Principes et Commandes en Java pour Maven - JUnit - Logging

Maven

Principe :

Un projet Maven permet notamment :

* gérer facilement les dépendances, cad les bibliothèques ajoutées au projet sans avoir à importer manuellement les .jar
* gérer le cycle de vie du projet (depuis la simple compilation du code jusqu'au déploiement complet en passant par la création de .jar runnable et les tests de fonctionnalités) => voir *Java-Maven\_Complement*
* l'utilisation simple de propriétés contenues dans un fichier externe

Instalation de Maven :

Télécharger Maven sur http://maven.apache.org

L'installer à la racine du lecteur C:

Créer une variable d'environnement JAVA\_HOME - adresse du JDK (jdk.1.8...)

Mettre à jour la variable PATH avec l'adresse du repertoire bin de Maven (permet d'utiliser mvn en ligne de commande)

tester avec la commande : mvn --version

Nouveau projet Maven :

Lors de la génération d'un projet Maven, il faut définir obligatoirement :

-groupId: Dev => identifiant de l'organisme qui développe le projet

-artifactID: nomduprojet => nom du projet

-version: 1.0SNAPSHOT => version du projet

-package: dev => nom du fichier lors du packaging

Contenu du projet :

Fichier POM

gère les dépendances

/src/main/

code source

/src/test/

test unitaire JUnit

/src/main/resources

contiens les fichier .properties => fichier contenant les propriétés et leurs valeurs que l'on pourra utiliser dans le code Java

contiens les autres fichiers resources

/target

code compilé (.class et .jar)

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Symbole | Signification | Code | | Utilisation | |
| Maven |  | | | | |
| en ligne de commande |  |  | |  |  |
| mvn compile | compile le code source du projet (= génère les fichiers .class dans le dossier *target*) | mvn compile | |  |  |
| mvn package | génère un JAR exécutable du projet dans le dossier *target* | mvn package | | *lancer le projet à partir du jar runnable :*  java -jar target/monProjet.jar |  |
| mvn clean | vide le contenu du dossier *target*  vide le dossier puis lance la compilation  vide le dossier puis fait le package | mvn clean  mvn clean compile  mvn clean package | |  |  |
| mvn instal | instale le projet selon les spécificités indiquées (valeurs des paramètres, test préalables, …)  (par défaut, instale dans le dossier : utilisateur/xxx/.m2/repository/…) | mvn instal | |  |  |
|  |  |  | |  |  |
|  |  |  | |  |  |
| POM |  |  | |  |  |
| <dependencies> |  |  | | |  |
| <dependency> | ajout d'une dépendance (= bibliothèque) au projet | <dependencies>  <dependency>  <groupId>monGroupe</groupId>  <artifactId>maBibliotheque</artifactId>  <version>3.8.1</version>  <scope>test</scope> *=> facultatif*  </dependency>  <dependency>  …  </dependency>  <dependencies> | | |  |
| <properties> |  |  | | |  |
| project.build.sourceEncoding | tout encoder en UTF-8 | <properties>  <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>  </properties> | | |  |
| <maPropriete> | déclarer une propriété et sa valeur qui seront potentiellement accessible dans le code Java  (utilisation de ces propriétés dans le POM via EL) | <properties>  <maProprieteA>maValeura</maProprieteA>  <maProprieteB>maValeurB</maProprieteB>  </properties> | | réutilisation de la propriété ailleur dans le POM :  <xxx>**${maProprieteA}**</xxx> |  |
| <build> |  |  | |  |  |
| <finalName> | Définition d'un nom qui sera utilisé lors de la création du package à la place du nom du projet | <build>  <finalName>monNomFinal</finalName>  </build> | |  |  |
| .properties | Activation du remplacement de la valeur des propriétés via Maven (= accès aux valeurs des propriétés contenues dans le fichier src/main/resources/.properties depuis le code Java) | <build>  <resources>  <resource>  <directory>src/main/resources</directory>  <filtering>true</filtering>  </resource>  </resources>  </build> | | *contenu du fichier* **.properties**  maPropriete1=ma String n°1  maPropriete2=maValeur2  maPropriete3=${maProprieteBdansLePOM}  *récupération de la valeur dans une classe Java*  import java.util.ResourceBundle; |  |
|  |  |  | | public class MaClasse {  public void maMethode {  String str = ResourceBundle.getBundle("application").getString("maPropriete1");  int nb = ResourceBundle.getBundle("application").getInteger("maPropriete2");  }  } | |
| <plugin> |  |  | |  |  |
|  | indique l'adresse du la classe Main à lancer au démarrage pour pouvoir générer un JAR exécutable  et précise avec maven-shade-plugin que le JAR doit contenir toutes les dépendances pour être fonctionnel |  | <plugin>  <groupId>...  <artifactId>...  <version>...  <configuration>  <archive>  <manifest>  <mainClass>monPackage.MaClasseMain</mainClass>  </manifest>  </archive>  </configuration>  </plugin>  <plugin>  <groupId>org.apache.maven.plugins</groupId>  <artifactId>maven-shade-plugin</artifactId>  <version>3.0.0</version>  <executions>  <execution>  <phase>package</phase>  <goals>  <goal>shade</goal>  </goals>  </execution>  </executions>  </plugin> | |  |
| <profiles> |  |  | |  |  |
| <profile> | permet de créer plusieurs profils pour utiliser des valeurs de propriétés différents selon celui activé | <profiles>  <profile>  <id>dev</id>  <activation>  <activeByDefault>true</activeByDefault>  </activation>  <properties>  <mode>DEV</mode>  </properties>  </profile>  <profile>  <id>prod</id>  <properties>  <mode>PROD</mode>  </properties>  </profile>  </profiles>  </project> | | *activation du profil PROD en ligne de commande :*  mvn package -Pprod |  |
|  |  |  | |  |  |
|  |  |  | |  |  |
|  |  |  | |  |  |
|  |  |  | |  |  |
|  |  |  | |  |  |
|  |  |  | |  |  |
|  |  |  | |  |  |
|  |  |  | |  |  |

JUnit

Principe :

Dans le dossier src/test/, on créé une classe miroir pour chacune des classes de notre projet afin de pratiquer des tests unitaires sur chacune des méthodes qu'elles contiennent.

Une classe métier => une classe test.

Une méthode => une ou plusieurs méthodes contenant un test unitaire vérifiant une condition particulière.

Permet de pratiquer des tests avant d'intégrer le code au projet et de corriger et consolider au maximum le code de chaque méthode afin d'éviter les bugs, les exceptions non gérées, …

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Symbole | Signification | Code | | Utilisation | |
| JUnit |  | | | | |
| <dependency> | ajout de la dépendance de JUnit dans le POM  Rq : JUnit est ajouté par défaut à la dépendance d'un projet Maven généré par Spring. | <dependencies>  <dependency>  <groupId>junit</groupId>  <artifactId>junit</artifactId>  <version>4.12</version>  </dependency>  </dependencies> |  | |  |
| MaClasseTest | conventions de nommages ! | *dans src/main/*  public class MaClasse {  public String maMethodeA(String monParam1) {  instruction(s);  }    public void maMethodeB() {  instruction(s);  } | *dans src/test/*  public class MaClasseTest {  @Test  public void testMaMethodeA() {  instruction(s) pour le test;  }  @Test  public void testMaMethodeB() {  instuction(s) pour le test;  }  @Test  public void testMaMethodeBAvecNull() {  instruction(s) pour le test;  }  } | |  |
|  |  |  |  | |  |
| Annotations |  |  |  | |  |
| @BeforeClass | exécution du code avant l'instanciation de la classe de test | public class MaClasseTest {  @BeforeClass  public static void setUpBeforeClass() {  }  @Before  public void setUp() {  }  @Test  public void testMaMethodeA() {  }  **@Test(expected = NullPointerException**.class**)**  public void testMaMethodeB() {  }  @After  public void tearDown() {  }  @AfterClass  public static void tearDownAfterClass() {  }  } |  | |  |
| @Before | exécution du code avant l'exécution de chaque test |  | |  |
| @Test  (timeout = )  (expected = ) | identifie les méthodes de test  *Options :*  le test échoue si l'exécution dure plus longtemps que la valeur indiquée en ms  le résultat attendu du test est la levée d'une exception indiquée en paramètre. Si l'exception n'est pas levée, le test échoue | @Test(timeout = 1000)  public void testMaMethodeA() {  } | |  |
| @After | exécution du code après exécution de chaque test |  | |  |
| @AfterClass | exécution après l'exécution de tour les tests |  | |  |
|  |  |  |  | |  |
|  |  |  |  | |  |
| Assert | objet vérifiant l'égalité entre deux objets et retournant un booléen validant ou invalidant le test | public class maClasseTest {  @Test  public void testMaMethodeA() {  MaClasse maClasse = new MaClasse();  String valeurObtenue = maClasse.maMethodeA(valeurParam1);  Assert.assertEquals(10, valeurObtenue.length());  Assert.assertEquals("Valeur attendue", valeurObtenue);  } | | | Rq: le test est validé pour cette méthode uniquement si les deux **Assert** ont renvoyés true. |
| .assertEquals( , ) | compare deux objets de même type  Rq: utilise **.equals( )** | Assert.assertEquals(objetAttendu, objetObtenu); |  | |  |
| .assertEquals( , , ) | idem que précédent, mais affiche le message passé en premier paramètre en cas d'erreur (=> plus facile pour identifier le problème) | Assert.assertEquals("Mon message d'erreur", objetAttendu, objet obtenu); |  | |  |
| .assertTrue( )  .assertFalse( ) | vérifie qu'un booleen est :  True  Flase | Assert.assertTrue(monBooleanObtenu); | Assert.assertTrue("Message erreur", booleenObtenu); | |  |
| .assertNull( )  .assertNotNull( ) | vérifie qu'un objet  est null  n'est pas null | Assert.assertNull(monObjetObtenu); | Assert.assertNull("Message erreur", monObjetObtenu); | |  |
| .assertArrayEquals( , ) | compare deux tableaux | Assert.assertArrayEquals(monTableauAttendu, monTableauObtenu); | Assert.assertArrayEquals("Message erreur", monTableauAttendu, monTableauObtenu); | |  |
| .fail() | vérifie que le test a échoué | Assert.fail(); | Assert.fail("Message d'erreur"); | |  |
|  |  |  |  | |  |
|  |  |  |  | |  |
|  |  |  |  | |  |
|  |  |  |  | |  |
|  |  |  |  | |  |
|  |  |  |  | |  |
|  |  |  |  | |  |
|  |  |  |  | |  |
|  |  |  |  | |  |
|  |  |  |  | |  |
|  |  |  |  | |  |
|  |  |  |  | |  |
|  |  |  |  | |  |
|  |  |  |  | |  |
|  |  |  |  | |  |
|  |  |  |  | |  |

Logging

Principe :

Placer des éléments dans le codes qui inscriront une trace de l'exécution du code (quand, par qui, pourquoi, qu'est-ce qui est fait) afin de pouvoir retrouver en cas de problème ce qui a été fait/demandé à ce moment-là.

Système :

Ceci nécessite l'ajout de 2 éléments au projet :

* une API d'abstraction : SLF4J = Simple Logging Facade For Java (permet de changer de bibliothèque de Logging sans avoir à redéfinir tous les commandes)
* une bibliothèque de Logging (ici LogBack qui est l'une des bibliothèques proposées par défaut par SLF4J)

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Symbole | Signification | Code | Utilisation | |
| Logging |  | | | |
| Configuration |  |  |  |  |
| POM | ajout de la dépendance de LogBack | <dependency>  <groupId>ch.qos.logback</groupId>  <artifactId>logback-classic</artifactId>  <version>1.2.3</version>  </dependency> |  |  |
|  | ajout de la dépendance de SLF4J | <dependency>  <groupId>ch.qos.slf4j</groupId>  <artifactId>slf4j-api</artifactId>  <version>1.7.25</version>  </dependency> |  |  |
| logback.xml | fichier de configuration utilisé par LogBack à placer à la racine du projet (dans Maven : src/main/resources/) |  |  |  |
|  |  |  |  |  |
| LoggerFactory  .trace( )  .debug( )  .info( )  .warn( )  .error( )  .fatal( )  .info( , ) | utilisation d'un LOGGER à partir des informations issues de logback.xml  Méthodes déclenchant le logging, selon le niveau de gravité du Log  message avec utilisation d'un paramètre | public class MaClasse {  private static final Logger LOGGER = **LoggerFactory**.getLogger("**monLogger**");  public void maMethode() {  LOGGER.info("maMethode vien d'être appellée");    try {  instruction(s);  } catch (Exception e) {  LOGGER.warn("Une exception vient d'être capturée dans maMethode");  }  } | *Utilisation d'un logger dédié au package auquel appartient la classe :*  public class MaClasse {  private static final Logger LOGGER = **LoggerFactory**.getLogger(**MaClasse.class**);  public void maMethode() {  instruction(s)  String param = "Valeur de mon parametre";  LOGGER.error("Msg d'erreur avec parametre : {}", param);  } |  |
| <logger >  <appender-ref= > | identifie un logger et précise le niveau minimal de gravité du log pour que celui-ci laisse une trace dans la sortie donnée dans appender-ref | <?xml version="1.0" encoding="UTF-8"?>  <configuration>  <logger name="**monLogger**" level="**INFO**" **additivity="false">**  <appender-ref ref="**STDOUT**" />  </logger>  <logger name="**dev.monPackage**" level="**WARN**" **additivity="false">**  <appender-ref ref="**FILE**" />  </logger>  <**root** level="ERROR">  <appender-ref ref="**STDOUT**" />  </**root**> | *6 niveaux de gravités du Log :*  **TRACE**  **DEBUG**  **INFO**  **WARN**  **ERROR**  **FATAL** | Rq: le message n'est envoyé vers le média de sortie que si le niveau de gravité du log est égal ou supérieur à celui défini dans le logger !  Rq: root est le logger par défaut. Tous les autres héritent de root. **additivity="false"** permet d'intérompre la chaine et de ne pas avoir de message en doublons par le logger lui-même et par son parent root. |
| <appender > | définit un média de sortie et le pattern pour le message | <appender name="**STDOUT**" class="ch.qos.logback.core.ConsoleAppender">  <layout class="ch.qos.logback.classic.PatternLayout">  <Pattern>%d{HH:mm:ss.SSS} [%thread] %level %logger{36} - %msg%n</Pattern>  </layout>  </appender>  <appender name="**FILE**" class="ch.qos.logback.core.FileAppender">  <file>monFile.log</file>  <append>true</append>  <immediateFlush>true</immediateFlush>  <encoder>  <Pattern>%d{HH:mm:ss.SSS} [%thread] %level %logger{36} - %msg%n</Pattern>  </encoder>  </appender>  </configuration | *Appenders de LogBack :*  ch.qos.logback.core  .ConsoleAppender  .OutputStreamAppender  .FileAppender  .RollingFileAppender  .SocketAppender  .SSLSocketAppender  .SMTPAppender  .DBAppender  *Pattern :*   |  |  | | --- | --- | | %d  %d{HH:mm:ss.SSS}  %thread  %level  %logger  %logger{n}  %msg  %n | horodatage  horodatage avec format  thread  niveau de gravité  nom du logger  nom du logger lim à n ca.  msg  retour à la ligne | |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |