DevOps Pull Request

Membres du Groupe: KONAN Yao Paul-David - BERTHE Moussa

Nous allons étudier ici différents outils de test et réaliser un tutorial pour certains d'entre eux (Ceux en rouge ont des tutoriels).

- Testcontainers : Testcontainers to test different environments on the developer's machine
- SauceLabs: Cross Browser Testing, Selenium Testing, and Mobile Testing
- Perfecto - Cloud-based Devops Testing
- Pitest: mutation testing
- Dspot: Automatic unit test amplification
- Stryker-Mutator: mutation testing in JS
- Bazel, Build and test software
- Scott Test Reporter

Scott Test Reporter

Définition de l'élément

Le Scott Test Reporter est un outil utilisé pour générer des rapports détaillés sur les résultats des tests. Il peut être intégré avec des outils de construction tels que Gradle et Maven, ainsi qu'avec d'autres logiciels de test comme JUnit, Mockito et Cucumber.

Comment l'utiliser

1. Intégrez Scott Test Reporter dans votre projet en ajoutant le plugin correspondant à votre système de construction (Gradle ou Maven).
2. Exécutez vos tests unitaires ou d'intégration normalement.
3. Consultez les rapports générés par Scott pour analyser les résultats de vos tests.

Pourquoi l'utiliser

- Générer des rapports détaillés sur les résultats des tests.
- Faciliter l'analyse et le suivi de la qualité du code.
- Améliorer la visibilité et la compréhension des résultats des tests par l'équipe de développement.

Type de projet

Idéal pour les projets Java qui nécessitent des rapports détaillés sur les résultats des tests, notamment ceux utilisant Gradle, Maven, JUnit, Mockito et Cucumber.

Tutorial

Pour ce tutoriel nous utiliserons le projet example fourni par Scott. Nous Utiliserons Maven dans ce projet car y sommes plus habitué par rapport à Gradle

Mettre à jour pom.xml :

- Mettez à jour votre fichier pom.xml.
- Incluez le plugin Maven de Scott et ses dépendances

```
<build>
    <plugins>
        <!-- Ajoutez le Plugin Scott. -->
        <plugin>
            <groupId>hu.advancedweb</groupId>
            <artifactId>scott-maven-plugin</artifactId>
            <version>4.0.1</version>
            <executions>
                <execution>
                    <goals>
                        <goal>prepare-agent</goal>
                    </goals>
                </execution>
            </executions>
        </plugin>

</build>

<dependencies>
```

```
      <!-- Ajoutez Scott comme dépendance -->
       <dependency>
           <groupId>hu.advancedweb</groupId>
           <artifactId>scott</artifactId>
           <version>4.0.1</version>
           <scope>test</scope>
       </dependency>

   </dependencies>
```

- Ensuite nous lancons la commande mvn test.

Prenons ce test comme exemple:

```
@Test
   public void test_1() {
     String first = "Hello";
     String last = "World";

     String concatenated = first + " " + last;

     assertEquals("Goodbye World", concatenated);
  }
```

Sans le plugin scott, voici ce que nous obtenons:

```
[ERROR] test_1(hu.advancedweb.example.StringTest)  Time elapsed: 0.005 s
<<< FAILURE!
org.junit.ComparisonFailure: expected:<[Goodbye] World> but was:<[Hello]
World>
       at hu.advancedweb.example.StringTest.test_1(StringTest.java:16)
```

Avec le plugin scott, voici ce que nous obtenons:

```
[ERROR] test_1(hu.advancedweb.example.StringTest)  Time elapsed: 0.166 s
<<< FAILURE!
java.lang.AssertionError:
```

```
 9|      @Test
10|      public void test_1() {
11|          String first = "Hello";  // first="Hello"
12|          String last = "World";  // last="World"
13|
14|          String concatenated = first + " " + last;  //
concatenated="Hello World"
15|
16|*         assertEquals("Goodbye World", concatenated);  //
ComparisonFailure: expected:<[Goodbye] World> but was:<[Hello] World>
17|      }


      at hu.advancedweb.example.StringTest.test_1(StringTest.java:16)
```

Nous voyons clairement ici la différence entre les différentes sorties. Le plugin scott non seulement nous permet de voir l'évolution du programme lors de son exécution mais aussi permet de voir l'endroit précis ayant causé l'erreur de test ainsi que l'erreur en question(la ligne de l'erreur est marquée par un étoile).
Ces rapports sont disponibles dans le dossier target ou en console.

Cet outil nous permet au final d'avoir plus d'informations sur les causes des erreurs de tests afin de faciliter la résolution de ces erreurs
.

Testcontainers

Définition de l'élément

Testcontainers est une bibliothèque Java qui permet de tester des environnements différents sur la machine du développeur en utilisant des conteneurs Docker.

Comment l'utiliser

1. Intégrez Testcontainers dans votre projet en ajoutant la dépendance correspondante à votre gestionnaire de dépendances (Gradle ou Maven).
2. Utilisez Testcontainers pour créer et gérer des conteneurs Docker lors de l'exécution de vos tests.
3. Configurez les conteneurs selon les environnements que vous souhaitez tester.
4. Écrivez vos tests unitaires ou d'intégration en utilisant les conteneurs créés par Testcontainers.

Pourquoi l'utiliser

- Faciliter le test d'applications dans des environnements variés sans avoir besoin de les installer localement.
- Assurer la portabilité des tests entre différents environnements de développement.
- Optimiser l'efficacité des tests en automatisant la configuration des environnements.

Type de projet

Parfait pour les projets Java où des tests d'intégration doivent être effectués dans des environnements divers, en utilisant des conteneurs Docker pour garantir la cohérence et la portabilité.

SauceLabs

Définition de l'élément

SauceLabs est une plateforme de test qui offre des fonctionnalités de test cross-browser, de test Selenium et de test mobile.

Comment l'utiliser

1. Inscrivez-vous à un compte SauceLabs et configurez vos paramètres de projet.

2. Utilisez les outils et les API fournis par SauceLabs pour créer et exécuter des tests cross-browser, Selenium et mobile.
3. Analysez les résultats des tests générés par SauceLabs pour identifier et résoudre les problèmes de compatibilité et de performance.

Pourquoi l'utiliser

- Tester la compatibilité et les performances des applications web et mobiles sur une large gamme de navigateurs et de plates-formes.
- Automatiser les tests de régression et garantir la qualité des applications sur différents environnements.
- Accélérer le processus de développement en identifiant rapidement les problèmes de compatibilité et de performance.

Type de projet

Recommandé pour les projets web et mobiles nécessitant des tests cross-browser et des tests de compatibilité multiplateforme, offrant une couverture étendue sur une variété de navigateurs et de périphériques.

Perfecto - Cloud-based DevOps Testing

Définition de l'élément

Perfecto est une plateforme de test basée sur le cloud qui offre des fonctionnalités de test DevOps pour les applications mobiles et web.

Comment l'utiliser

1. Inscrivez-vous à un compte Perfecto et configurez votre projet.
2. Utilisez les fonctionnalités de test automatisé et manuel de Perfecto pour tester vos applications sur une variété de périphériques et de plates-formes.
3. Analysez les résultats des tests pour identifier les problèmes de qualité et les goulots d'étranglement dans le processus de développement.

Pourquoi l'utiliser

- Accélérer le processus de développement en automatisant les tests et en intégrant la plateforme de test dans un pipeline DevOps.
- Assurer la qualité des applications sur une variété de périphériques et de plates-formes grâce à une couverture de test étendue.
- Réduire les coûts et les efforts liés aux tests en utilisant une solution basée sur le cloud.

Type de projet

Idéal pour les projets DevOps nécessitant une automatisation complète des tests, notamment les tests mobiles et web, sur une infrastructure cloud fiable et évolutive.

Pitest

Définition de l'élément

Pitest est un outil de test de mutation qui permet d'évaluer la qualité des suites de tests en introduisant des mutations dans le code source et en vérifiant si les tests parviennent à détecter ces mutations.

Comment l'utiliser

1. Intégrez Pitest dans votre projet en ajoutant la dépendance correspondante à votre gestionnaire de dépendances (Gradle ou Maven).
2. Configurez les paramètres de Pitest selon vos besoins, notamment les options de mutation à appliquer.
3. Exécutez Pitest pour générer des rapports sur la couverture de test et l'efficacité de vos suites de tests.

Pourquoi l'utiliser

- Évaluer la robustesse des suites de tests en simulant des mutations dans le code source.
- Identifier les zones de code non couvertes par les tests et améliorer la qualité des tests.
- Faciliter l'identification et la correction des défauts de conception et des erreurs de logique dans le code.

Type de projet

Adapté aux projets Java qui cherchent à améliorer la qualité de leurs tests en identifiant les zones de code non couvertes et en évaluant l'efficacité des suites de tests existantes.

Tutorial: Tests par mutation

Dans ce tutoriel nous utilisons le projet doodle sur le dépôt git. On utilisera Pitest comme outil.

Mettre à jour pom.xml :

- Mettez à jour votre fichier pom.xml.
- Incluez le plugin Maven de pitest comme suit:

```
<plugin>
<groupId>org.pitest</groupId>
<artifactId>pitest-maven</artifactId>
<version>1.4.2</version>
<configuration>
    <targetClasses>
        <!-- Spécifiez le chemin des classes compilées -->
        <param>fr.istic.tlc.services.*</param>
    </targetClasses>
    <targetTests>
        <!-- Spécifiez le chemin des classes de tests compilées -->
        <param>org.acme.services.*Test</param>
    </targetTests>
</configuration>
</plugin>
```

- Nous utiliserons la classe qui Utils.java se trouvant dans de dossier /src/main/services
- voici les tests unitaire avec des mutateurs

```java
public class UtilsTest {
    private static final String CHARS = "abcdefghijkmnopqrstuvwxyzABCDEFGHJKLMNOPQRSTUVWXYZ234567890";

    @Test
    public void testGenerateSlug() {
    Utils utils = Utils.getInstance();
    // Mutation: Changer la valeur de retour de generateSlug à une chaîne vide
    String slug = ""; // Mutation appliquée
    assertEquals(expected:10, slug.length()); // Cette assertion devrait échouer
}

@Test
public void testIntersect() {
    Utils utils = Utils.getInstance();
    // Mutation: Modifier les dates pour créer des plages de dates non chevauchantes
    Date start1 = new Date(2024, 4, 1);
    Date end1 = new Date(2024, 4, 3); // Modifier cette date
    Date start2 = new Date(2024, 4, 5); // Modifier cette date
    Date end2 = new Date(2024, 4, 7);
    boolean result = utils.intersect(start1, end1, start2, end2);
    assertEquals(expected:true, result); // Cette assertion devrait échouer
}
@Test
public void testIntersect_shouldReturnFalseForNonOverlappingDates() {
    // Mutation: Modifier les dates pour qu'elles se chevauchent
    Date start1 = new Date(1234567890L); // Sample date
    Date end1 = new Date(start1.getTime() + 10000); // 10 seconds later
    Date start2 = new Date(start1.getTime() + 5000); // Après range 1
    Date end2 = new Date(start2.getTime() + 5000); // Avant range 2
    assertFalse(Utils.getInstance().intersect(start1, end1, start2, end2));
}
```

```java
@Test
public void testGenerateSlug_shouldGenerateStringWithCorrectLength() {
    int desiredLength = 10;
    // Mutation: Changer la valeur de retour de generateSlug à une chaîne vide
    String slug = ""; // Mutation appliquée
    assertEquals(desiredLength, slug.length()); // Cette assertion devrait échouer

    // Mutation: Modifier la boucle pour vérifier les caractères incorrects
    for (char c : slug.toCharArray()) {
        // Modifier la condition pour tester une caractère invalide
        assertTrue(CHARS.indexOf(c) != -1); // Cette assertion devrait échouer
    }
}
```

- Il faut executer la commande : mvn org.pitest:pitest-maven:mutationCoverage Pour lancer les tests;
- voici les résultats:

```
================================================================================
- Mutators
================================================================================
> org.pitest.mutationtest.engine.gregor.mutators.ConditionalsBoundaryMutator
>> Generated 1 Killed 0 (0%)
> KILLED 0 SURVIVED 0 TIMED_OUT 0 NON_VIABLE 0
> MEMORY_ERROR 0 NOT_STARTED 0 STARTED 0 RUN_ERROR 0
> NO_COVERAGE 1
--------------------------------------------------------------------------------
> org.pitest.mutationtest.engine.gregor.mutators.IncrementsMutator
>> Generated 1 Killed 0 (0%)
> KILLED 0 SURVIVED 0 TIMED_OUT 0 NON_VIABLE 0
> MEMORY_ERROR 0 NOT_STARTED 0 STARTED 0 RUN_ERROR 0
> NO_COVERAGE 1
--------------------------------------------------------------------------------
> org.pitest.mutationtest.engine.gregor.mutators.VoidMethodCallMutator
>> Generated 1 Killed 0 (0%)
> KILLED 0 SURVIVED 0 TIMED_OUT 0 NON_VIABLE 0
> MEMORY_ERROR 0 NOT_STARTED 0 STARTED 0 RUN_ERROR 0
> NO_COVERAGE 1
--------------------------------------------------------------------------------
> org.pitest.mutationtest.engine.gregor.mutators.ReturnValsMutator
>> Generated 6 Killed 0 (0%)
> KILLED 0 SURVIVED 0 TIMED_OUT 0 NON_VIABLE 0
> MEMORY_ERROR 0 NOT_STARTED 0 STARTED 0 RUN_ERROR 0
> NO_COVERAGE 6
--------------------------------------------------------------------------------
> org.pitest.mutationtest.engine.gregor.mutators.NegateConditionalsMutator
>> Generated 9 Killed 0 (0%)
> KILLED 0 SURVIVED 0 TIMED_OUT 0 NON_VIABLE 0
> MEMORY_ERROR 0 NOT_STARTED 0 STARTED 0 RUN_ERROR 0
> NO_COVERAGE 9
--------------------------------------------------------------------------------
================================================================================
- Timings
================================================================================
```

```
================================================================================
> scan classpath : < 1 second
> coverage and dependency analysis : < 1 second
> build mutation tests : < 1 second
> run mutation analysis : < 1 second
--------------------------------------------------------------------------------
> Total  : 1 seconds
--------------------------------------------------------------------------------
================================================================================
- Statistics
================================================================================
>> Generated 18 mutations Killed 0 (0%)
>> Ran 0 tests (0 tests per mutation)
```

- Interprétation et mutateur :

❖ ConditionalsBoundaryMutator :

- Ce mutateur modifie les conditions des expressions conditionnelles en remplaçant les limites par des limites alternatives pour tester les comportements aux limites.

- Dans votre cas, il a généré 1 mutation, mais aucune n'a été tuée par les tests.

❖ IncrementsMutator :

- Ce mutateur remplace les opérateurs d'incrémentation et de décrémentation par leurs équivalents inverses.
- Dans votre cas, il a généré 1 mutation, mais aucune n'a été tuée par les tests.

❖ VoidMethodCallMutator :

- Ce mutateur remplace les appels de méthode qui retournent une valeur par des appels de méthode similaires qui ne retournent rien (void).
- Dans votre cas, il a généré 1 mutation, mais aucune n'a été tuée par les tests.

❖ ReturnValsMutator :

- Ce mutateur modifie les valeurs retournées par les méthodes.
- Dans votre cas, il a généré 6 mutations, mais aucune n'a été tuée par les tests.

❖ NegateConditionalsMutator :

- Ce mutateur inverse les conditions booléennes.
- Dans votre cas, il a généré 9 mutations, mais aucune n'a été tuée par les tests.

Dspot

Définition de l'élément

Dspot est un outil d'amplification de tests unitaires automatique qui vise à améliorer la couverture de test en générant automatiquement des cas de test supplémentaires.

Comment l'utiliser

1. Intégrez Dspot dans votre projet en ajoutant la dépendance correspondante à votre gestionnaire de dépendances (Gradle ou Maven).
2. Exécutez Dspot en spécifiant les classes ou les méthodes à tester et les paramètres de configuration.

3. Analysez les cas de test générés par Dspot et incorporez-les dans votre suite de tests existante.

Pourquoi l'utiliser

- Améliorer la couverture de test en générant automatiquement des cas de test supplémentaires.
- Identifier les zones de code non testées et réduire les risques de défauts de logique.
- Accélérer le processus de développement en automatisant la création de tests unitaires.

Type de projet

Recommandé pour les projets Java où une augmentation de la couverture de test est nécessaire, en automatisant la génération de cas de test supplémentaires pour améliorer la qualité du code.

Tutorial

1. Configuration dans pom.xml

Ajoutez le plugin DSpot Maven dans la section des plugins de votre fichier pom.xml.

```
</project>



<plugin>

   <groupId>eu.stamp-project</groupId>

   <artifactId>dspot-maven</artifactId>

   <version>LATEST</version>

   <configuration>

       <!-- Votre configuration ici -->
```

```
    </configuration>

</plugin>
```

Ici on remplace ‘LATEST’ par le numéro de la version desirée de dspot. Au moment de la réalisation de ce tutoriel, la derniere version de dspot est: 3.2.0

2. Configuration du Plugin

Configurez les options nécessaires dans la section `<configuration>` du plugin pour spécifier les amplificateurs et les sélecteurs de tests à utiliser.

Par exemple, pour amplifier les tests unitaires et améliorer la couverture de code :

```
<configuration>
   <test-criterion>JacocoCoverageSelector</test-criterion>
</configuration>
```

Pour plus d’informations sur les options de configurations, consulter https://github.com/STAMP-project/dspot

3. Exécution

Après avoir configuré DSpot dans votre fichier pom.xml, exécutez la commande suivante à partir de la racine de votre projet :

```
mvn dspot:amplify-unit-tests
```

4. Options Avancées

Vous pouvez également spécifier des options supplémentaires directement à partir de la ligne de commande en préfixant chaque option avec `-D`, comme indiqué dans l'exemple suivant :

```
mvn eu.stamp-project:dspot-maven:amplify-unit-tests
-Dtest=my.package.TestClass -Dcases=testMethod
```

- `-Dtest=my.package.TestClass`: C'est une option système Maven (`-D`) pour spécifier le test unitaire à amplifier. Dans cet exemple, `my.package.TestClass` est le nom de la classe de test que vous souhaitez amplifier.

- `-Dcases=testMethod`: C'est une autre option système Maven (`-D`) pour spécifier la méthode de test à amplifier. Dans cet exemple, `testMethod` est le nom de la méthode spécifique que vous souhaitez amplifier dans la classe de test spécifiée.

Pour une telle classe test:

```
package com.example;

import static org.junit.Assert.assertEquals;

import org.junit.Test;

public class CalculatorTest {

   @Test
   public void testAdd() {
       Calculator calculator = new Calculator();
       int result = calculator.add(3, 4);
       assertEquals(7, result);
   }
}
```

Voici ce qu'on pourrait obtenir:
Test class that has been amplified: com.example.CalculatorTest

The original test suite kills 2 mutants

The amplification results with 0 new tests

it kills 0 more mutants

Le rapport se trouve dans le fichier target/dspot/output/report.txt

Stryker-Mutator

Définition de l'élément

Stryker-Mutator est un outil de test de mutation spécialement conçu pour les projets JavaScript. Il permet d'évaluer la qualité des suites de tests en introduisant des mutations dans le code source JavaScript et en vérifiant si les tests parviennent à détecter ces mutations.

Comment l'utiliser

1. Intégrez Stryker-Mutator dans votre projet JavaScript en l'installant via npm ou en l'ajoutant à votre configuration de build.
2. Configurez les paramètres de Stryker-Mutator selon vos besoins, notamment les options de mutation à appliquer.
3. Exécutez Stryker-Mutator pour générer des rapports sur la couverture de test et l'efficacité de vos suites de tests JavaScript.

Pourquoi l'utiliser

- Évaluer la robustesse des suites de tests JavaScript en simulant des mutations dans le code source.
- Identifier les zones de code non couvertes par les tests et améliorer la qualité des tests.
- Faciliter l'identification et la correction des défauts de conception et des erreurs de logique dans le code JavaScript.

Type de projet

Spécialement conçu pour les projets JavaScript qui cherchent à évaluer la qualité de leurs tests en simulant des mutations dans le code source et en identifiant les zones de code non testées.Type de projet :
Parfait pour les projets de grande envergure nécessitant une construction rapide et fiable, offrant une parallélisation efficace des tâches de construction et une gestion avancée des dépendances.

Bazel

Définition de l'élément

Bazel est un système de construction et de test open-source développé par Google. Il est conçu pour permettre une construction rapide, fiable et extensible de logiciels à grande échelle.

Comment l'utiliser

1. Intégrez Bazel dans votre projet en l'installant et en configurant votre projet pour l'utiliser comme système de construction.
2. Définissez les règles de construction et de test dans votre configuration Bazel pour décrire comment votre projet doit être construit et testé.
3. Utilisez les commandes Bazel pour construire et tester votre projet de manière efficace et fiable.

Pourquoi l'utiliser

- Accélérer le processus de construction en parallélisant les tâches de construction et en évitant la recompilation inutile.
- Assurer la fiabilité des constructions en utilisant une vérification de cache et une gestion des dépendances avancées.
- Faciliter le développement collaboratif en fournissant des outils de construction et de test cohérents et extensibles.

Type de projet

Parfait pour les projets de grande envergure nécessitant une construction rapide et fiable, offrant une parallélisation efficace des tâches de construction et une gestion avancée des

dépendances.

L'intégration d'outils de test dans un pipeline DevOps est essentielle pour garantir la qualité et la fiabilité des logiciels tout au long du cycle de développement. Les outils présentés ici offrent une gamme de fonctionnalités pour améliorer la couverture de test, automatiser les tests d'intégration et de régression, ainsi que pour identifier les défauts de logique et les erreurs de conception.

Scott Test Reporter fournit des rapports détaillés sur les résultats des tests, facilitant ainsi l'analyse et le suivi de la qualité du code. En intégrant Scott dans un pipeline DevOps, les équipes de développement peuvent rapidement identifier et résoudre les problèmes de test, améliorant ainsi la robustesse et la fiabilité des applications.

Testcontainers simplifie le test d'applications dans des environnements variés en utilisant des conteneurs Docker, offrant ainsi une approche portable et reproductible pour les tests d'intégration. En intégrant Testcontainers dans un pipeline DevOps, les équipes peuvent garantir la cohérence des tests sur différents environnements de développement.

SauceLabs et Perfecto offrent des solutions de test cross-browser et mobile dans le cloud, permettant aux équipes de développement de tester leurs applications sur une large gamme de plates-formes et de navigateurs. En automatisant les tests avec SauceLabs et Perfecto, les équipes peuvent accélérer le processus de développement tout en garantissant la compatibilité et les performances des applications.

Pitest et Dspot sont des outils de test mutationnel qui permettent d'évaluer la qualité des suites de tests en introduisant des mutations dans le code source. En identifiant les zones de code non couvertes et en évaluant l'efficacité des suites de tests, Pitest et Dspot aident les équipes de développement à améliorer la qualité du code et à réduire les risques de défauts de logique.

Enfin, Stryker-Mutator offre une solution similaire pour les projets JavaScript, permettant d'évaluer la qualité des tests en simulant des mutations dans le code source

JavaScript. En intégrant ces outils dans un pipeline DevOps, les équipes de développement peuvent automatiser les tests et garantir la qualité et la fiabilité des logiciels à chaque étape du processus de développement.