Programmation C avancée

Concepts & Outils
pour le développement

maj 01/2023

Intégration Continue

L'Intégration Continue

- Lors que l'on développe, on est sur un système particulier :
  - type du système (unix, linux, windows, macosx, etc.)
  - version du compilateur
  - version des bibliothèques
  - environnement général (ressources…)
- Avant de transmettre une modification (commit), le développeur doit s'assurer que ses modifications fonctionnent pour l'ensemble des systèmes/configurations cibles.

L'Intégration Continue

- Pour cela, on dispose d'un ensemble de machines.
  - Solution 1 : avant de transmettre mes modifications, je me connecte sur chacune des machines et je teste.
  - Solution 2 : j'utilise un système qui fait cela automatiquement pour moi !

    ⇒ C'est ce que l'on appelle l'Intégration Continue.

  - l'IC *garantie* une « stabilité » des développements au fur et à mesure. Cela permet de contrôler certaines dettes techniques.

L'Intégration Continue

- Il existe plusieurs plateformes d'intégration continue :
  - Jenkins (successeur de Hudson, java-open source)
  - TeamCity (JetBrain, commercial)
  - CruiseControl (java-open source)
  - Team Foundation Server (Microsoft, commercial)
  - Travis IC (online IC for github projects).
  - ...

L'Intégration Continue

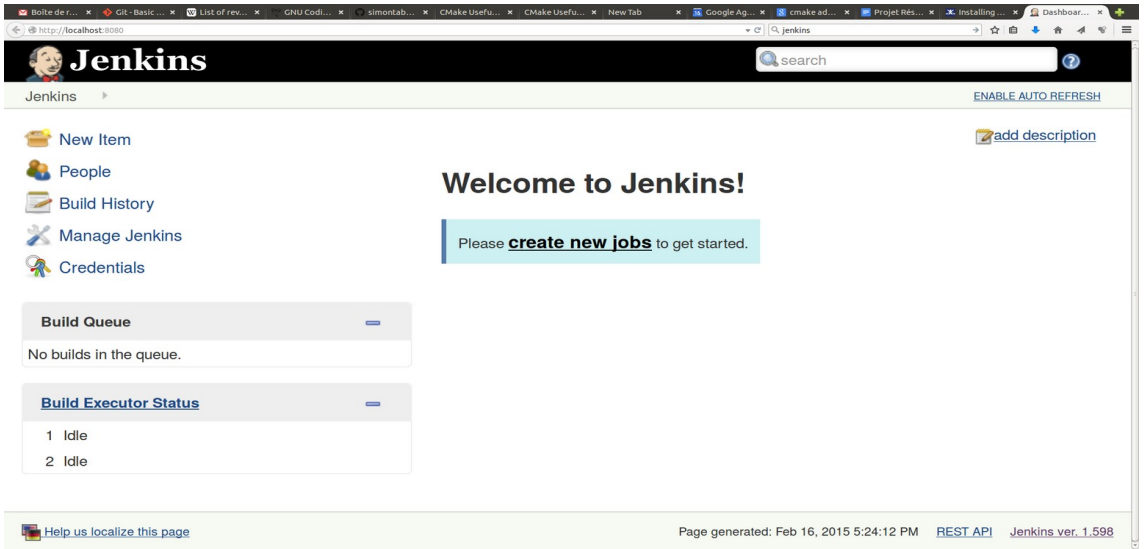
- Un serveur d'intégration continue va se synchroniser avec un dépôt
- A intervalle régulier, il va vérifier que le dépôt est à jour. Si une mise à jour est intervenue, il va effectuer une série de taches (compilation par exemple).
- En fonction du résultat des taches, le serveur va indiquer l'état du projet et possiblement transmettre des alertes.
- Afin de gérer plusieurs environnements, le serveur d'IC va piloter des clients sur lesquels il lancera les taches (via ssh par exemple).

L'Intégration Continue

- La qualité qu'offre l'IC va dépendre principalement de deux facteurs :
  - la nature et la diversité des clients (environnement de validation)
  - la complexité des taches à réaliser :
    - de la compilation
    - à l'exécution de taches complexes de validation
- Le serveur d'IC peut également rendre compte de facteurs comme les ressources utilisées (cpu, temps, mémoire).

Exemple avec Jenkins

- répertoire projet :
  - projet/ :
    - makefile
    - main.c
- installation de jenkins et connexion au port 8080 sur localhost :



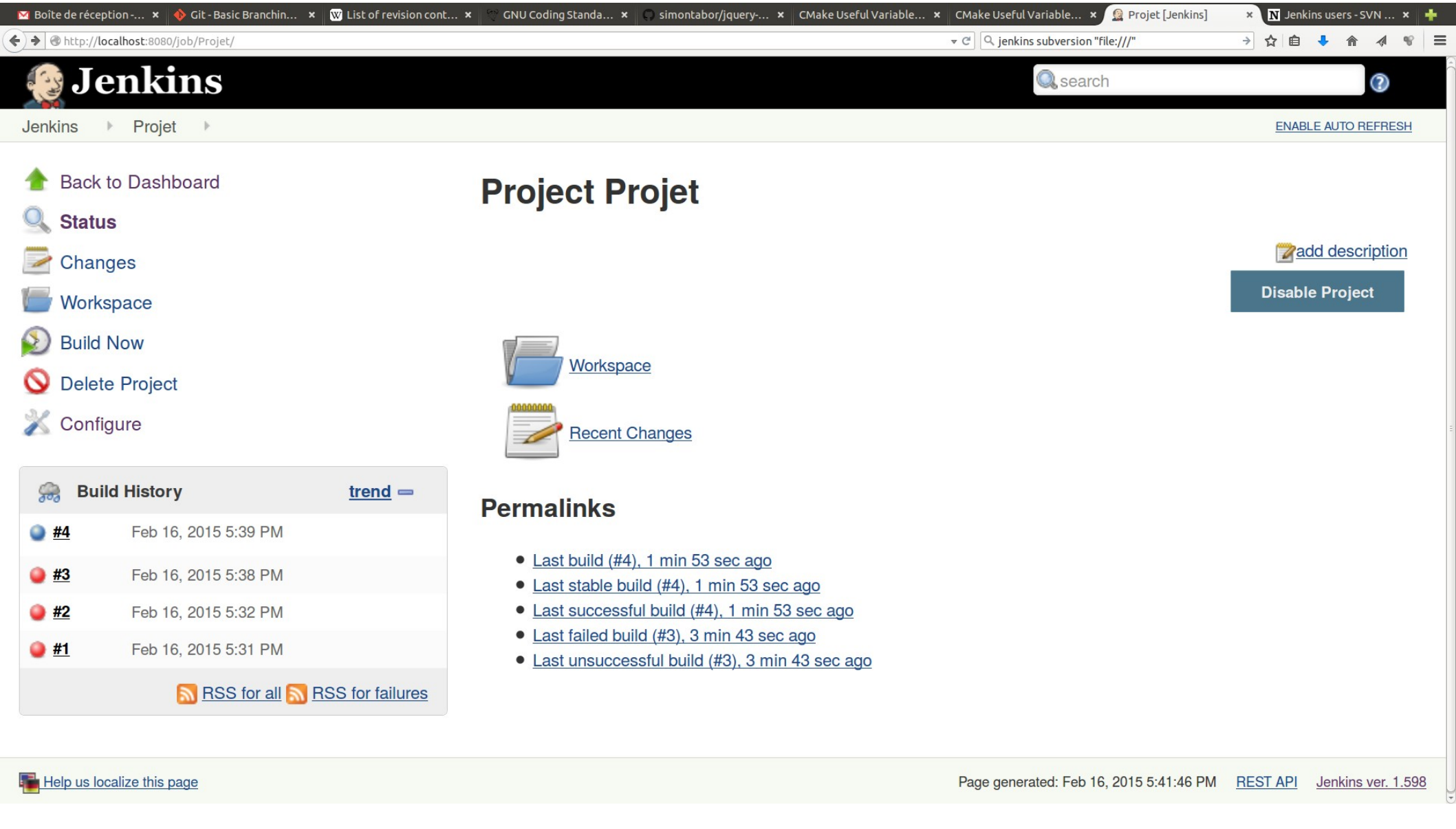
Jenkins : exemple

- création d'un dépôt local avec les sources :

  svnadmin create /tmp/projet

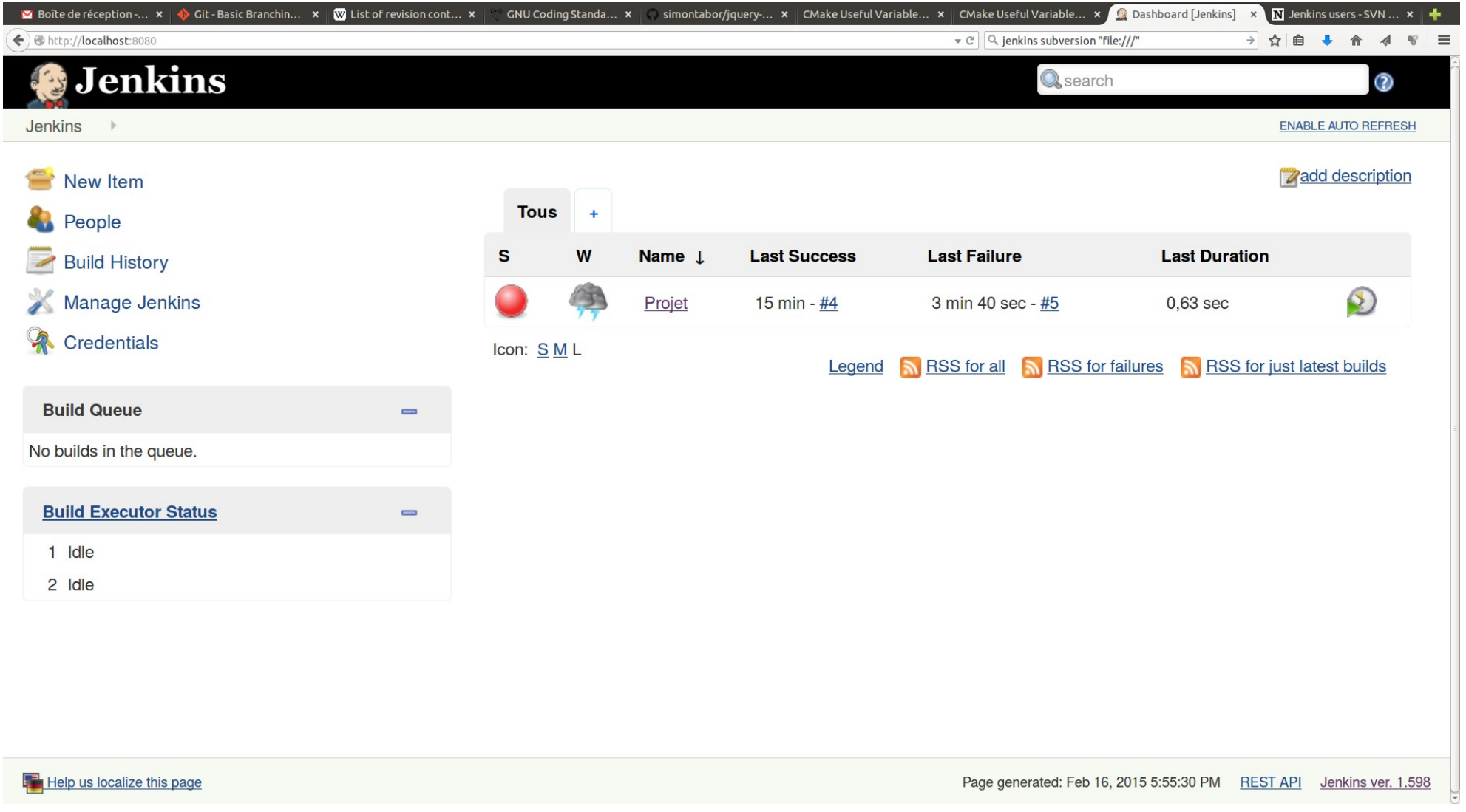
  checkout + ajout des sources + commit
- Ajout du dépôt dans Jenkins en utilisant comme url *svn+ssh://localhost/tmp/projet/*
- Ajout comme commande de build : make
- Lancement d'un build dans jenkins

Jenkins : statut du projet

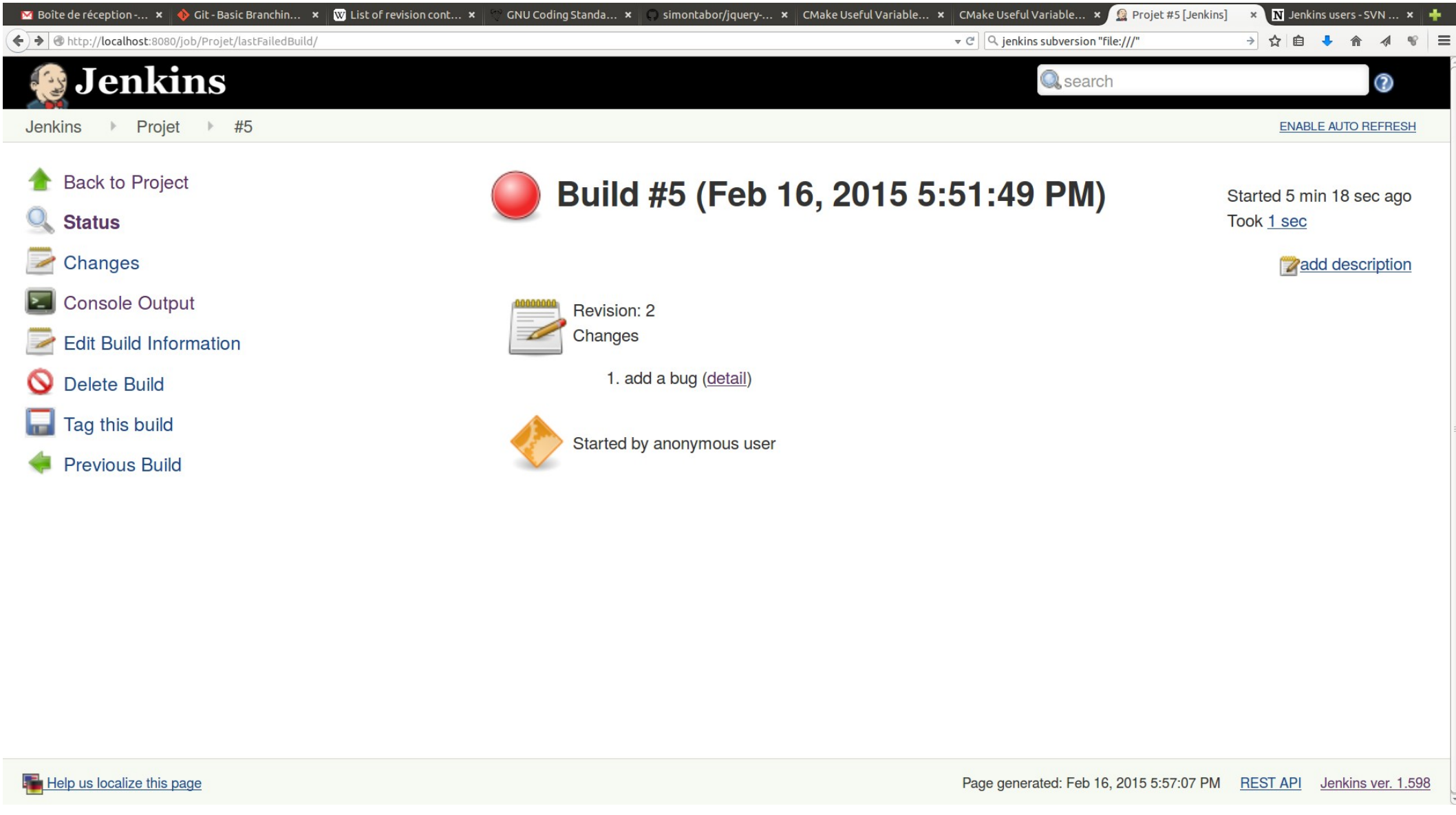


Jenkins : commit

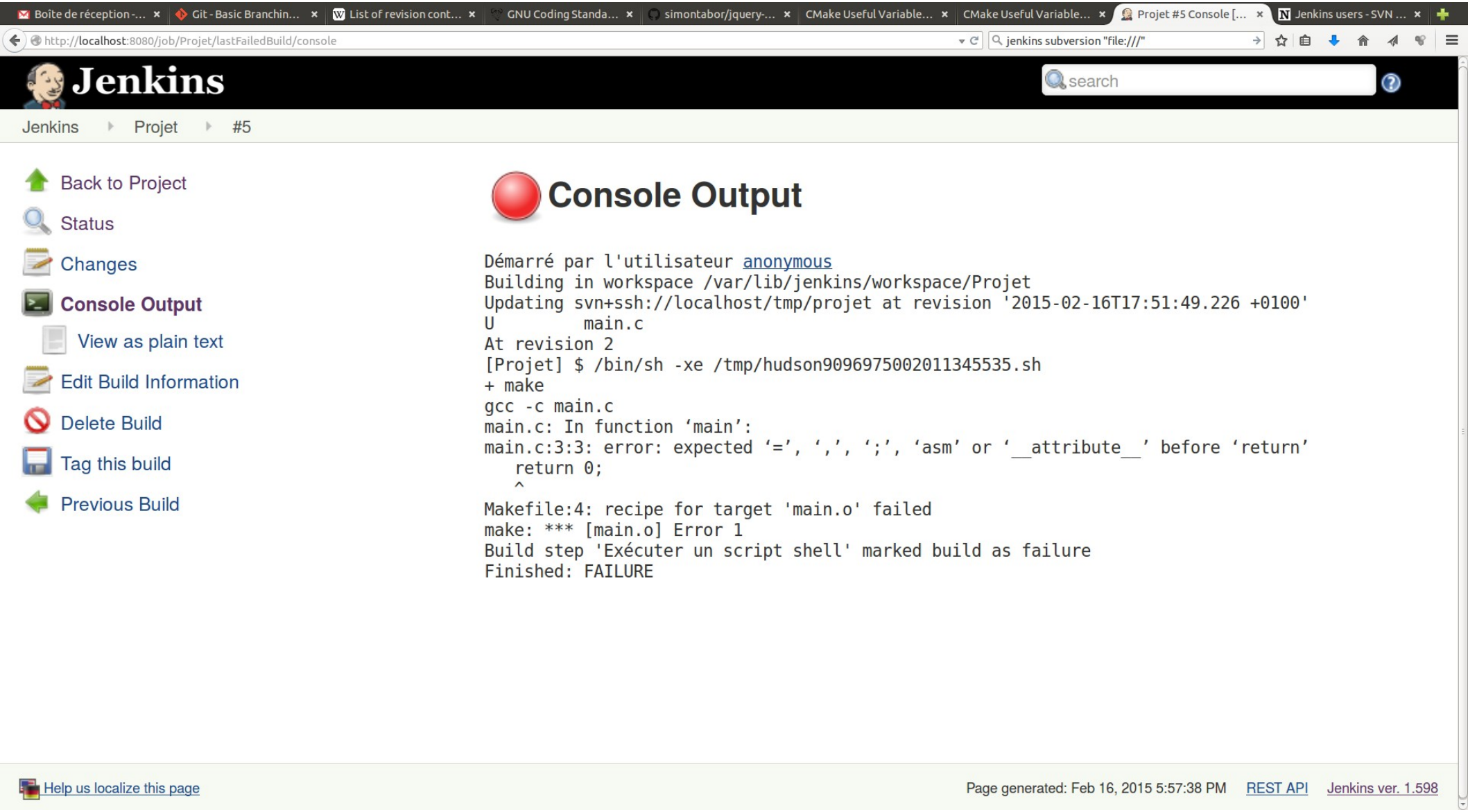
- Ajout d'un bug, commit dans le dépôt



Jenkins : bug


Boîte de réception -...
Git - Basic Branchin...
List of revision cont...
GNU Coding Standa...
simontabor/jquery-...
CMake Useful Variable...
CMake Useful Variable...
Projet #5 [Jenkins]
Jenkins users - SVN ...
http://localhost:8080/job/Projet/lastFailedBuild/
jenkins subversion "file:///"
Jenkins
search
Jenkins
Projet
#5
ENABLE AUTO REFRESH
Back to Project
Status
Changes
Console Output
Edit Build Information
Delete Build
Tag this build
Previous Build
Build #5 (Feb 16, 2015 5:51:49 PM)
Started 5 min 18 sec ago
Took 1 sec
add description
Revision: 2
Changes
1. add a bug (detail)
Started by anonymous user
Help us localize this page
Page generated: Feb 16, 2015 5:57:07 PM
REST API
Jenkins ver. 1.598

Jenkins : bug

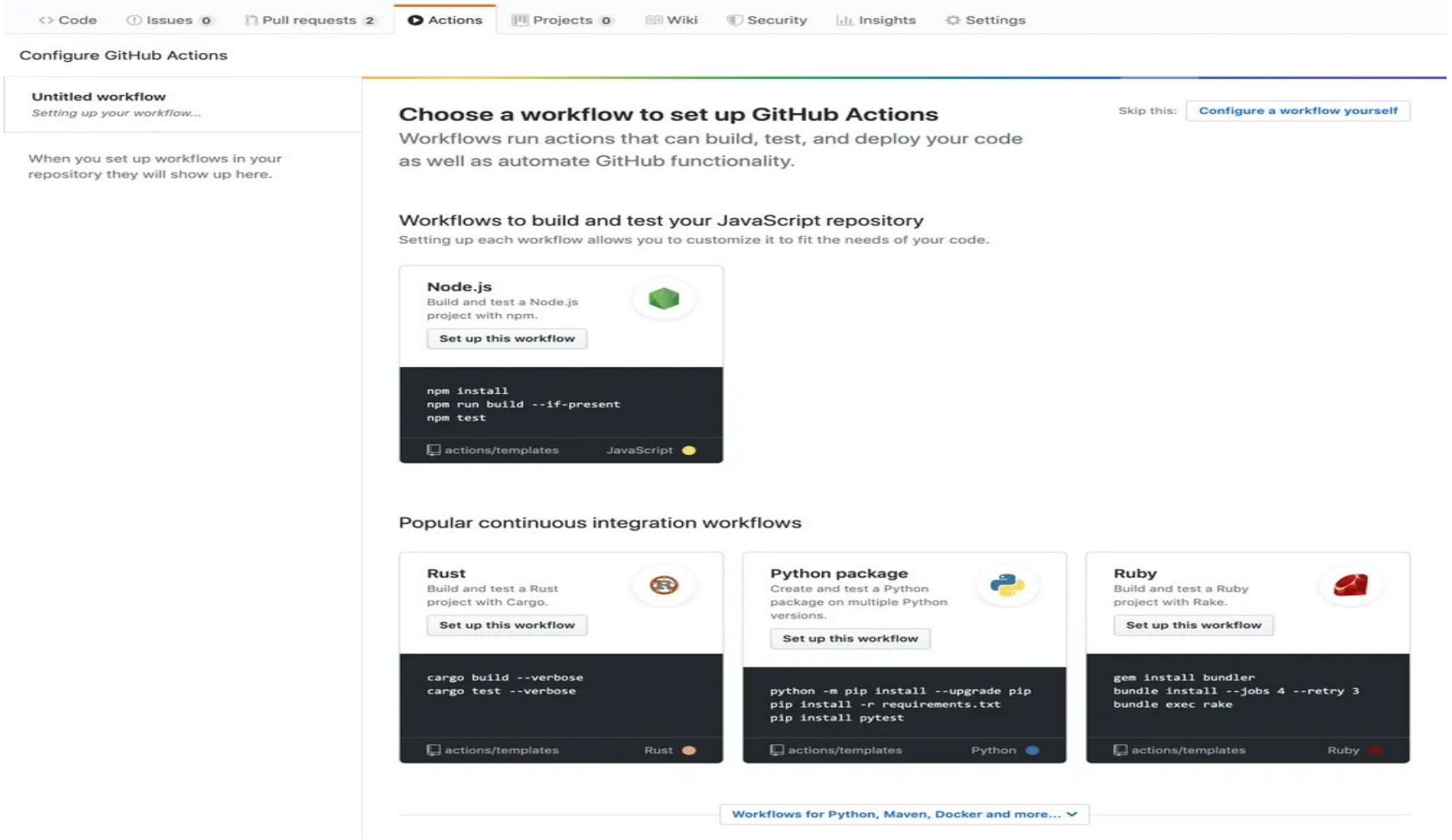

Boîte de réception -...
Git - Basic Branchin...
List of revision cont...
GNU Coding Standa...
simontabor/jquery-...
CMake Useful Variable...
CMake Useful Variable...
Projet #5 Console [...
Jenkins users - SVN ...
http://localhost:8080/job/Projet/lastFailedBuild/console
jenkins subversion "file:///"
Jenkins
search
Jenkins
Projet
#5
Back to Project
Status
Changes
Console Output
View as plain text
Edit Build Information
Delete Build
Tag this build
Previous Build
Console Output
Démarré par l'utilisateur anonymous
Building in workspace /var/lib/jenkins/workspace/Projet
Updating svn+ssh://localhost/tmp/projet at revision '2015-02-16T17:51:49.226 +0100'
U main.c
At revision 2
[Projet] $ /bin/sh -xe /tmp/hudson9096975002011345535.sh
+ make
gcc -c main.c
main.c: In function ‘main’:
main.c:3:3: error: expected ‘=’, ‘,’, ‘;’, ‘asm’ or ‘__attribute__’ before ‘return’
return 0;
^
Makefile:4: recipe for target 'main.o' failed
make: *** [main.o] Error 1
Build step 'Exécuter un script shell' marked build as failure
Finished: FAILURE
Help us localize this page
Page generated: Feb 16, 2015 5:57:38 PM
REST API
Jenkins ver. 1.598

Intégration Continue

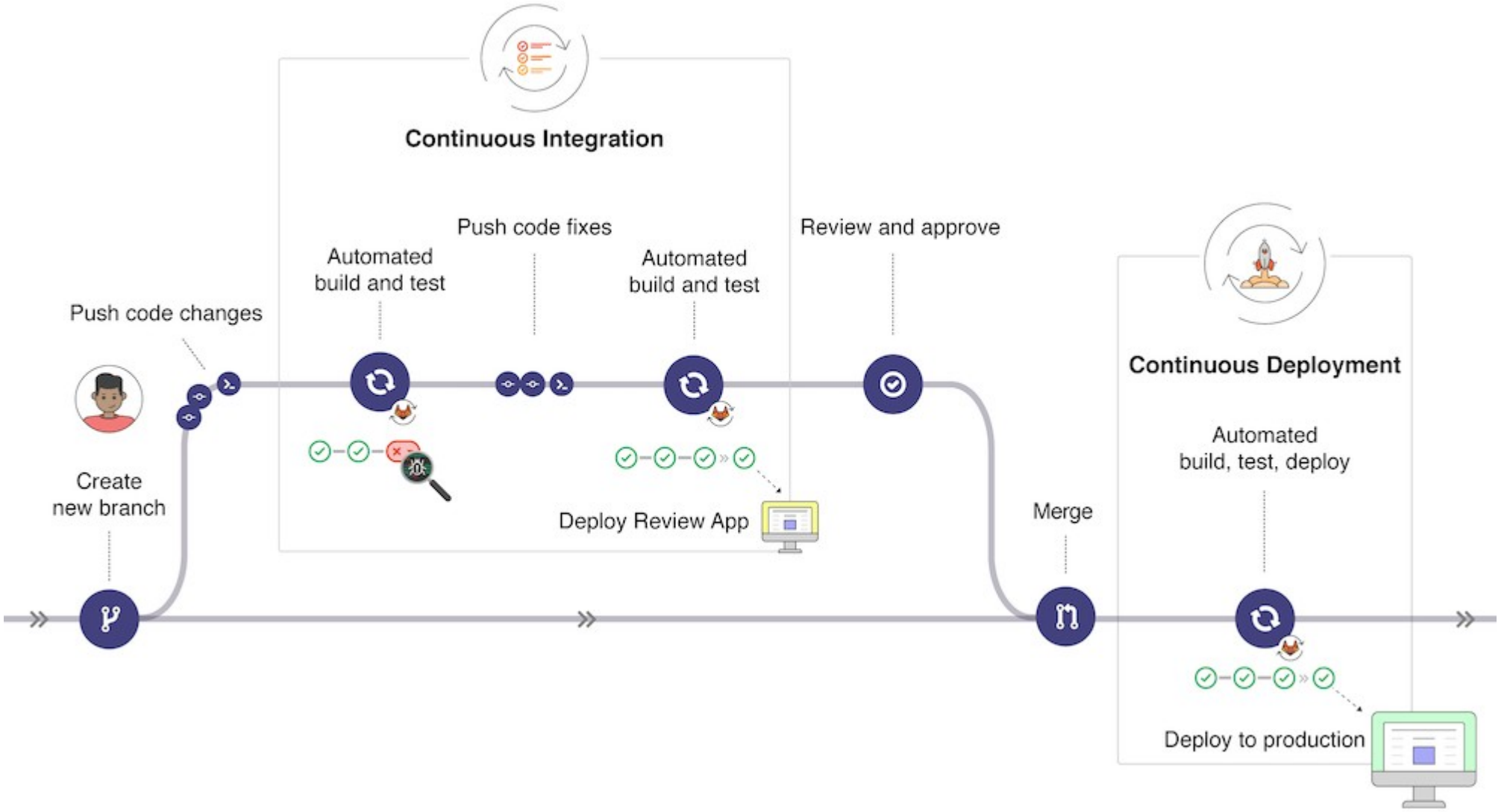
- Tester que le programme compile sous plusieurs environnements est bien mais cela n'offre que peu de garanties quand à l'état **fonctionnel** du projet.
- Pour *garantir* une qualité tout au long du développement, il faut ajouter des **tests**

Intégration Continue & github

- GitHub propose un système d'intégration continue appelé GitHub Action:



Intégration continue & gitlab



Intégration continue & gitlab

- l'IC repose sur l'écriture d'un fichier yml:

```yaml
build-job:
  stage: build
  script:
    - echo "Hello, $GITLAB_USER_LOGIN!"

test-job1:
  stage: test
  script:
    - echo "This job tests something"

test-job2:
  stage: test
  script:
    - echo "This job tests something, but takes more time than test-job1."
    - echo "After the echo commands complete, it runs the sleep command for 20 seconds"
    - echo "which simulates a test that runs 20 seconds longer than test-job1"
    - sleep 20

deploy-prod:
  stage: deploy
  script:
    - echo "This job deploys something from the $CI_COMMIT_BRANCH branch."
  environment: production
```

Les Tests

Les Tests

- Il existe de nombreux types de tests
- Les tests ont pour objectifs de valider votre code :
  - au niveau d'une fonction : tests unitaires
  - au niveau d'un module : tests fonctionnels
  - entre plusieurs modules : tests d'intégration
  - au niveau général, applicatif : tests de recette

TDD : test driven development

- La méthodologie TDD repose sur l'écriture d'abord de tests puis du code validant les tests.
- La méthode XP (extreme programming) repose en partie sur partie sur TDD.
- TDD repose sur des cycles courts consistant :
  - à écrire un test fonctionnel
  - vérifier que le test plante
  - à écrire un test unitaire
  - vérifier que le test plante
  - écrire le code minimal pour que le test fonctionne
  - vérifier que le test passe

TDD

- A la fin de l'écriture d'un code et lorsque tout les tests passent, on peut vouloir refactoriser le code (copier/coller, simplification, unification, …)
- Dans ce cas, on ne touche surtout pas aux tests et on remanie le code jusqu'à ce que à nouveau il valide l'ensemble des tests.

TDD par l'exemple : bowling

- Supposons que l'on souhaite écrire un module de calcul de feuille de scores de bowling :

exemple grandement inspirée de la présentation « TDD in C » par Olve Maudal (dispo. slideshare)

bowling

- Le joueur a 10 sets
- Pour chaque set, le joueur lance la boule une ou deux fois:
  - Si le joueur élimine les 10 quilles du premier coup, il fait **strike** et il ne joue pas de 2ème boule
  - Si le joueur élimine les 10 quilles au deuxième coup, il fait **spare**
- Le score d'un set fait :
  - le score précédent + la somme des deux lancés si pas de strike ni de spare
  - le score précédent + 10 + le score du premier lancé du prochain set si spare
  - le score précédent + 10 + le score du prochain set si strike.
- Pour le 10ème set, le joueur peut avoir un troisième lancé

bowling

- La spécification client est d'avoir un module BowlingGame avoir les méthodes suivantes
  - void roll(BowlingGame *,int nbPinsDown) : enregistre un nouveau lancé
  - int score(BowlingGame *) : renvoie le score actuel

bowling

- Tout d'abord il nous faut une structure pour modéliser une partie :
  - struct Game
- puis une structure pour modéliser un set
  - struct Set
- Une partie est composée de 10 sets :

```
struct Game {
    struct Set sets[10] ;
}
```

- Il faut connaître le set en cours (ajout de currentSet dans Game)
- Pour un set il faut le score de chaque lancé
- Pour le dernier set, il y a peut-être 3 lancés

bowling

- Tout d'abord il nous faut une structure pour modéliser une partie :
  - struct Game
- puis une structure pour modéliser un set
  - struct Set
- Une partie est composée de 10 sets :

```
struct Game {
    struct Set set[10] ;
}
```

- Il faut connaitre le set en cours (ajout de currentSet dans Game)
- Pour un set il faut le score de chaque lancé
- Pour le dernier set, il y a peut-être 3 lancés

PAS TDD !

bowling : en TDD

- En TDD on décrit ce que doit faire le système plutôt que comment le faire
- On commence « basique » :

bowling : en TDD

- En TDD on décrit ce que doit faire le système plutôt que comment le faire
- On commence « basique » :

```
#include<assert.h>
#include<stdbool.h>

int main(){
	assert(false && « c'est parti ») ;
}
```

```
$ make bow
gcc -Wall  bow.c   -o bow
$ ./bow
bow: bow.c:5: main: Assertion `0 && "c'est parti"' failed.
```

ok, le système de test fonctionne !

bowling : cas vide

- Commençons par le cas vide

```
#include<assert.h>
#include<stdbool.h>

void test_empty(){
  int i;
  struct BowlingGame *game=bg_init();
  for(i=0;i<20;++i)
    bg_roll(game,0);
  assert(bg_score(game)==0 && "test empty");
  bg_free(game) ;
}

int main(){
  test_empty();
}
```

```
$ gcc -Wall bow.c
bow.c: In function ‘test_empty’:
bow.c:6:10: warning: implicit declaration of function ‘bg_init’ [-Wimplicit-function-declaration]
   struct BowlingGame *game=bg_init();
          ^
bow.c:6:28: warning: initialization makes pointer from integer without a cast
   struct BowlingGame *game=bg_init();
```

Ajout de bowling.h

```
#ifndef BOWLING_H
#define BOWLING_H


struct BowlingGame;

struct BowlingGame *bg_init();
void bg_roll(struct BowlingGame *,int );
int bg_score(struct BowlingGame *);
void bg_free(struct BowlingGame *) ;

#endif
```

bowling : cas vide

```
$ gcc -Wall bow.c
/tmp/ccTstSlJ.o: In function `test_empty':
bow.c:(.text+0xe): undefined reference to `bg_init'
bow.c:(.text+0x2c): undefined reference to `bg_roll'
bow.c:(.text+0x42): undefined reference to `bg_score'
bow.c:(.text+0x6b): undefined reference to `bg_free'
collect2: error: ld returned 1 exit status
```


Ajout de bowling.c

bowling : cas vide

```
$ gcc -Wall bow.c
/tmp/ccTstSlJ.o: In function `test_empty':
bow.c:(.text+0xe): undefined reference to `bg_init'
bow.c:(.text+0x2c): undefined reference to `bg_roll'
bow.c:(.text+0x42): undefined reference to `bg_score'
bow.c:(.text+0x6b): undefined reference to `bg_free'
collect2: error: ld returned 1 exit status
```

Ajout de bowling.c

```
#include "bowling.h"
#include <stdlib.h>

struct BowlingGame{};

struct BowlingGame *bg_init(){
  return NULL;
}
void bg_roll(struct BowlingGame *g,int s){
}
int bg_score(struct BowlingGame *g){
  return -1;
}
void bg_free(struct BowlingGame *g){}
```

```
$ gcc -Wall bow.c bowling.c
allali@hebus:/tmp/bowling$ ./a.out
a.out: bow.c:10: test_empty: Assertion `bg_score(game)==0 && "test empty"' failed.
```

bowling : cas vide

```
$ gcc -Wall bow.c bowling.c
allali@hebus:/tmp/bowling$ ./a.out
a.out: bow.c:10: test_empty: Assertion `bg_score(game)==0 && "test empty"' failed.
```

Ajout du score

```
#include "bowling.h"
#include <stdlib.h>

struct BowlingGame{
        int score ;
};

struct BowlingGame *bg_init(){
  struct BowlingGame * g=malloc(sizeof(* g)) ;
  g->score=0 ;
  return g ;
}
void bg_roll(struct BowlingGame *g,int s){
}
int bg_score(struct BowlingGame *g){
  return g->score;
}
void bg_free(struct BowlingGame *g){
  free(g) ;
}
```

```
$ gcc bowling.c bow.c -
Wall
$ ./a.out
$ valgrind ./a.out
ok
```

bowling : test tout à 1

```
#include<assert.h>
#include<stdbool.h>

void test_empty(){ … }

void test_all_ones(){
  int i;
  struct BowlingGame *game=bg_init();
  for(i=0;i<20;++i)
    bg_roll(game,1);
  assert(bg_score(game)==20 && "test all ones");
  bg_free(game) ;
}

int main(){
  test_empty();
  test_all_ones() ;
}
```

```
$ ./a.out
a.out: bow.c:19: test_all_ones: Assertion `bg_score(game)==20 && "test all ones"' failed.
```

bowling : test tout à 1

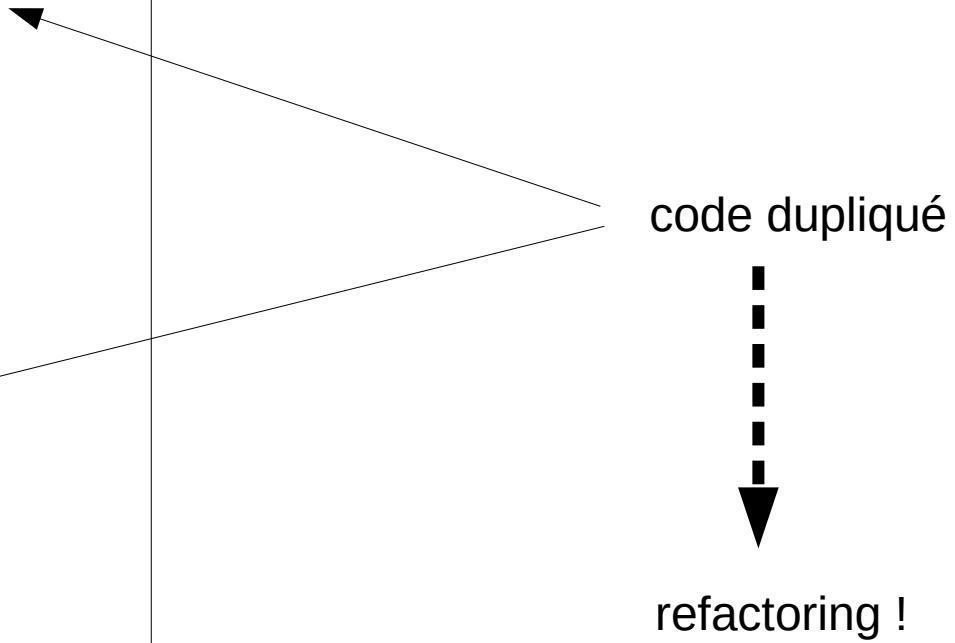
```
$ ./a.out
a.out: bow.c:19: test_all_ones: Assertion `bg_score(game)==20 && "test all ones"' failed.
```

```
$ gcc bowling.c bow.c -Wall
$ ./a.out
$
```

```
#include "bowling.h"
#include <stdlib.h>

struct BowlingGame{
        int score ;
};

struct BowlingGame *bg_init(){
  struct BowlingGame * g=malloc(sizeof(* g)) ;
  g->score=0 ;
  return g ;
}
void bg_roll(struct BowlingGame *g,int s){
   g->score+=s ;
}
int bg_score(struct BowlingGame *g){
  return g->score;
}
void bg_free(struct BowlingGame *g){
  free(g) ;
}
```

bowling : code smell...

```
void test_empty(){
  int i;
  struct BowlingGame *game=bg_init();
  for(i=0;i<20;++i)
    bg_roll(game,0);
  assert(bg_score(game)==0 && "test empty");
  bg_free(game);
}

void test_all_ones(){
  int i;
  struct BowlingGame *game=bg_init();
  for(i=0;i<20;++i)
    bg_roll(game,1);
  assert(bg_score(game)==20 && "test all ones");
  bg_free(game);
}

int main(){
  test_empty();
  test_all_ones();
  return 0;
}
```



bowling : code smell...

```
void test_empty(){
  int i;
  struct BowlingGame *game=bg_init();
  for(i=0;i<20;++i)
    bg_roll(game,0);
  assert(bg_score(game)==0 && "test empty");
  bg_free(game);
}

void test_all_ones(){
  int i;
  struct BowlingGame *game=bg_init();
  for(i=0;i<20;++i)
    bg_roll(game,1);
  assert(bg_score(game)==20 && "test all ones");
  bg_free(game);
}

int main(){
  test_empty();
  test_all_ones();
  return 0;
}
```

```
void rolls(struct BowlingGame *game,int n, int v){
  int i;
  for(i=0;i<n;++i)
    bg_roll(game,v);
}

void test_empty(){
  struct BowlingGame *game=bg_init();
  rolls(game,20,0);
  assert(bg_score(game)==0 && "test empty");
  bg_free(game);
}

void test_all_ones(){
  struct BowlingGame *game=bg_init();
  rolls(game,20,1);
  assert(bg_score(game)==20 && "test all ones");
  bg_free(game);
}

int main(){
  test_empty();
  test_all_ones();
  return 0;
}
```

bowling : un spare

```
void test_one_spare(){
  struct BowlingGame *game=bg_init();
  bg_roll(game,5);
  bg_roll(game,5);
  bg_roll(game,3);
  rolls(game,17,0);
  assert(bg_score(game)==16 && "test one spare");
}
```

```
$ gcc bow.c bowling.c -Wall
allali@hebus:~/SVN_LaBRI/ENSEIRB/PG106/Cours$ ./a.out
a.out: bow.c:31: test_one_spare: Assertion `bg_score(game)==16 && "test one spare"' failed.
```

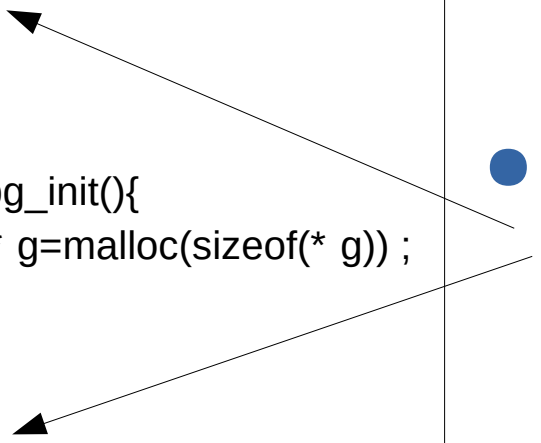
bowling : conception

```
#include "bowling.h"
#include <stdlib.h>

struct BowlingGame{
        int score ;
};

struct BowlingGame *bg_init(){
  struct BowlingGame * g=malloc(sizeof(* g)) ;
  g->score=0 ;
  return g ;
}
void bg_roll(struct BowlingGame *g,int s){
   g->score+=s ;
}
int bg_score(struct BowlingGame *g){
  return g->score;
}
void bg_free(struct BowlingGame *g){
  free(g) ;
}
```

- Pour gérer un spare, il faut connaître le coup d'avant.

bowling : conception

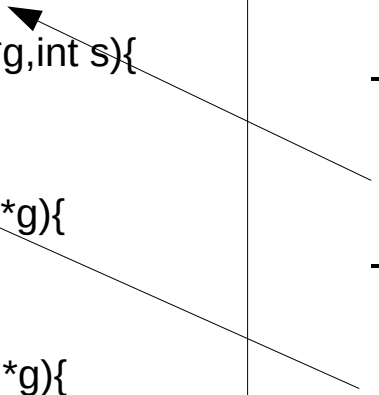
```
#include "bowling.h"
#include <stdlib.h>

struct BowlingGame{
        int score ;
};

struct BowlingGame *bg_init(){
  struct BowlingGame * g=malloc(sizeof(* g)) ;
  g->score=0 ;
  return g ;
}
void bg_roll(struct BowlingGame *g,int s){
   g->score+=s ;
}
int bg_score(struct BowlingGame *g){
  return g->score;
}
void bg_free(struct BowlingGame *g){
  free(g) ;
}
```



- Pour gérer un spare, il faut connaitre le coup d'avant.
- On pourrait ajouter un temporaire pour cela

bowling : conception

```
#include "bowling.h"
#include <stdlib.h>

struct BowlingGame{
        int score ;
};

struct BowlingGame *bg_init(){
  struct BowlingGame * g=malloc(sizeof(* g)) ;
  g->score=0 ;
  return g ;
}
void bg_roll(struct BowlingGame *g,int s){
   g->score+=s ;
}
int bg_score(struct BowlingGame *g){
  return g->score;
}
void bg_free(struct BowlingGame *g){
  free(g) ;
}
```



- Pour gérer un spare, il faut connaitre le coup d'avant.
- On pourrait ajouter un temporaire pour cela
- il y a un problème de conception :
  - roll : calcule le score mais ne devrait pas
  - score : doit calculer le score mais ne le calcul pas

bowling : conception

```
#include "bowling.h"
#include <stdlib.h>

struct BowlingGame{
        int score ;
};

struct BowlingGame *bg_init(){
  struct BowlingGame * g=malloc(sizeof(* g)) ;
  g->score=0 ;
  return g ;
}
void bg_roll(struct BowlingGame *g,int s){
   g->score+=s ;
}
int bg_score(struct BowlingGame *g){
  return g->score;
}
void bg_free(struct BowlingGame *g){
  free(g) ;
}
```

Refactoring !

- Pour gérer un spare, il faut connaitre le coup d'avant.
- On pourrait ajouter un temporaire pour cela
- il y a un problème de conception :
  - roll : calcule le score mais ne devrait pas
  - score : doit calculer le score mais ne le calcul pas

bowling : refactoring

- On revient en arrière :

```
int main(){
  test_empty();
  test_all_ones();
  // test_one_spare() ;
  return 0;
}
```

```
$ gcc bow.c bowling.c -Wall
$ ./a.out
$
```

- On modifie le code : ajout d'un tableau de score, du coup en cours et mise à jour de la fonction de calcul

bowling : refactoring

```
#include "bowling.h"
#include <stdlib.h>

struct BowlingGame{
        int rolls[21] ;
        int current;
        int score ;
};

struct BowlingGame *bg_init(){
  struct BowlingGame * g=malloc(sizeof(* g)) ;
  g->current=0 ;
  return g ;
}
void bg_roll(struct BowlingGame *g,int s){
  g->score+=s ;
   g->rolls[g->current++]=s ;
}
int bg_score(struct BowlingGame *g){
  int score=0,i ;
  for(i=0;i<g->current;++i) score+=g->rolls[i] ;
  return score ;
 return g->score;
}
void bg_free(struct BowlingGame *g){
  free(g) ;
}
```

```
$ gcc bow.c bowling.c -Wall
$ ./a.out
$
```

```
int main(){
  test_empty();
  test_all_ones();
  test_one_spare() ;
  return 0;
}
```

```
$ gcc bow.c bowling.c -Wall
$ ./a.out
a.out: bow.c:31: test_one_spare: Assertion `bg_score(game)==16
&& "test one spare"' failed.
```

bowling : one spare (back)

```
int bg_score(struct BowlingGame *g){
  int score=0,i ;
  for(i=0;i<g->current;++i) score+=g->rolls[i] ;
  return score ;
}
```

```
int bg_score(struct BowlingGame *g){
  int score=0,i ;
  for(i=0;i<g->current;++i) {
      if (g->rollls[i]+g->rolls[i+1]==10){
            // this is a spare...
            score= ... ; // ?
       }
       score+=g->rolls[i] ;
  }
  return score ;
}
```

ca ne marchera pas car il faut compter
par set. On doit encore faire un refactoring !

```
int main(){
  test_empty();
  test_all_ones();
  //test_one_spare() ;
  return 0;
}
```

```
$ gcc bow.c bowling.c -Wall
$ ./a.out
$
```

bowling : refactoring (again)

```
int bg_score(struct BowlingGame *g){
  int score=0,i ;
  for(i=0;i<g->current;++i) score+=g->rolls[i] ;
  return score ;
}
```

```
struct BowlingGame *bg_init(){
  int i ;
  struct BowlingGame * g=malloc(sizeof(* g)) ;
  g->current=0 ;
  for(i=0;i<21;++i) g->rolls[i]=0 ;
  return g ;
}

int bg_score(struct BowlingGame *g){
  int score=0, frame;
  for(frame=0;frame<10;++frame) {
        score+=g->rolls[2*frame]+g->rolls[2*frame+1] ;
  }
  return score ;
}
```

```
$ gcc bow.c bowling.c -Wall
$ ./a.out
$
```

bowling : one spare (again)

```
$ gcc bow.c bowling.c -Wall
$ ./a.out
a.out: bow.c:31: test_one_spare: Assertion `bg_score(game)==16
&& "test one spare"' failed.
```

```
int bg_score(struct BowlingGame *g){
  int score=0, frame, hits;
  for(frame=0;frame<10;++frame) {
        hits=g->rolls[2*frame]+g->rolls[2*frame+1] ;
        score+=hits ;
        if (hits==10) // spare
              score+=+g->rolls[2*frame+2] ;
  }
  return score ;
}
```

```
$ gcc bow.c bowling.c -Wall
$ ./a.out
$
```

bowling : TDD

- Et ainsi de suite :
  - ajout d'un test avec un strike
  - cas pour la fin de partie
  - …
- Le cycle à suivre en TDD est :
  - écriture d'un test
  - le test ne passe pas : ROUGE
  - écriture du code
  - le test passe : VERT
- Lorsqu'on ré-écrit des tests, on ne touche pas au code jusqu'à ce que ça repasse au vert.
- Lorsqu'on ré-écrit le code, on ne touche pas aux tests jusqu'à ce que ça repasse au vert.