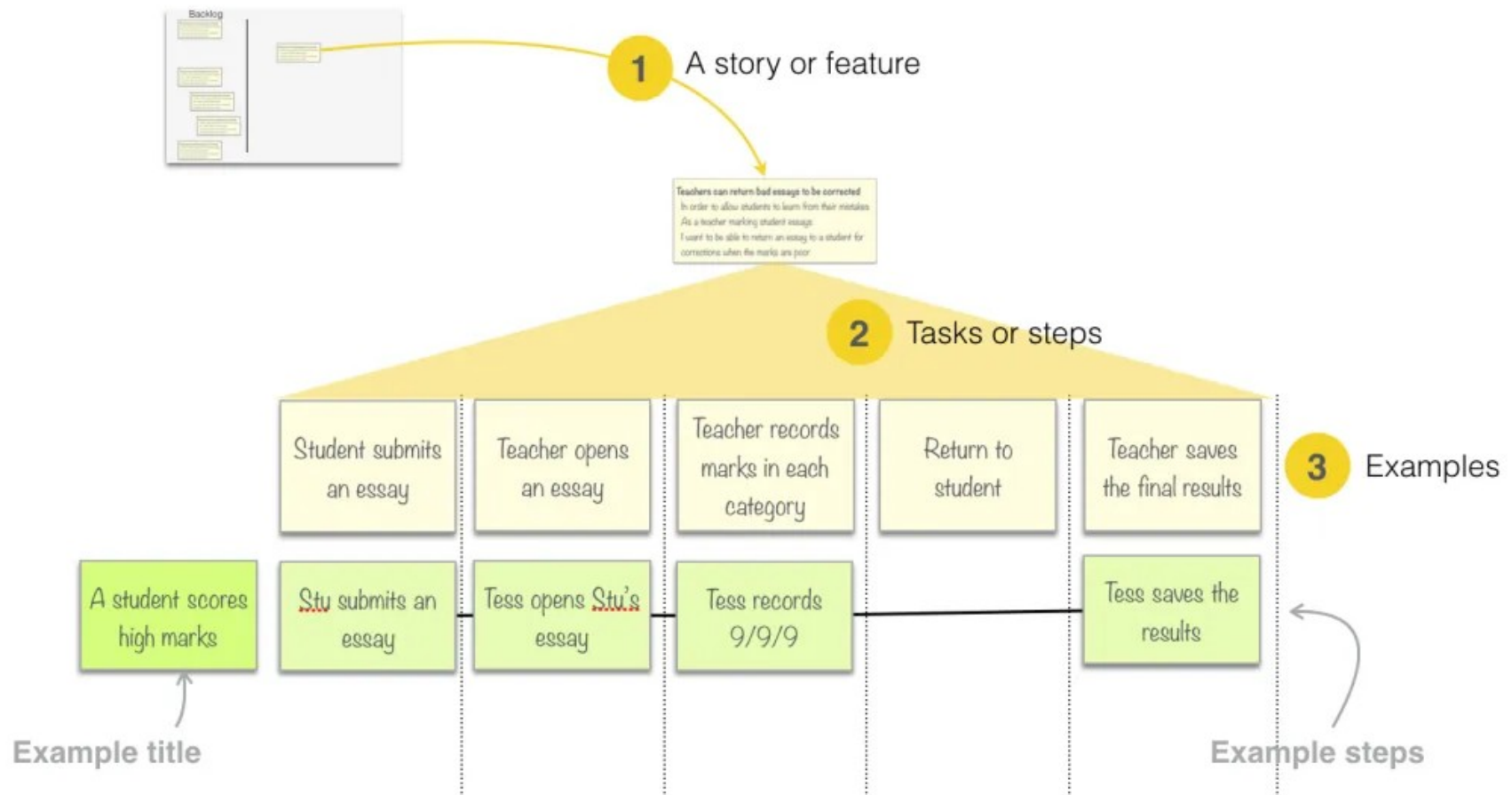
---

Utilise également des post-it de différentes couleurs.

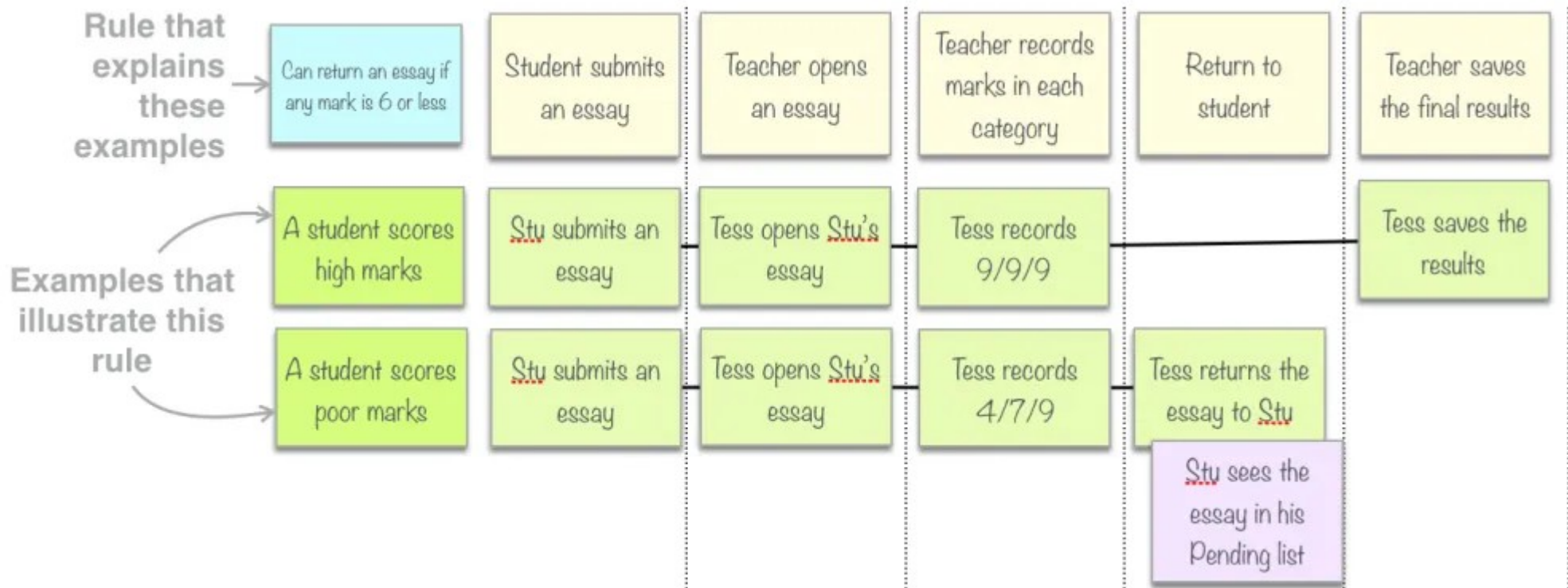
L'équipe :

- 1) Choisit une story dans le backlog,
- 2) Identifie les acteurs impliqués,
- 3) Décompose l'histoire en tâches
- 4) Mappe ces tâches à des exemples spécifiques.

# Tâches puis exemple



# Règles et conséquences





# Recommandations pragmatiques

---

Les exemples BDD sont concrets, ils mentionnent les noms des personnes, des lieux, les dates des montants.

Ils décrivent le comportement prévu du système, pas l'implémentation, i.e le quoi, pas le comment.

Ils ne mentionnent pas les détails techniques.

Les scénarios sont orthogonaux et indépendants

Chaque scénario correspond à 1 seul chemin testé

Les critères d'acceptation évitent de mentionner les interfaces utilisateur, les formats de champ, les boutons, etc.

*Avoir des conversations est plus important que de capturer des conversations qui est plus important que d'automatiser des conversations<sup>1</sup>.*

=> Commencer par les discussions avant d'automatiser les scénarios

**Attention** : Utiliser *Cucumber* ne signifie pas que l'on fait du BDD

=> Ateliers découvertes entre les 3 amigos

1. <https://lizkeogh.com/2014/01/22/using-bdd-with-legacy-systems/>



# Introduction

---

Tests d'acceptation et BDD  
**Eco-système Cucumber**  
Installation et outils





# Historique

---

***Cucumber*** a tout d'abord été développé en Ruby et utilisé exclusivement pour les tests Ruby en complément du framework *RSpec BDD*.

Désormais, *Cucumber* prend en charge une variété de langages de programmation (Java, JavaScript, C++, Kotlin, Go).

Des projets OpenSource tiers effectuent des portages des distributions officiels en d'autres langages



# Vue haut-niveau

---

Lors de son exécution, Cucumber lit les spécifications à partir de fichiers texte en langage simple appelés **fonctionnalités (features)**, y cherche des **scénarios** à tester et les exécute.

Chaque scénario est une liste **d'étapes (steps)** à suivre.

Afin de comprendre les spécifications des fonctionnalités, les fichiers doivent respecter des règles syntaxiques : *Gherkin*

De plus, du **code glue** transformant la syntaxe *Gherkin* en code exécutable doit être fourni.

Le *code glue* est constitué de **définitions d'étapes (step definitions)** correspondant aux étapes utilisées dans les spécifications.



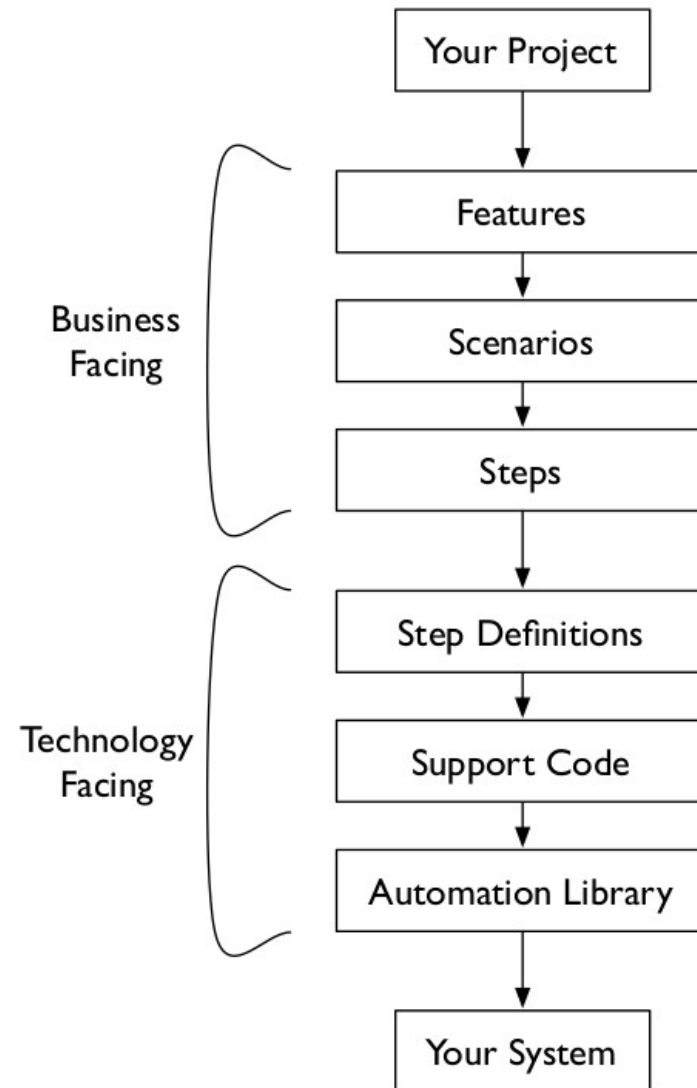
# Architecture

---

Dans une suite de tests idéale, les définitions d'étape ne seront probablement qu'1 ou 2 lignes de code qui délégueront à une bibliothèque de ***code de support***, spécifique au domaine de votre application.

Généralement, cela impliquera l'utilisation d'une ***bibliothèque d'automatisation (automation library)***, comme la bibliothèque d'automatisation du navigateur *Capybara* ou *Selenium*.

# Vue haut-niveau (3)





# Exécution des tests

---

- Si le code de la définition d'étape s'exécute sans erreur, *Cucumber* passe à l'étape suivante du scénario.
- S'il arrive à la fin du scénario sans qu'aucune des étapes ne déclenche une erreur, il marque le scénario comme ayant réussi.
- Si l'une des étapes du scénario échoue, *Cucumber* marque le scénario comme ayant échoué et passe scénario au suivant.
- Au fur et à mesure de l'exécution des scénarios, *Cucumber* affiche les résultats montrant exactement ce qui fonctionne et ce qui ne fonctionne pas.



























# Introduction

---

Tests d'acceptation et BDD  
Eco-système Cucumber  
**Installation et outils**

# Implémentations Cucumber

 <b>Cucumber-JVM</b>   Java official	 <b>Cucumber.js</b>   Node.js and browsers official	 <b>Cucumber.rb</b>   Ruby, Ruby on Rails official
 <b>Cucumber.ml</b>   OCaml official	 <b>Cucumber.cpp</b>   C++ official	 <b>Cucumber-Lua</b>   Lua official
 <b>Android™</b>   Java official	 <b>Kotlin</b>   Cucumber-JVM with Kotlin official	 <b>Cucumber-Tcl</b>   Tcl official
 <b>Godog</b>   Go official	 <b>Behat</b>   PHP semi-official	 <b>Behave</b>   Python semi-official
 <b>SpecFlow</b>   C#, F#, VB.NET semi-official	 <b>Cucumberish</b>   iOS, Swift, ObjC semi-official	 <b>Test::BDD-Cucumber</b>   Perl semi-official
 <b>Cucumber-Rust</b>   Rust unofficial	 <b>unencumbered</b>   D unofficial	 <b>Cucumber-Clojure</b>   Clojure unmaintained
 <b>Cucumber-Gosu</b>   Gosu unmaintained	 <b>Cucumber-Groovy</b>   Groovy unmaintained	 <b>Cucumber-JRuby</b>   JRuby unmaintained
 <b>Cucumber-Jython</b>   Jython unmaintained	 <b>Cucumber-Rhino</b>   Rhino unmaintained	 <b>Cucumber-Scala</b>   Scala unmaintained



# Editeurs de base

---

La plupart des éditeurs font de la coloration syntaxique de Gherkin (même *vi* !!!)

Les éditeurs *Atom*, *TextMate* et *Visual Studio Code* ont des extensions pour *Cucumber*





# Outils Java

---

Plugins pour IntelliJ IDEA, Eclipse

Dépendances Maven :

`io.cucumber : cucumber-java`

Archetype Maven :

`io.cucumber : cucumber-archetype`

Plugins Maven :

*ClueCumber* : Rapports de tests

*Cucable* : Tests en //

Runners :

Junit : `cucumber-junit`

TestNG : `cucumber-testng`

Plugin Gradle

<https://github.com/tsundberg/gradle-cucumber-runner>

Typiquement :

- Fichier *.feature* files dans *src/test/resources*
- Fichier Java de définition de *step* dans *src/test/java*



# Langages dérivés

---

Clojure : cucumber-clojure

Groovy : cucumber-groovy

Ioke : cucumber-ioke

JRuby : cucumber-jruby

Jython : cucumber-jython

Rhino : cucumber-rhino

Scala : cucumber-scala

Java for Android : cucumber-android



# Outils pour autres langages

---

JavaScript : Installation via *npm* ou *yarn*

```
npm install --save-dev cucumber
```

```
yarn add --dev cucumber
```

Ruby : Avec un *fichier Rakefile* approprié

```
rake cucumber
```

Ruby on Rails : simplement

```
rake features
```



# Fichier *rakefile*

---

```
require 'rubygems'
```

```
require 'cucumber'
```

```
require 'cucumber/rake/task'
```

```
Cucumber::Rake::Task.new(:features) do |t|  
  # Option de la ligne de commande.
```

```
  t.cucumber_opts = "--format pretty"
```

```
end
```



# Cucumber JUnit Runner

---

Une classe annotée avec **@RunWith (Cucumber.class)** exécutera les fichiers de fonctionnalités en tant que tests *JUnit*.

Par défaut, elle recherche les fichiers *.feature* et les définitions d'étapes dans le même package ou elle réside.

*Par exemple, si la classe annotée est com.example.RunCucumber, les fonctionnalités et le code glue sont supposés se trouver dans com.example.*

Les options peuvent être fournies par (ordre de priorité):

- Propriétés JVM (Option -D *System.getProperties ()*)
- Propriétés de l'environnement
- Via l'annotation **@CucumberOptions**
- Propriétés définies dans "*cucumber.properties*"



# Options *Cucumber*

---

```
# Utilisation oui ou non des couleurs pour le résultat des tests
cucumber.ansi-colors.disabled
# Vérification que toutes les étapes des feature sont implémentés
cucumber.execution.dry-run
# L'exécution échoue si une étape n'est pas définie
cucumber.execution.strict
# Seulement les features taggées « Work In Progress »
cucumber.execution.wip
# Chemins vers les fichiers de features
cucumber.features
# Filtre sur les noms
cucumber.filter.name
# Expression de tags
cucumber.filter.tags
# Packages du glue code
cucumber.glue
# Liste de Plugins : exemple: pretty,
cucumber.plugin
# Classe de création d'objet (Voir container)
cucumber.object-factory
# underscore ou camelcase pour les snippets
cucumber.snippet-type
```



# Plugins

---

*Cucumber* est basé sur des plugins inclus dans la distribution ou devant être installés.

- Ils sont principalement dédiés au reporting des résultats des tests
- Les plugins peuvent différer selon les langages
- Il est possible d'implémenter son plugin



# Plugins de reporting inclus

---

## Plugins disponibles sans installation

- ***progress*** : Moins verbeux, ne montre que la progression
- ***pretty*** : Format
- ***html***
- ***json***
- ***rerun*** : Ré-exécuter les fails tests
- ***junit*** : Format *xUnit*





# Fondations Cucumber

---

## **Syntaxe Gherkin**

Définitions d'étapes

Réutilisation, lisibilité

Tags

Hooks

Reporting



# Introduction

---

*«La partie la plus difficile dans la construction d'un logiciel est de décider précisément quoi construire.»<sup>1</sup>*

Une technique peut faciliter : l'utilisation d'**exemples concrets** pour illustrer ce que nous voulons que le logiciel fasse

C'est le concept de base de Gherkin



# Expression en langage naturel

---

Pour le métier, les spécifications sont exprimées en langage naturel. Mais pas n'importe comment ...

Comparer :

Les clients doivent être empêchés de saisir des informations de carte de crédit non valides.

Avec :

Si un client saisit un numéro de carte de crédit qui n'a pas exactement 16 chiffres, lorsqu'il essaie de soumettre le formulaire, celui-ci doit être affiché de nouveau avec un message d'erreur l'informant du nombre correct de chiffres.

=> En lisant, la 2ème spéc, le développeur sait quoi coder, l'expert a une meilleure idée de ce que le développeur construit



# Bénéfices des exemples

---

En relisant la spécification, l'expert métier pourrait signaler qu'il existe certains types de cartes qui sont valides avec moins de 16 chiffres et donner alors un autre exemple.

C'est le vrai pouvoir des exemples: ils nous permettent **d'explorer et de découvrir** des cas marginaux que nous aurions pu trouver que bien plus tard.

En donnant un exemple pour illustrer notre exigence, nous avons transformé un critère d'acceptation en un test d'acceptation **exécutable** manuellement ou automatiquement.



# Apport de Gherkin

---

**Gherkin** apporte une structure légère pour documenter des exemples de comportement voulu par le métier

La syntaxe, est comprise à la fois par le métier et à la fois par *Cucumber*.

Son principal objectif reste la lisibilité humaine  
=> Les tests automatisés se lisent comme de la documentation

*Gherkin* est disponible dans de nombreuses langues



# Exemple

---

#language : fr

**Fonctionnalité:** Retour lors de la saisie de détails de carte de crédit invalides

Dans les tests utilisateurs, nous avons vu beaucoup de gens qui ont fait des erreurs en entrant leur n° de carte de crédit. Nous devons être aussi utiles que possible ici pour éviter de perdre des utilisateurs à ce stade crucial de la transaction.

**Contexte:**

Étant donné que j'ai choisi certains articles à acheter

Et que je suis sur le point d'entrer les détails de ma carte de crédit

**Scénario:** numéro de carte de crédit trop court

Lorsque j'entre un numéro de carte de 15 chiffres seulement

Et tous les autres détails sont corrects

Et je soumetts le formulaire

Alors, le formulaire doit être affiché de nouveau

Et je devrais voir un message m'informant du nombre correct de chiffres

**Scénario:** la date d'expiration ne doit pas être antérieure

Lorsque j'entre une date d'expiration de carte qui est dans le passé

Et tous les autres détails sont corrects

Et je soumetts le formulaire

Alors, le formulaire doit être affiché de nouveau

Et je devrais voir un message me disant que la date d'expiration doit être fausse



# Format et syntaxe

---

Les fichiers Gherkin utilisent l'extension **.feature**.

Ils sont enregistrés en texte brut, ils peuvent être lus et modifiés à l'aide d'outils simples.

Gherkin est très similaire aux formats de fichiers tels que *Markdown*, *Textile* et *YAML*.

Chaque fichier commence avec une ligne indiquant la langue (par défaut anglais) puis le mot-clé **Feature** (Fonctionnalité)

```
# language: fr  
Fonctionnalité
```



# Traduction françaises

Anglais	Français
feature	Fonctionnalité
background	Contexte
scenario	scénario
scenarioOutline	Plan du scénario Plan du Scénario
examples	Exemples
given	* Soit Etant donné que Etant donné qu' Etant donné Etant donnée Etant donnés Etant données Étant donné que Étant donné qu' Étant donné Étant donnée Étant donnés Étant données

Anglais	Français
when	* Quand Lorsque Lorsqu'
then	* Alors
and	* Et que Et qu' Et
but	* Mais que Mais qu' Mais





# Feature / Fonctionnalité

---

**Feature/Fonctionnalité** n'affecte pas le comportement des tests

Il permet de spécifier une documentation récapitulative sur le groupe de tests qui suit.

- Le texte sur la même ligne est le nom de la fonctionnalité
- Les lignes restantes sont sa description.

Par convention, le nom de fichier suit le nom de la *Feature*.

Exemple :

**Fonctionnalité**: Ceci est le titre

La description de la fonctionnalité peut tenir sur plusieurs lignes.

Il est possible d'inclure des lignes vides

En fait tout le texte jusqu'au prochain mot-clé Gherkin est inclus dans la description.



# Gabarit de description

---

Un gabarit connu sous le nom ***Feature Injection*** est reconnu comme une bonne façon de décrire une fonctionnalité

In order to <meet some goal>

As a <user role>

I want <a feature>

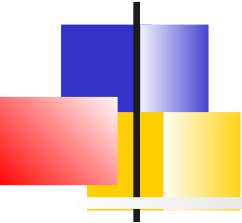


# Mots-clés suivants

---

Les mots-clés suivant *Feature* sont :

- **Rule (v6)**
- **Scenario** ou **Exemple**
- **Background**
- **Scenario outline**



# Scénario / Exemple

---

Chaque fonctionnalité contient plusieurs ***scénarios/exemples***.

- Chaque scénario est un exemple concret unique de la façon dont le système devrait se comporter dans une situation particulière.
- Si on additionne le comportement défini par tous les scénarios, c'est le comportement attendu de la fonctionnalité elle-même.
- Typiquement, chaque fonctionnalité comporte entre 5 et 20 scénarios.



# Gabarit d'un scénario

---

Tous les scénarios suivent le même gabarit :

1. Mettre le système dans un état particulier.
2. Le solliciter
3. Examinez le nouvel état

C'est la suite ***Given, When, Then***



# Exemple

---

**Scénario:** Retrait réussi d'un compte en crédit

**Étant donné que** j'ai 100 \$ dans mon compte **# le contexte**

**Lorsque** je demande 20 \$ **# les événements**

**Alors,** 20 \$ devraient être distribués **# le résultat**

Chaque ligne est une étape

Il est possible d'ajouter plusieurs étapes  
en utilisant les mots-clé **And** ou **But**



# Et, Mais

---

**Scénario:** Tentative de retrait à l'aide d'une carte volée

**Étant donné que** j'ai 100 \$ dans mon compte

**Mais** ma carte n'est pas valide

**Quand** je demande 50 \$

**Alors**, ma carte ne doit pas être retournée

**Et** je devrais être dit de contacter la banque

*Et et Mais* sont plus expressifs mais on pourrait tout aussi bien répéter *Étant donné que* et *Alors*



# Le caractère \*

---

Le caractère \* peut remplacer les mots-clés *Given* , *When* , *Then* , *And* et *But*.

Pour *Cucumber*, c'est la même chose ...  
mais c'est peut être moins expressif

**Scénario:** Tentative de retrait à l'aide d'une carte volée

- \* J'ai 100 \$ sur mon compte
- \* ma carte n'est pas valide
- \* Je demande 50 \$
- \* ma carte ne doit pas être retournée
- \* On devrait me dire de contacter la banque





# Recommandations

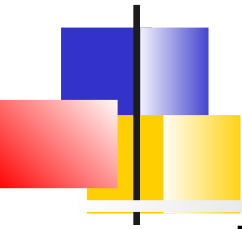
---

Chaque scénario doit avoir un sens et doit pouvoir être exécuté indépendamment de tout autre scénario.

- Autrement dit : stateless
- Toujours supposer que le scénario démarre avec un système tout neuf et re-spécifier toutes les clauses *Given* afin de le placer dans l'état désiré

Les scénarios ont un **nom** et peuvent avoir une **description**.

Choisir le nom avec attention, il apparaîtra dans les rapports de test



# Commentaires

---

En plus des champs de description qui suivent les mots clés *Feature* et *Scenario*, *Cucumber* permet de mettre des commentaires avant ses mots-clés ou à l'intérieur d'un scénario.

Les commentaires commencent par un caractère **#** et doivent être la première et la seule chose sur une ligne



# Fondations Cucumber

---

Syntaxe Gherkin

**Définitions d'étapes**

Réutilisation, lisibilité

Tags

Hooks

Reporting



# Introduction

---

Les **définitions d'étapes** se situent à la frontière entre le domaine métier et le domaine du programmeur.

Ils peuvent être écrits dans de nombreux langages et leur responsabilité est de traduire chaque étape des scénarios Gherkin en actions concrètes dans le code.



# Illustration

---

Considérons l'étape

**Etant donné que** j'ai \$100 sur mon compte

La définition correspondante nécessite d'exécuter 2 choses :

- Créer un compte pour le protagoniste du scénario (si celui n'existe pas encore).
- Positionner la balance du compte à \$100



## Illustration (2)

---

La manière dont les 2 objectifs précédents sont atteints dépend des spécificités de l'application.

- Cela peut impliquer de cliquer sur des boutons sur une interface utilisateur ou appeler directement une couche service ou autre.

Les tests d'acceptation simulent généralement des interactions des utilisateurs avec le système

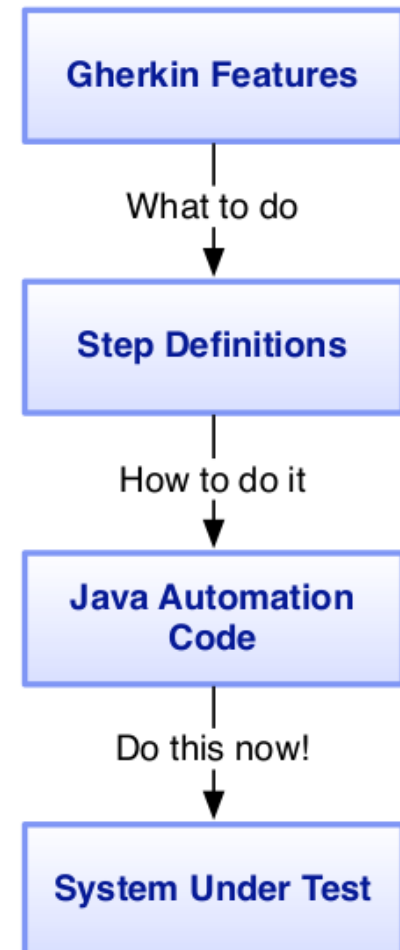
=> Les définitions d'étape permettent d'indiquer à Cucumber comment vous souhaitez que votre système soit testé.



# Architecture

Les définitions d'étapes sont principalement responsables de traduire les features Gherkin.

Elles doivent s'appuyer sur une couche d'automatisation permettant de solliciter le système (Selenium, Client Rest, Client JMS, ...)





# Définition

---

Une définition d'étape est un **fragment de code**<sup>1</sup> avec une **expression**<sup>2</sup> qui la relie à une ou plusieurs étapes Gherkin.

Lorsque *Cucumber* exécute une étape Gherkin dans un scénario, il recherche une définition d'étape correspondante à exécuter.

1. Dans le cas de Java, une méthode
2. Dans le cas de Java une annotation





# Recherche des définitions d'étapes

---

Pour chaque fichier *.feature*, Cucumber scanne tous les fichiers *.java* présent dans le même package

Cucumber scanne le texte de chaque étape à la recherche de *pattern* définis à base d'expression passées aux annotations *@Given*, *@When*, *@Then* :

- Soit une expression Cucumber
- Soit via des expressions régulières



# Illustration

---

Scenario: Some cukes

Given I have 48 cukes in my belly

```
public class StepDefinitions {  
    @Given("I have {int} cukes in my belly")  
    public void i_have_n_cukes_in_my_belly(int cukes) {  
        System.out.format("Cukes: %n\n", cukes);  
    }  
}
```



# Idem avec Lambda

---

Scenario: Some cukes

Given I have 48 cukes in my belly

```
public class StepDefinitions implements En {  
    public StepDefinitions() {  
        Given("I have {int} cukes in my belly",  
            (Integer cukes) -> {  
                System.out.format("Cukes: %n\n", cukes);  
            });  
    }  
}
```



# Expression Cucumber

---

Les expressions Cucumber offrent des fonctionnalités similaires aux expressions régulières, avec une syntaxe plus humaine à lire et à écrire.

- Elles sont également extensibles avec des types de paramètres.
- Elles permettent d'extraire des données de l'étape et de les passer en arguments de méthodes.
- La notation utilisée est les **{ }** en précisant le type de donnée que l'on veut récupérer



# Types de paramètres

---

***{int}*** : Correspond à des entiers : 71 ou -19.

***{float}*** : Des floats 3.6, .8 ou -9.2.

***{word}*** : Un mot (séparation par les espaces)

***{string}*** : Une chaîne de caractère délimitée par des simples ou double quotes .

***{}*** : N'importe quoi (/.\*//).

Les valeurs extraites peuvent être converties en d'autres types d'objets via des ***Object Mapper*** fournis par Cucumber ou par du code de configuration



# Expression régulières

---

**Given** I have deposited \$10 in my Checking Account

```
@Given("I have deposited \\$(\\d+) in my (\\w+) Account")
public void iHaveDeposited$InMyAccount(int amount,
                                         String accountType) {
    // TODO: code goes here
}
```



# Apporter de la flexibilité

---

**Given** I have 1 cucumber in my basket

**Given** I have 256 cucumbers in my basket

**@Given("I have (\\d+) cucumbers? in my basket")**

```
public void iHaveCucumbersInMyBasket(int number) {}
```

---

**When** I visit the homepage

**When** I go to the homepage

**@When("I (?:visit|go to) the homepage")**

```
public void iVisitTheHomepage() {}
```

---

**Given** I have deposited \$100 in my Account from a check my Grandma gave to me

**@Given("I have deposited \\\$(\\d+) in my Account")**

```
public void iHaveDeposited$InMyAccount(int amount) {}
```



# String multi-ligne et liste de valeurs

---

Pour pouvoir de passer des arguments de type String multi-ligne ou liste de valeur, Gherkin propose :

- Les ***Doc Strings*** : Données qui ne tiennent pas sur une seule ligne
- Les ***Data table*** : Liste de valeurs





# Doc Strings

---

Les ***Doc Strings*** sont des textes multi-lignes délimités par `"""`

**Given** a blog post named "Random" with Markdown body  
`"""`

Some Title, Eh?

=====

Here is the first paragraph of my blog post. Lorem  
ipsum dolor sit amet,  
consectetur adipiscing elit.

`"""`



# *Data Table*

Les ***Data Table*** sont pratiques pour passer une liste de valeurs

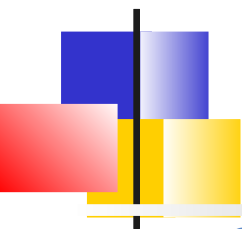
**Given** the following users exist:

name	email	twitter	
Aslak	aslak@cucumber.io	@aslak_hellesoy	
Julien	julien@cucumber.io	@jbppros	
Matt	matt@cucumber.io	@mattwynne	

```
@Given("^the following users exist:$")
public void usersExists(DataTable table) {

    List<Map<String, String>> rows = table.asMaps(String.class, String.class);

    for (Map<String, String> columns : rows) {
        store.addUser(new User(columns.get("name"), columns.get("email"),
columns.get("twitter")));
    }
}
```



# *DataTable* autres exemples

---

**Given** the following users exist:

Aslak	aslak@cucumber.io	@aslak_hellesoy	
Julien	julien@cucumber.io	@jbpros	
Matt	matt@cucumber.io	@mattwynne	

```
@Given("^the following users exist:$")
public void usersExists(DataTable table) {
```

```
    List<List<String>> rows = table.asLists(String.class);
```

```
    for (List<String> columns : rows) {
        store.addUser(new User(columns.get(0), columns.get(1),
columns.get(2)));
    }
```

```
}
```



# *DataTable* autres exemples

---

**Given** the following users exist:

```
| aslak@cucumber.io |  
| julien@cucumber.io |  
| matt@cucumber.io |
```

```
@Given("^the following users exist:$")  
public void usersExists(List<String> emails) {  
  
    for (String email : emails) {  
        store.addUser(new User(email));  
    }  
}
```



# Exception et échecs

---

*Cucumber* utilise les exceptions pour indiquer l'échec d'un test.

- Chaque étape est exécutée séquentiellement.
- Si pour une étape aucune exception n'est lancée et si toutes les assertions sont vérifiées, le test réussit et *Cucumber* passe à l'étape suivante
- Si une exception de type *PendingException* est lancée, l'étape est marquée comme *pending*
- Toutes les autres exceptions provoquent un échec

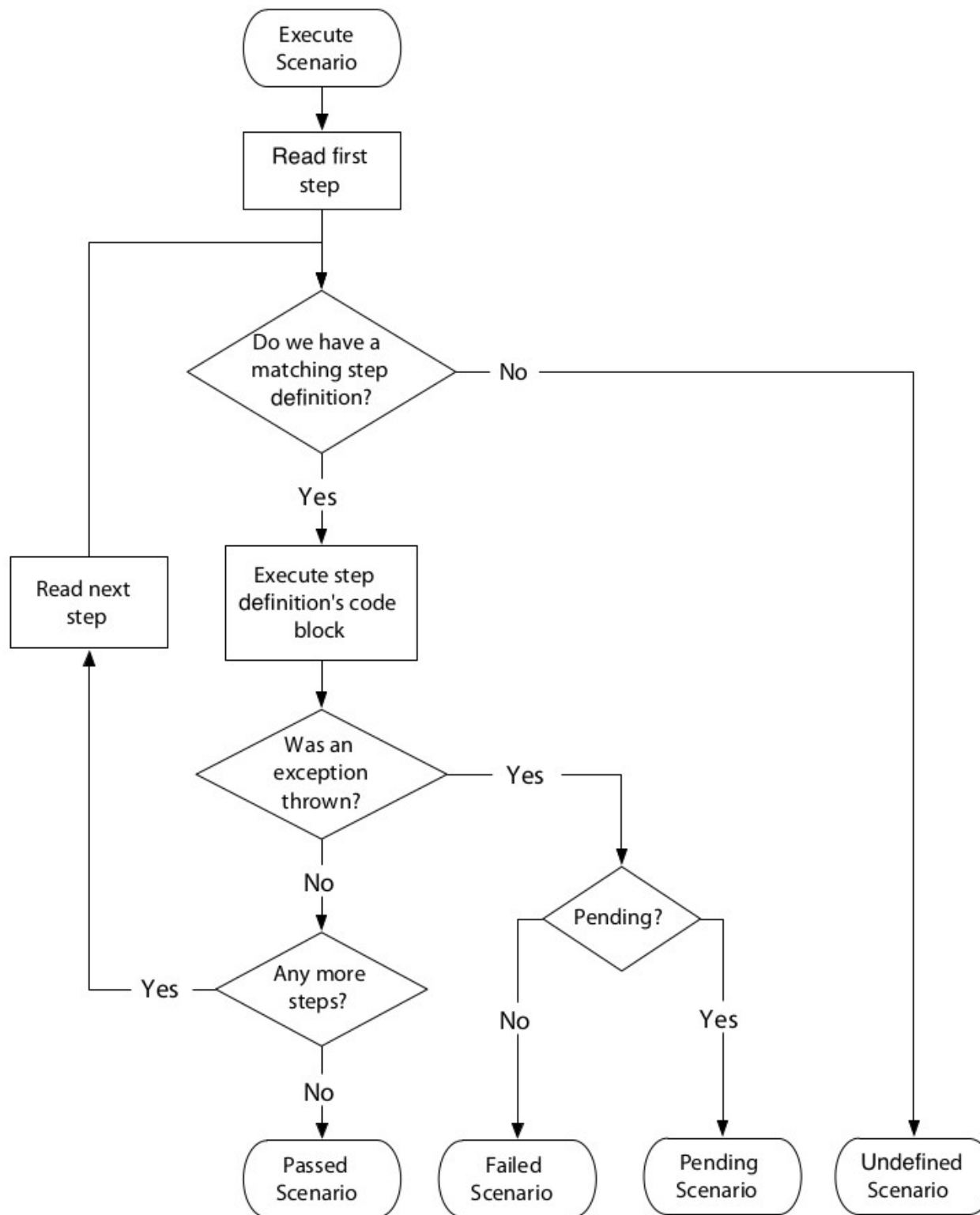
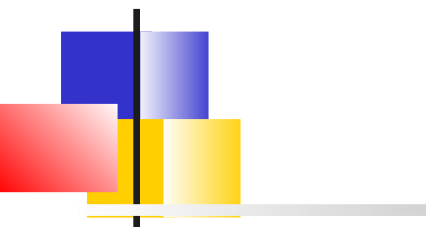


# Statut des étapes

---

Les statuts pour une étape peuvent donc être :

- **Success** : L'étape ne provoque pas d'erreur
- **Undefined** : *Cucumber* n'a pas pu trouver de définition d'étapes
- **Pending** : La définition d'étape a appelé la méthode *pending*
- **Failed** : La définition d'étape a provoqué une erreur
- **Skipped** : Les étapes qui suivent des étapes *undefined*, *pending*, ou *failed*
- **Ambiguous** : Plusieurs définitions d'étapes trouvées





# Utilisation des assertions

---

*Cucumber* n'apporte pas de librairie d'assertion mais il est recommandé d'utiliser les assertions de l'outil de tests (*JUnit* et *TestNG*) dans les clauses *Then/Alors* .

Lorsqu'une assertion n'est pas respectée, une ***AssertionError*** est lancée provoquant l'échec du test

Pour utiliser *JUnit*, il faut ajouter ***cucumber-junit*** dans le classpath

JUnit5 n'est pas encore supporté par Cucumber





# Fondations Cucumber

---

Syntaxe Gherkin  
Définitions d'étapes  
**Réutilisation, lisibilité**  
Tags  
Hooks  
Reporting



# Background/Contexte

---

Une section ***background/contexte*** dans une fonctionnalité permet de spécifier des étapes communes à tous les scénarios du fichier

L'intérêt de ce mot-clé :

- La maintenabilité. 1 seul endroit pour une évolution
- Des scénarios plus clairs

1 seul mot-clé *Background* avant tous les scénarios

On peut indiquer un nom et une description



# Exemple

---

**Fonctionnalité:** Changer de PIN

Afin que les clients puissent changer leur PIN en un code dont ils se puissent se rappeler facilement, il doit être possible de changer de pin en utilisant l'automate

**Contexte:** Insérer une carte et s'authentifier

Lorsque la banque délivre une nouvelle carte, elle fournit au client un Personal Identification Number (PIN) qui est généré aléatoirement.

**Etant donné que** j'ai récupéré une nouvelle carte

**Et que** j'ai inséré la carte et entré un PIN correct

...



# Recommandations

---

- N'utilisez pas *Contexte* pour configurer un état compliqué, sauf si cet état est quelque chose que le lecteur doit réellement savoir
- Gardez votre section *Contexte* courte
- Rendez votre section *Contexte* vivante. Afin que les utilisateurs s'en souviennent facilement
- Gardez vos scénarios courts et n'en n'ayez pas trop. Si le contexte comporte plus de 3,4 étapes, pensez à utiliser des étapes de plus haut-niveau ou à diviser la fonctionnalité en deux.
- Évitez de mettre des détails techniques



# Scenario Outline / Plan du scénario

---

Le mot clé ***Plan du scénario*** permet également d'éviter les répétitions et rend plus lisible la fonctionnalité

Il définit un ensemble d'étapes variabilisées

Le même scénario peut alors être exécuté plusieurs fois pour différentes valeurs



# Exemple

---

**Fonctionnalité:** Retirer un montant fixe

Le menu "Retirer de l'argent" contient plusieurs montants fixes pour accélérer les transactions des utilisateurs.

**Plan du scénario:** Retirer un montant fixe

**Etant donné que** j'ai <Avoir> sur mon compte

**Lorsque** je choisis de retire la somme de <Retrait>

**Alors** je devrais recevoir <Résultat>

**Et** le solde de mon compte devrait être <Restant>

**Exemples:**

Avoir	Retrait	Résultat	Restant
500€	50€	recevoir 50€	450€
500€	100€	recevoir 100€	400€
100€	200€	voir un message d'erreur	100€



# Compléments

---

Il n'y pas de limite sur le nombre de plans de scénario

Pour chaque plan de scénario, doit correspondre un tableau d'exemples

*Cucumber* convertit chaque ligne du tableau *Exemples* en scénario avant de l'exécuter.



# Autre exemple

---

**Fonctionnalité:** Création de compte

**Plan du scénario:** Validation de mot de passe

**Etant donné que** j'essaie de créer un compte avec le mot de passe "<Password>"

**Alors** je devrais voir si le mot de passe est <Valide ou Invalide>

**Exemples:** Trop court

Les mots de passe doivent être supérieurs à 4 caractères

Password	Valide ou Invalide	
abc	invalide	
ab1	invalide	

**Exemples:** Lettres et Nombres

Les mots de passe doivent contenir des lettres et des nombres pour être valides

Password	Valide ou Invalide	
abc1	valide	
abcd	invalide	
abcd1	valide	





# Réutilisation de steps et niveau d'abstraction

---

Différents niveaux de détail sont appropriés pour les différents scénarios dans le même système

Par exemple

Scenario: Successful login with PIN

Given I have pushed my card in the slot

When I enter my PIN

And I press "OK"

Then I should see the main menu

Scenario: Withdraw fixed amount of \$50

Given I have \$500 in my account

**# Reprend le scénario précédent**

**Given I have authenticated with the correct PIN**

When I choose to withdraw the fixed amount of \$50

Then I should receive \$50 cash

And the balance of my account should be \$450



# Implémentation

---

En fonction des langages, différentes techniques seront disponible pour réutiliser des définitions d'étapes :

Ruby :

```
Given(/^I have authenticated with the correct PIN$/) do
  steps %{
    And I have pushed my card into the slot
    And I enter my PIN
    And I press "OK"
  }
End
```

Java :

Appel explicitement des méthodes des steps abstraites



# Rule

---

Le mot-clé optionnel **Rule** est apparu dans la v6 de *Gherkin*. Il n'est pas supporté par toutes les implémentations de *Cucumber*.

- Il a pour objectif de représenter une règle métier à mettre en œuvre.
- Il fournit des informations supplémentaires sur une fonctionnalité.
- Une règle regroupe un ou plusieurs scénarios appartenant à cette règle métier.



# Example

---

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



# Conversion de type

---

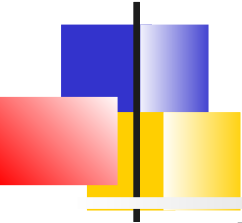
Les expressions *Cucumber* peuvent être étendues pour convertir automatiquement les paramètres de sortie en ses propres types

Un code glue peut définir via l'annotation **@ParameterType** des conversions

Exemple :

**red** is my favorite color

```
@ParameterType("red|blue|yellow") // regexp
public Color color(String color){ // type, name (from method)
    return new Color(color);      // transformer function
}
@Given("{color} is my favorite color")
public void this_is_my_favorite_color(Color color) {
    // step implementation
}
```



# Conversion pour les DataTable et String multi-ligne

---

Pour les *DataTable* et les String multi-lignes, on peut utiliser les annotations :

- **@DataTableType** :  
Utile pour transformer une *DataTable* (Map ou Liste) ou une simple cellule
- **@DocStringType**



# Exemple DataTableType

---

**Given** the following users exist:

name	email	twitter	
Aslak	aslak@cucumber.io	@aslak_hellesoy	
Julien	julien@cucumber.io	@jbpros	
Matt	matt@cucumber.io	@mattwynne	

**@DataTableType**

```
public User userEntry(Map<String, String> entry) {  
    return new User(  
        entry.get("name"),  
        entry.get("email"),  
        entry.get("twitter"));  
}
```

```
@Given("^the following users exist:$")  
public void usersExists(List<User> users) {  
  
    for (User user : users) {  
        store.addUser(user);  
    }  
}
```



# Exemple *DocStringType*

---

```
// Object Mapper de la librairie Jackson
```

```
private static ObjectMapper objectMapper = new ObjectMapper();
```

```
@DocStringType
```

```
public JsonNode json(String docString) throws  
JsonProcessingException {  
    return objectMapper.readValue(docString, JsonNode.class);  
}
```

```
@Given("Books are defined by json")
```

```
public void books_are_defined_by_json(JsonNode books) {  
    // step implementation  
}
```





# Fondations Cucumber

---

Syntaxe Gherkin  
Définitions d'étapes  
Réutilisation, lisibilité  
**Tags**  
Hooks  
Options d'exécution



# Tags

---

Lorsque le nombre de fonctionnalités commencent à être important. Il est possible d'utiliser des sous-dossiers mais également des **tags**

- Chaque fonctionnalité peut être taggés  
=> On peut sélectionner toutes les fonctionnalités d'un certain tag

Les tags utilisent le caractère @



# Exemple

---

**@nightly @slow**

**Feature:** Nightly Reports

**@widgets**

**Scenario:** Generate overnight widgets report

. . .

**@doofers**

**Scenario:** Generate overnight doofers report



# Avantages

---

3 raisons principales pour utiliser les tags :

- Documentation
- Filtrage : *Cucumber* permet de filtrer les scénarios par des tags au moment de l'exécution ou du reporting
- Hooks : Exécuter du code lorsqu'un scénario taggé avec une certaine valeur démarre ou se termine.



# Syntaxe et héritage

---

Les tags peuvent être placés devant

- *Feature*
- *Scenario*
- *Scenario Outline*
- *Examples*

Ils sont hérités par les éléments fils

Pour sélectionner des tags, on utilise des expressions qui permettent de combiner les contraintes sur les tags

Ex : (@smoke or @ui) and (not @slow)



# Restriction d'exécution

---

On peut demander à Cucumber d'exécuter un sous-ensemble de scénarios taggés

- Via un fichier .properties

```
src/test/resources/cucumber.properties
```

- Via une propriété JVM

```
mvn test -Dcucumber.filter.tags="@smoke and @fast"
```

- Via une variable d'environnement

```
# Windows:  
set CUCUMBER_FILTER_TAGS="@smoke and @fast"  
mvn test
```

- Ou par le runner Junit

```
@CucumberOptions(tags = "@smoke and @fast")  
public class RunCucumberTest {}
```

```
@CucumberOptions(tags = "not @smoke")  
public class RunCucumberTest {}
```



# Fondations Cucumber

---

Syntaxe Gherkin  
Définitions d'étapes  
Réutilisation, lisibilité  
Tags  
**Hooks**  
Reporting



# Introduction

---

Les hooks permettent d'exécuter de la logique transverse à différents points du cycle d'exécution de Cucumber

Non visibles dans les features, ils permettent plutôt d'exécuter du code technique.





# Hooks de scénarios

---

Il est possible d'exécuter du code avant chaque scénario

```
@Before
public void doSomethingBefore() {
}
Before(() -> {
})
```

Plutôt privilégier *Background/Contexte*



# Hooks de scénarios

---

Il est possible d'exécuter du code après chaque scénario quelque soit son statut

```
@After
public void doSomethingAfter(Scenario scenario){
    // Do something after after scenario
}
After((Scenario scenario) -> {
});
```

Le paramètre *Scenario* est optionnel, si il est présent le statut du scénario peut être testé



# Hooks d'étapes

---

Ils sont appelés avant et après une étape.

- Ils ont une sémantique "*invoke around*".  
Si un hook **BeforeStep** est exécuté, les hooks **AfterStep** seront également exécutés quel que soit le résultat de l'étape.
- Si une étape n'a pas réussi, l'étape suivante et ses hooks sont ignorés.



# Syntaxe

---

```
@BeforeStep  
public void doSomethingBeforeStep(Scenario scenario){  
}
```

```
BeforeStep((Scenario scenario) -> {  
});
```

```
@AfterStep  
public void doSomethingAfterStep(Scenario scenario){  
}  
AfterStep((Scenario scenario) -> {  
});
```



# Hooks conditionnels

---

L'exécution d'un hooks peut être conditionnée en fonction des tags.

Il suffit d'associer un hook *After* ou *Before* à une expression de tags.

```
@After("@browser and not @headless")
public void doSomethingAfter(Scenario scenario){
}
```

```
After("@browser and not @headless", (Scenario scenario) ->
{
});
```



# Fondations Cucumber

---

Syntaxe Gherkin  
Définitions d'étapes  
Réutilisation, lisibilité  
Tags  
Hooks  
**Reporting**



# Introduction

---

*Cucumber* utilise des plugins pour produire des rapports sur l'exécution des scénarios

- Certains plugins sont intégrés, d'autres doivent être installés séparément.
- *Cucumber* fournit également un service de reporting sur le cloud permettant de partager des rapports au format HTML



# Plugins intégrés

---

*Cucumber* propose les plugins intégrés suivants :

- ***progress*** (défaut) : Rapport minimal avec un seul caractère par étape représentant le statut  
. OK ; U Undefined ; - Skipped ; F Failed
- ***pretty***
- ***html***
- ***json***
- ***rerun*** : N'affiche que les échoués avec le n° de ligne
- ***junit*** : Format Junit (défaut si Runner Junit)





# Activation des plugins

---

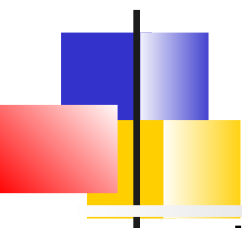
Par défaut, les plugins génère leur sortie sur la sortie standard, il est possible de les rediriger

## Via Junit

```
@RunWith(Cucumber.class)
@CucumberOptions(features = "src/test/resources",
    plugin = { "json:target/cucumber.json", "pretty",
        "html:target/cucumber-reports.html" })
public class CucumberIntegrationTest {
}
```

## Via la commande en ligne

```
java cucumber.api.cli.Main -p pretty -p html:cukes.html
-p rerun:rerun.txt features
```



# Service Cucumber Reports

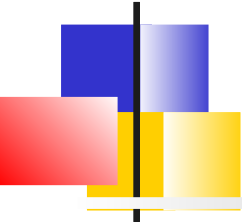
---

Le service est disponible pour les technologies Ruby et Java

Pour l'activer en Java, il suffit de positionner la propriété de configuration ***published*** :

- *src/test/resources/cucumber.properties*:  
cucumber.publish.enabled=true
- Variable d'environnement :  
CUCUMBER\_PUBLISH\_ENABLED=true
- JUnit:  
@CucumberOptions(publish = true)

Les rapports sont supprimés au bout de 24h, il est possible de les conserver si on s'intègre avec un compte Github



# Cas d'usage

---

## **Recommandations et pratiques**

API Rest

Application web

Intégration CI



# Pas de partage d'état entre le scénarios

---

Il est important d'empêcher que l'état créé par un scénario est une influence sur un autre scénario

Pour éviter les fuites d'état entre les scénarios:

- Évitez d'utiliser des variables globales ou statiques.
- Assurez-vous de nettoyer votre base de données dans un hook *@Before*.
- Si on utilise un navigateur, supprimez les cookies dans un hook *@Before*.



# Partage d'état entre les étapes

---

Il est possible de stocker l'état des objets dans des variables à l'intérieur de vos définitions d'étape.

Attention cependant, les étapes couplées sont plus difficiles à réutiliser.



# Frameworks de DI

---

La plupart du temps, on utilise un framework d'injection de dépendance.

Les tests ont souvent besoin d'accéder à des instances d'objet spécifiques à l'application qui doivent également être fournies par l'injecteur. Ces instances doivent être mises à la disposition de vos définitions d'étape afin que des actions puissent y être appliquées et que les résultats fournis puissent être testés.

Il existe donc des bibliothèques d'intégration qui permettent d'utiliser ses frameworks,

Les classes de Cucumber (Step definitions, Hooks, etc..) sont alors également instanciées par le framework



# Cucumber Spring

---

Dépendance :

`io.cucumber:cucumber-spring`

Afin que *Cucumber* récupère le contexte Spring de test, une classe du code glue doit être annotée comme suit :

```
@CucumberContextConfiguration
@SpringBootTest(classes = TestConfig.class)
public class CucumberSpringConfiguration {
```



# Injection

---

Les composants Spring peuvent alors être injectés dans les définitions d'étapes :

```
public class MyStepDefinitions {
```

```
    @Autowired
```

```
    private MyService myService;
```

```
    @Given("feed back is requested from my service")
```

```
    public void feed_back_is_requested(){
```

```
        myService.requestFeedBack();
```

```
    }
```

```
}
```





# Partage d'états entre les étapes

---

Pour empêcher le partage de l'état de test entre les scénarios, les beans contenant du code glue sont liés au scope ***cucumber-glue*** qui commence avant un scénario et se termine après un scénario.

- Tous les beans de ce scope sont supprimés à la fin du scénario

En utilisant l'annotation ***@ScenarioScope***, des composants applicatifs peuvent être utilisés pour partager un état entre des étapes



# Example

---

```
@Component
@ScenarioScope
public class TestUserInformation {
    private User testUser;

    // getter and setter
}

public class UserStepDefinitions {

    @Autowired
    private UserService userService;

    @Autowired
    private TestUserInformation testUserInformation;

    @Given("there is a user")
    public void there_is_as_user() {
        User testUser = userService.createUser();
        testUserInformation.setTestUser(testUser);
    }
}
```



# Example (2)

---

```
public class PurchaseStepDefinitions {  
  
    @Autowired  
    private PurchaseService purchaseService;  
  
    @Autowired  
    private TestUserInformation testUserInformation;  
  
    @When("the user makes a purchase")  
    public void the_user_makes_a_purchase(){  
        Order order = ....  
        User user = testUserInformation.getTestUser();  
        purchaseService.purchase(user, order);  
    }  
}
```



# Base de données

---

Différentes alternatives pour réinitialiser la base entre les scénarios

- @Before Hook
- Roll-back de transactions
- Utilisation de base de données embarquées



# Mock

---

Il n'est en général pas nécessaire d'utiliser des mocks lorsque l'on travaille avec *Cucumber*

- En général, c'est l'intégralité du système que l'on veut tester

Cependant, si notre système dépend d'un service externe ou si l'on applique les tests BDD à un sous-ensemble du système, il peut être utilisé d'utiliser des framework de mocking :

- Java : Mockito, MockServer, WireMock
- Ruby : Rspec 2.X



# Cas d'usage

---

Recommandations et pratiques

**API Rest**

Application web

Intégration CI



# Introduction

---

Dans le cas d'une API REST,  
l'automatisation des scénarios nécessite  
que les définition d'étapes accèdent au  
service Web comme un client http

Cela nécessite de fournir des classes  
utilitaires permettant d'effectuer des  
requêtes REST

- Certains outils facilitent cette intégration



# Exemple de feature

---

**Feature:** Fruit list

To make a great smoothie, I need some fruit

**Scenario:** List fruit

**Given** the system knows about the following fruit:

```
| name
| color |
| banana
| yellow |
|strawberry | red
|
```

**When** the client requests GET /fruits

**Then** the response should be JSON:

```
""
[ {"name": "banana",
  "color": "yellow"},
  {"name": "strawberry", "color": "red"}
]
""
```





# Outils

---

On peut utiliser les apis de bas niveau pour exécuter ses requêtes HTTP et vérifier les réponses

- *HttpURLConnection* standard.
- Bibliothèque Apache *HttpClient*.
- *RestTemplate* Spring

Quelques outils se sont spécialisés sur cette fonctionnalité

- *RestAssured*
- *Karate*



# Exemple *RestAssured*

---

```
@Test public void
lotto_resource_returns_200_with_expected_id_and_winners() {

    when().
        get("/lotto/{id}", 5).
    then().
        statusCode(200).
        body("lotto.lottoId", equalTo(5),
            "lotto.winners.winnerId", hasItems(23, 54));

}
```



# Exemple Karate

---

**Scenario:** create and retrieve a cat

**Given** url 'http://myhost.com/v1/cats'

**And** request { name: 'Billie' }

**When** method **post**

**Then** status 201

**And** match response == { id: '#notnull', name: 'Billie' }

**Given** path **response.id**

**When** method **get**

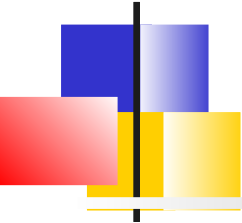
**Then** status 200

JSON is 'native'  
to the syntax

Intuitive DSL  
for HTTP

Payload  
assertion in  
one line

Second HTTP  
call using  
response data



# Cas d'usage

---

Recommandations et pratiques

API Rest

**Application web**

Intégration CI



# Introduction

---

Dans le cas d'une application web, l'automatisation des scénarios nécessite que la définition d'étapes simule les interactions utilisateur avec l'UI

Cela nécessite de s'équiper d'outils de tests fonctionnels (Selenium, HttpUnit, Serenity BDD, Protractor, ...)

Cucumber permet alors l'écriture et la compréhension des tests fonctionnels par le métier.



# Intégration Selenium

---

L'intégration Selenium (Java) consiste à :

- Déclarer la dépendance Maven
- Télécharger le ou les exécutables pouvant piloter les navigateurs
- Instancier un WebDriver dans les steps Definitions



# Dépendance Maven

---

```
<dependency>
  <groupId>org.seleniumhq.selenium</groupId>
  <artifactId>selenium-java</artifactId>
  <version>3.141.59</version>
</dependency>
```

## Installation du Driver

- Téléchargement Driver  
*[https://www.selenium.dev/documentation/fr/webdriver/driver\\_requirements/](https://www.selenium.dev/documentation/fr/webdriver/driver_requirements/)*
- Ensuite positionner le driver dans le PATH ou indiquer des propriétés *webdriver.chrome.driver*, *webdriver.geckodriver.driver*, ...



# Example

---

```
public class ExampleSteps {

    private final WebDriver driver = new FirefoxDriver();

    @Given("I am on the Google search page")
    public void I_visit_google() {
        driver.get("https://www.google.com");
    }

    @When("I search for {string}")
    public void search_for(String query) {
        WebElement element = driver.findElement(By.name("q"));
        element.sendKeys(query);
        element.submit();
    }

    @Then("the page title should start with {string}")
    public void checkTitle(String titleStartsWith) {
        new WebDriverWait(driver, 10L).until(new ExpectedCondition<Boolean>() {
            public Boolean apply(WebDriver d) {
                return d.getTitle().toLowerCase().startsWith(titleStartsWith);
            }
        });
    }

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





# Différents navigateurs

---

Les tests peuvent être effectués avec différents navigateur à partir d'une propriété de configuration.

```
public class WebDriverFactory {  
    public static WebDriver createWebDriver() {  
        String webdriver = System.getProperty("browser", "firefox");  
        switch(webdriver) {  
            case "firefox":  
                return new FirefoxDriver();  
            case "chrome":  
                return new ChromeDriver();  
            default:  
                throw new RuntimeException("Unsupported webdriver: " + webdriver);  
        }  
    }  
}  
  
mvn test -Dbrowser=chrome
```



# Copie d'écran embarquée

---

Il est possible d'associer une capture d'écran lorsqu'un scénario échoue.

Il suffit de configurer un hook after.

Exemple Selenium :

```
if (scenario.isFailed()) {  
    byte[] screenshot =  
        webDriver.getScreenshotAs(OutputType.BYTES);  
    scenario.embed(screenshot, "image/png");  
}
```



# Cas d'usage

---

Recommandations et pratiques  
API Rest  
Application web  
**Intégration CI**



# Introduction

---

Les tests d'acceptation valident une spécification, il est naturel de les intégrer dans une pipeline de CI/CD

Ils sont généralement exécutés en fin de pipeline (avant la production d'une release par exemple)

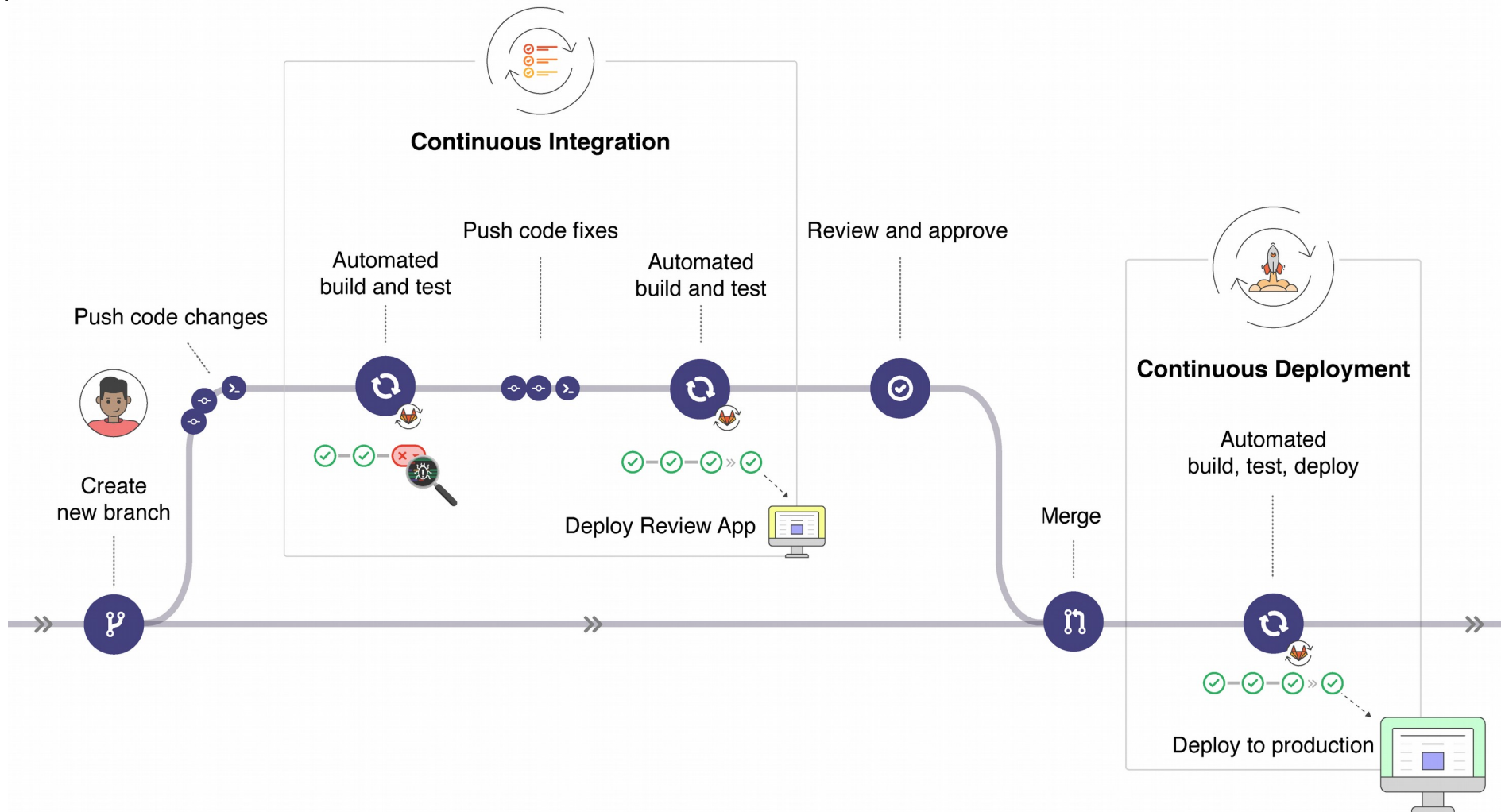
- Ils nécessitent le déploiement de l'application (en intégration, pré-production)
- Ils peuvent être longs

Leur échec interrompt la pipeline

La PIC peut historiser les résultats des builds pour afficher des graphiques de tendance

# Pipeline typique CI/CD

(AutoDevOps Gitlab CI)





# Intégration avec la PIC

---

L'intégration avec la PIC consiste à démarrer les tests par la ligne de commande :

- Démarrage du programme principal en ligne de commande
- Intégration avec framework de test JUnit, TestNG
- Intégration avec l'outil de build (Ant, Maven, Gradle)



# Options ligne de commande

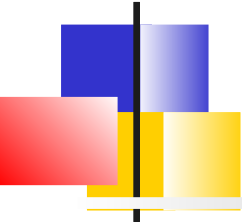
---

**java cucumber.api.cli.Main --help**

Usage: java cucumber.api.cli.Main [options] [ [FILE|DIR]  
[:LINE[:LINE]\*] ]+

Options:

- g, --glue PATH : Emplacement du code glue.
- p, --plugin PLUGIN[:PATH\_OR\_URL] : Enregistrement d'un plugin .
- t, --tags TAG\_EXPRESSION:limte à des tags
- n, --name REGEXP : Filtre des scénarios sur expression régulières
- d, --[no-]-dry-run : Ne pas exécuter le code glue
- s, --[no-]-strict : Traite les étapes indéfinies et en cours comme des erreurs
- i18n LANG: Langue



# Intégration Junit, TestNG

---

## JUnit

```
@RunWith(Cucumber.class)
@CucumberOptions(plugin={"pretty", "html:out.html"}, glue="nicebank",
features ="src/test/resources/nicebank")
public class RunCukesTest {
}
```

```
java -cp ./usr/share/java/junit.jar org.junit.runner.JUnitCore [test class name]
```

## TestNG

```
@CucumberOptions(plugin = "json:target/cucumber-report.json")
public class RunCukesTest extends AbstractTestNGCucumberTests {
}
```

```
java -cp "path-tojar/testng.jar:path_to_yourtest_classes" org.testng.TestNG
testng.xml
```





# Intégration Maven

---

Le plus simple est d'exécuter les tests *Cucumber* en même temps que les tests unitaires via le plugin *surefire*.

- Mais cela ne permet pas de distinguer TU et TA

Si l'on veut séparer les tests d'acceptation des TUs

- Utiliser le plugin *Maven Failsafe* et la phase tests d'intégration
- Lancer *Cucumber* via la commande en ligne



# Maven FailSafe

---

Placer les tests Cucumber et les ressources dans *src/acceptance*

```
<profiles>
  <profile>
    <id>acceptance-tests</id>
    <build>
      <plugins><plugin>
        <artifactId>maven-failsafe-plugin</artifactId>
        <executions> <execution>
          <goals>
            <goal>integration-test</goal>
            <goal>verify</goal>
          </goals>
        </execution> </executions>
      </plugin> </plugins>
    </build>
  </profile>
</profiles>
```



# Ajout des répertoires spécifiques

---

```
<groupId>org.codehaus.mojo</groupId>
<artifactId>build-helper-maven-plugin</artifactId>
<executions> <execution>
  <id>add-source</id>
  <phase>generate-sources</phase>
  <goals> <goal>add-test-source</goal> </goals>
  <configuration>
    <sources> <source>src/acceptance/java</source> </sources>
  </configuration>
</execution>
<execution>
  <id>add-resource</id>
  <phase>generate-sources</phase>
  <goals> <goal>add-test-resource</goal> </goals>
  <configuration>
    <resources> <resource>
      <directory>src/acceptance/resources</directory>
    </resource> </resources>
  </configuration>
</execution> </executions>
```

=> mvn -Pacceptance-tests integration-test



# Exécution Cucumber via Maven

---

```
<plugin>
<groupId>org.codehaus.mojo</groupId>
<artifactId>exec-maven-plugin</artifactId>
<executions><execution>
<phase>integration-test</phase>
<goals><goal>java</goal></goals>
<configuration>
<classpathScope>test</classpathScope>
<mainClass>cucumber.cli.Main</mainClass>
<arguments>
<argument>- -plugin</argument>
<argument>pretty</argument>
<argument>- -glue</argument>
<argument>nicebank</argument>
<argument>src/test/resources/nicebank</argument>
</arguments>
</configuration>
</executions></execution>
</plugin>
```



# Partager les rapports

---

Dans un contexte de CI, il est intéressant de conserver et partager les rapports des différents build

- Reporting Services de *Cucumber*  
Possibilité d'ajouter un lien vers la plateforme de CI : CircleCI, Github, Gitlab CI, ...
- Plugin Jenkins :  
<https://plugins.jenkins.io/cucumber-reports/>



# Accélération des tests

---

Pour accélérer l'exécution des tests et réduire le temps d'exécution de la pipeline, il est possible d'utiliser le parallélisme

En java, une thread est utilisé pour chaque fonctionnalité  
=> Tous les scénarios d'une même fonctionnalité sont utilisés par la même thread



# Configuration

---

**< !-- Commande en ligne -->**

```
java -cp ... io.cucumber.core.cli.Main --threads 4 -g parallel parallel
```

**< !-- Plugin Maven failsafe -->**

```
<plugin>
  <groupId>org.apache.maven.plugins</groupId>
  <artifactId>maven-failsafe-plugin</artifactId>
  <version>3.0.0-M5</version>
  <executions>
    <execution>
      <goals>
        <goal>integration-test</goal>
        <goal>verify</goal>
      </goals>
      <configuration>
        <parallel>methods</parallel>
        <useUnlimitedThreads>true</useUnlimitedThreads>
      </configuration>
    </execution>
  </executions>
</plugin>
```



# Introduction (1)

---

<https://github.com/eugenp/tutorials/tree/master/spring-cucumber>

<https://www.baeldung.com/cucumber-rest-api-testing>

<https://www.guru99.com/using-cucumber-selenium.html>