מערכת בנייה - גריידל

פרוייקטי-תוכנה גדולים מורכבים מהרבה חלקים התלויים זה בזה, ותלויים גם ברכיבים חיצוניים התלויים ברכיבים אחרים, וכן הלאה. כדי לבנות פרוייקטים כאלה, משתמשים בכלים מיוחדים היודעים לנתח את התלויות בין הרכיבים השונים ולבנות את כל המערכת בסדר הנכון. הכלים האלה נקראים build automation systems.

ישנם כלים רבים הממלאים תפקיד דומה, למשל: make, ant, maven. אנחנו נלמד על **gradle - גריידל**.

התקנת גריידל

ישנן דרכים רבות להתקין גריידל, בהתאם למערכת ההפעלה שלכם.
אני התקנתי באופן ידני דרך הקישור הזה: https://gradle.org/install/#manually

משימות - תיקייה 11

כשמריצים גריידל, הוא מחפש בתיקיה הנוכחית קובץ בשם build.gradle, ומחפש בתוכו משימות (task) לביצוע. לדוגמה, בתיקיה 01 יש קובץ build.gradle עם שלוש משימות: hello1, hello2, hello3. כל משימה מוגדרת בתחביר קצת שונה, אבל למעשה הן עושות כמעט אותו הדבר. hello1 מוגדרת בתחביר דמוי-ג'אבה, hello2 מוגדרת בתחביר ג'אבה מקוצר (עם פחות נקודות-פסיק וסוגריים) שנקרא Groovy, ו-hello3 מוגדרת בתחביר מקוצר עוד יותר. התחביר השלישי הוא המקובל בגריידל, אבל כדאי לדעת שמאחרי הקלעים מסתתרת תוכנית ג'אבה.

אם ניכנס לתיקיה בחלון command ונכתוב

```
gradle hello1
```

נראה שמשימה 1 אכן התבצעה. באותו אופן אפשר להריץ את שאר המשימות.

תלויות - תיקייה 12

התפקיד העיקרי שלשמו המציאו את גריידל הוא לנהל **תלויות** בין משימות. נתחיל מדוגמה פשוטה (אל תדאגו, זה יסתבך בהמשך...). אנחנו צריכים לנעול נעליים. לשם כך צריך קודם לגרוב גרביים. אבל כדי למצוא את הנעליים ואת הגרביים, חייבים לפני כן לעשות סדר בחדר. אפשר לשקף עובדות אלה ע"י הפעולה dependsOn, באופן הבא:

```
task putOnShoes {
    dependsOn "putOnSocks", "makeOrderInRoom"
    doLast {println "Putting on Shoes."}
}

task makeOrderInRoom {
    doLast {println "Making order in room."}
}

task putOnSocks {
    dependsOn makeOrderInRoom
    doLast {println "Putting on Socks."}
```

```
}
```

עכשיו, כשנריץ

```
gradle putOnShoes
```

גריידל קודם יסדר את החדר, אחר-כך יגרוב גרביים ובסוף ינעל נעליים.

dependsOn מגדיר פעולה החייבת להתבצע **לפני** הפעולה הנוכחית; בדומה לכך, finalizeBy מגדיר פעולה החייבת להתבצע **אחרי** הפעולה הנוכחית. לדוגמה, אחרי שאוכלים ארוחת בוקר צריך לצחצח שיניים:

```
task eatBreakfast {
    finalizedBy "brushYourTeeth"
    doLast{println "Om nom nom breakfast!"}
}

task brushYourTeeth {
    doLast {println "Brushie Brushie Brushie."}
}
```

נסו להריץ gradle eatBreakfast ותראו מה קורה.

סוג שלישי של תלות הוא mustRunAfter. הוא מגדיר פעולה החייבת להתבצע לפני הפעולה הנוכחית, אבל רק אם שתיהן רצות יחד. למשל, השורה:

```
putOnShoes.mustRunAfter takeShower
```

קובעת שאם רוצים גם להתקלח וגם לנעול נעליים, צריך קודם להתקלח.
הערה: משום-מה, הפעולה המקבילה mustRunBefore לא קיימת.

תרגיל כיתה: נסו ליצור תלות מעגלית בין משימות ובדקו מה קורה.

קבצים - תיקייה 13

עד עכשיו עבדנו עם משימות פשוטות. בגריידל ישנן גם משימות מורכבות יותר. ישנן כמה משימות לביצוע פעולות נפוצות בקבצים, כגון: העתקה, מחיקה ואריזה. כדי להשתמש במשימות מסוגים אלה, צריך להעביר פרמטר type בהגדרת המשימה. למשל, הקטע הבא:

```
task copyImages(type: Copy) {

    from 'images'

    into 'build'

}
```

מגדיר משימה בשם copyImages שמעתיקה את כל הקבצים מהתיקיה images לתיקיה build (אם התיקיה לא קיימת, גריידל יידע ליצור אותה).

ניתן להעביר פרמטרים נוספים למשימת העתקה, הקובעים איזה קבצים בדיוק להעתיק, למשל, אם נוסיף למשימה למעלה את השורה:

```
include '*.jpg'
```

יועתקו רק קבצי jpg, ואם נוסיף

```
exclude '*.jpg'
```

יועתקו רק הקבצים שאינם jpg.

הקטע הבא:

```
task clean(type: Delete) {
    delete 'build'
}
```

מגדיר משימה בשם clean המוחקת את התיקיה build.

משימות מסוג Jar, Zip משמשות לדחיסת קבצים וארזיתם. כדי לפתוח אריזה יש להשתמש במשימה מסוג Copy. ראו דוגמאות בקובץ.

גריידל יודע לבצע **בנייה אינקרמנטלית** (incremental build): אחרי שהוא ביצע משימה כלשהי, הוא יבצע אותה מחדש רק אם יש צורך. לדוגמה, נסו להריץ פעמיים ברציפות את המשימה copyJpegs. בפעם הראשונה מקבלים:

```
1 actionable task: 1 executed
```

כלומר משימה אחת בוצעה. בפעם השניה מקבלים:

```
1 actionable task: 1 up-to-date
```

כלומר משימה אחת לא בוצעה, כי המצב הקיים הוא עדכני. מתי המצב הקיים לא עדכני? בשני מקרים:

- כשהקלט השתנה - למשל נוסף קובץ חדש לתיקיה שממנה מעתיקים, או השתנה קובץ קיים.
- כשהפלט השתנה - למשל נמחק קובץ מהתיקיה שאליה מעתיקים, או השתנה קובץ קיים.

בשני המקרים הללו, המשימה תתבצע מחדש - נסו ותראו.

משימות עם פרמטרים - תיקייה 14

אפשר להעביר פרמטרים לגריידל משורת הפקודה. למשל, נניח בקובץ גריידל מוגדרת המשימה:

```
task greet {
    doLast {println greeting+" world"}
}
```

אם נריץ gradle greet נקבל הודעת שגיאה על כך שהמשתנה greeting לא מוגדר, אבל אם נריץ:

```
gradle -Pgreeting=hi greet
```

נקבל "hi world" - המתנה עבר משורת הפקודה לגריידל.

משימות ג'אבה - תיקייה 21

הגיע הזמן לעלות ברמה ולהריץ משימות מהסוג שמעניין אותנו - משימות של ג'אבה. גריידל יכול לעבוד עם הרבה שפות, בתנאי שמוסיפים לו את התוסף (plugin) המתאים. התוסף של ג'אבה בא יחד עם ההתקנה הרגילה של גריידל, כל מה שצריך לעשות כדי להשתמש בו זה לכתוב את השורה הבאה בראש הקובץ:

```
apply plugin: "java"
```

השורה הזאת כבר מוסיפה לנו הרבה משימות הקשורות לניהול פרוייקט ג'אבה, עם התלויות ביניהן. ניתן לראות תרשים תלויות כאן:

https://docs.gradle.org/current/userguide/img/javaPluginTasks.png

ההסבר כאן:

https://docs.gradle.org/current/userguide/java_plugin.html#sec:java_tasks

מבט בתרשים מאפשר לנו להעריך את המורכבות של בניית פרוייקט ג'אבה: צריך לקמפל קבצי ג'אבה, לעבד משאבים כלשהם הדרושים להרצת התוכנית (כגון קבצי קלט), לבנות את המחלקות, לארוז את המחלקות ב-jar לצורך הפצה, לחבר לזה את הרכיבים החיצוניים שהתוכנה שלנו משתמשת בהם, לקמפל את קבצי הבדיקות, לעבד את המשאבים הדרושים להרצת הבדיקות, להריץ את הבדיקות ליצור את התיעוד (javadoc), ועוד...

בפרוייקט גריידל, מקובל לארגן את קבצי ג'אבה במבנה מסויים של תיקיות:

- src
  - main
    - java
      - package1
      - package2 ...
  - test
    - java
      - package1
      - package2 ...

אפשר לשנות את זה, אבל זו ברירת-המחדל. כדוגמה, העתקתי את הקבצים שדיברנו עליהם בתחילת השיעור (על כתיבה וקריאה של אובייקטים) לתיקייה 21 לתוך src/java/main (בינתיים לא העתקתי את קבצי הבדיקות). הוספתי לקובץ build.gradle שתי משימות-הרצה - אחת מריצה את המחלקה הראשית WritePoint הכותבת נקודה לקובץ ע"י writeObject, והשניה מריצה את המחלקה הראשית ReadPoint הקוראת נקודה מהקובץ ע"י readObject:

```
task write(type: JavaExec) {
    main = "lesson10.points.WritePoint"
    classpath = sourceSets.main.runtimeClasspath
}
task read(type: JavaExec) {
    dependsOn write
    main = "lesson10.points.ReadPoint"
```

```
    classpath = sourceSets.main.runtimeClasspath
}
```

שימו לב שהמשימה השניה תלויה בראשונה. עכשיו, אם נריץ:

```
gradle read
```

גריידל יקמפל את כל המחלקות שב-main/java, יריץ את WritePoint ואז את ReadPoint.

דרך נוספת להריץ מחלקות בג'אבה היא לארוז אותן בקובץ jar; זה שימושי לצורך העברת התוכנה למחשב אחר. לשם כך ניתן להריץ את המשימה:

```
gradle assemble
```

המשימה יוצרת, בתיקיה build/libs, קובץ jar המכיל את כל המחלקות ב-main/java. אפשר להריץ מחלקות מתוך האריזה ע"י הפקודה הבאה:

```
java -cp build/libs/<jar-name>.jar <main-class-name>
```

למשל:

```
java -cp build/libs/21-java.jar lesson10.points.ReadPoint
```

תרגיל כיתה: מה צריך לשנות בקובץ build.gradle כך שגריידל יכתוב "hello" לפני תחילת פעולת assemble ו"goodbye" אחרי שהסתיימה?

תשובה: להוסיף שתי משימות:

```
task hello { doLast {println "hello!"} }
task goodbye { doLast {println "goodbye!"} }
```

ולהוסיף תלויות למשימות של ג'אבה (זיכרו שכל משימה היא בסה"כ אובייקט בג'אבה):

```
assemble.dependsOn(hello)
assemble.finalizedBy(goodbye)
```

מאגרי תוספים - תיקייה 22

אחד היתרונות הגדולים של שימוש בגריידל הוא שמקבלים גישה למאגרים עצומים של תוספים לג'אבה.

לדוגמה, ראינו בתחילת השיעור את התוסף GSON המאפשר לכתוב ולקרוא עצמים בפורמט JSON. כשעבדנו באקליפס, הוספנו אותו לתוכנה שלנו ע"י הורדת קובץ jar והוספתו ל-build path. החיסרון הוא, שכאשר אנחנו מעבירים את התוכנה שלנו לשותפים או לקוחות, אנחנו צריכים להעביר יחד איתה את ה-jar. חיסרון נוסף הוא, שאם רוצים להשתמש בתוסף שהוא בעצמו תלוי בתוספים נוספים, נצטרך להוריד את כולם אחד אחד.

בגריידל כל התהליך הזה הרבה יותר פשוט. כל מה שצריך לעשות כדי להשתמש ב-GSON הוא:

א. להוסיף קישור למאגר של תוספים. ישנם כמה מאגרים, הכי גדול הוא mavenCentral שאפשר להוסיף כך:

```
repositories {
    mavenCentral()
}
```

יש גם מאגרים נוספים כגון jcenter.

ב. להכניס תלות בתוסף מסויים מתוך המאגר, למשל:

```
dependencies {
    compile    group: 'com.google.code.gson', name: 'gson',
version: '2.8.2'
}
```

המשמעות היא, שהפעולה "compile" תלויה ברכיב מהקבוצה "com.google.code.gson", שם הרכיב הוא "gson" והגירסה היא "2.8.2".

איך ידעתי מה לכתוב שם? פשוט, חיפשתי "GSON maven", מצאתי את הגירסה המעודכנת ביותר כאן:

https://mvnrepository.com/artifact/com.google.code.gson/gson/2.8.2

והעתקתי את הטקסט המתאים לגריידל (יש שם טקסטים אחרים המתאימים למערכות-בנייה אחרות).

אפשר גם לקצר את השורה ל:

```
dependencies {
    compile    'com.google.code.gson:gson:2.8.2'
}
```

עכשיו, אם נריץ gradle GsonDemo, נראה שגריידל קודם-כל מוריד כמה חבילות מהאתר של maven, אחר-כך מקמפל ובסוף מריץ. אם נריץ שוב אותה פקודה, גריידל כבר לא יצטרך להוריד שוב את החבילות - הן כבר שמורות במחשב.

(איך למחוק את החבילות שגריידל הוריד כדי לפנות מקום על המחשב? בלינוקס, ניתן להריץ rm -rf ~/.gradle/caches. במערכות הפעלה אחרות - צריך לבדוק).

איך בוחרים איזו גירסה להריץ? אפשר פשוט לבחור את הגירסה המעודכנת ביותר שנמצאת עכשיו במאגר, ולכתוב אותה (למשל במקרה של GSON הגירסה הכי מעודכנת שמצאתי היתה 2.8.2). אבל לפעמים אנחנו צריכים דווקא להריץ גירסה ישנה יותר ואז אפשר לבחור גירסה אחרת.

אם כותבים את מספר הגירסה המלא, התוכנה שלנו תמיד תריץ את אותה גירסה - גם כשיהיו גירסאות חדשות. אם אנחנו רוצים שהתוכנה שלנו תתעדכן באופן קבוע כשיש גירסאות חדשות, אפשר לכתוב

במקום מספר הגירסה "+", זה נותן הוראה לגריידל להוריד את הגירסה החדשה ביותר. אפשר גם לכתוב "2.+", ואז תרד הגירסה החדשה ביותר שמתחילה ב-2 (אבל לא נעבור אוטומטית לגירסה 3).

כדי לראות את רשימת התלויות של הפרוייקט שלנו, ניתן להריץ:

```
gradle dependencies
```

במקרה שלנו, נראה משימות שאינן תלויות בשום דבר (למשל compileOnly - קימפול המחלקות שלנו בלבד), ומשימות התלויות ב-GSON, למשל:

```
compileClasspath - ...
\--- com.google.code.gson:gson:+ -> 2.8.2
```

זה ציור של עץ קצר מאד - עם שורש אחד ועלה אחד בלבד. המשמעות היא שהקומפילציה תלויה בחבילה gson, אנחנו ביקשנו את הגירסה החדשה ביותר (+), וגריידל מצא שהחבילה העדכנית ביותר היא 2.8.2.

תלויות מורכבות - תיקייה 23

בואו נראה רכיב עם מערכת תלויות מורכבת יותר. בתיקיה 23 הוספתי את תלות ברכיב שנקרא ספארק:

```
dependencies {
    compile 'com.sparkjava:spark-core:+'
}
```

הרכיב הזה מאפשר לבנות שרתי-רשת בקלות ובנוחות (הרבה יותר מהשרתים שבנינו בשיעורים הקודמים).

לדוגמה ראו בקובץ src/main/java/lesson10/Hello.java.

בקובץ build.gradle יש משימה בשם hello שמריצה את התוכנית הראשית בקובץ זה.

נריץ gradle hello ונראה שגריידל מוריד המון חבילות, מקמפל ומריץ את התוכנה שלנו. הרצנו שרת ווב! אפשר לבדוק אותו בכתובת: http://localhost:8080/hello.

נריץ gradle dependencies ונראה פתאום עץ ענק של תלויות! מתברר שהחבילה עם השם הקצר spark למעשה תלויה בהמון חבילות אחרות, כגון jetty, websocket. גריידל הוריד עבורנו את כל החבילות האלו. אילו היינו צריכים לחפש כל חבילה כזאת, להוריד אותה בעצמנו ולשים באקליפס, היה לוקח לנו שנים (טוב, אולי לא ממש שנים, אבל הרבה מאד זמן).

בדיקות - תיקיה 24

הרצת בדיקות היא חלק בלתי נפרד מתהליך הבניה של פרוייקטים. בגריידל מקובל לשים את הבדיקות בתיקיה נפרדת בשם test, מקבילה לתיקיה main (ראו מבנה למעלה).
כדי להריץ בדיקות JUnit אנחנו צריכים להכליל את התוסף של junit, שאפשר להוריד מ-mavenCentral באופן הבא (אם רוצים את הגירסה החדשה ביותר שמתחילה ב-4):

```
repositories {
    mavenCentral()
}
```

```
dependencies {
    testCompile 'junit:junit:4.+'
}
```

שימו לב: התלות היא רק עבור testCompile. כלומר מי שרוצה רק לקמפל ולהריץ את התוכנית שלנו, לא יצטרך את JUnit. רק מי שרוצה להריץ בדיקות יצטרך את JUnit, וגריידל יבין את זה לבד ויוריד רק את החבילות שאנחנו צריכים.

נסו בעצמכם: הריצו gradle read ותראו שאין זכר ל junit; הריצו gradle test ותראו שגריידל מוריד את junit.

אם הבדיקות לא עוברות - גריידל יודיע לנו שהבנייה נכשלה (BUILD FAILED) כי היו בדיקות שלא עברו (There were failing tests), ויתן לנו קישור לקובץ על המחשב שלנו שמראה פירוט של הבדיקות שנכשלו.

חידה: מה צריך לעשות כדי שגריידל יכתוב הודעה משמחת רק במקרה שכל הבדיקות עברו?

תשובה:

```
task success {  doLast {println "All tests passed!"} }

test.finalizedBy(success)
```

טריקים נוספים - תיקיה 25

1. אפשר לעבוד עם גריידל ועם אקליפס במקביל. דרך אחת לעשות זאת היא:

א. להשתמש תוסף-אקליפס מתוך קובץ גריידל, באופן הבא:

```
apply plugin: 'java'
apply plugin: 'eclipse'
```

ב. להריץ את המשימה gradle eclipse

המשימה יוצרת קבצי הגדרת פרוייקט של אקליפס - project, classpath ועוד אחד עם שם ארוך. עכשיו, אפשר לייבא את הפרוייקט לתוך אקליפס, לעבוד לקמפל ולהריץ כרגיל.

יש גם תוסף-גריידל שאפשר להוסיף לאקליפס. הוא מוסיף את התצוגה Gradle Tasks שממנה ניתן להריץ משימות גריידל. **שימו לב**: החלון מראה רק משימות שיש להן קבוצה (group).

2. כשעובדים עם שותפים על פרוייקט גריידל, חשוב לוודא שיש להם אותה גירסת גריידל שיש לנו. לשם כך אפשר ליצור "עטיפה" ע"י הפעולה:

```
gradle wrapper
```

הפעולה הזאת יוצרת כמה קבצים: קובץ jar, קובץ properties, ועוד שני קבצי הרצה:

- gradlew - להרצה על מערכות מבוססות לינוקס,
- gradlew.bat - להרצה על מערכות מבוססות חלונות.

אנחנו נשמור את הקבצים האלה בבקרת-תצורה (למשל גיט) יחד עם הפרוייקט. כל מי שיוריד את הפרוייקט, יוכל להריץ gradlew או gradlew.bat בהתאם למערכת ההפעלה שלו; הסקריפט יוריד ויתקין אצלו את הגירסה המתאימה של גריידל (בהתאם לגירסה שאצלנו), וכן יריץ את המשימה המתאימה עם כל התלויות שלה.

מקורות

- Gradle for Android and Java, Udacity course: https://classroom.udacity.com/courses/ud867

סיכם: אראל סגל-הלוי.