BUT-INFO-R4.01

None

Nicolas Le Sommer

Table of contents

1. Accueil

1.1

Resource R4.01: Architecture Logicielle

Nicolas Le Sommer, Matthieu Le Lain et Serge Dubois

1.1.1 1. Contenu du programme national du BUT

L'objectif de cette ressource est de présenter des composants de programmation qui peuvent être utilisés dans plusieurs domaines.

Les savoirs de référence étudiés dans cette ressource sont:

- les patrons d'architecture (MVC, MVVM, ...);
- l'utilisation de briques logicielles, d'interfaces de programmation et de bibliothèques tierces;
- le développement de services Web.

1.1.2 2. Organisation des enseignements

Deux séances de 1h30. Un cours de 45' sera intégré à l'une des deux séances. Il y a 6 semaines d'enseignement.

1.1.3 3. Les supports de cours et de travaux pratiques

- Supports de cours
- Sujets de travaux pratiques

1.1.4 4. Modalités d'évaluation

- Note de module \(= \frac{1}{3} *\) note CC \(+ \frac{2}{3} *\) note CT
- Contrôle terminal en semaine 42

2. Cours

2.1 Supports de cours de la ressource R4.01

Nicolas Le Sommer, Matthieu Le Lain et Serge Dubois

Le cours aborde les patrons architecturaux, la gestion des dépendances logicielles, la programmation orientée services, les services Web, et la sécurisation d'une API REST.

Support de cours

3. Travaux pratiques

3.1 Présentation des travaux

3.1.1 Travaux pratiques de la ressource R4.01

Nicolas Le Sommer, Matthieu Le Lain et Serge Dubois

Dans le cadre de travaux pratiques de la ressource R4.01, vous allez développer des applications en utilisant des bibliothèques logicielles, des cadriciels et des boîtes à outils, et des approches à base de micro-services. Vous étudierez et utiliserez également des outils vous permettant de gérer le cycle de développement d'un projet Java, comme Maven.

> **Note**
>
> Pour ces travaux pratiques, assurez-vous avoir une version de JDK supérieure ou égale à 16.

2. Planning des séances

Semaines	Séances	Travail à réaliser
14	1 et 2	Les outils Ant, Maven et Gradle
15	3 et 4	Mise en œuvre d'une application Web reposant sur le patron MVVM avec le cadriciel ZK
16	5 et 6	Développement d'un micro-service REST avec Spring Boot
17	/	Vacances
18	/	Vacances
19	7 et 8	Développement d'une API REST conforme à la spécification OpenAPI
20	9 et 10	Développement d'une application à base de micro-services
21	11 et 12	Développement d'une application à base de micro-services

3. Rendus

- semaine 14: Application JSON et Maven
- semaine 15: Application Web ZK et Maven
- semaine 16: Micro-service SpringBoot
- semaine 19: Micro-service SpringBoot OpenAPI
- semaine 21: Application à base de micro-services SpringBoot

3.2 Ant, Maven et Gradle

Nicolas Le Sommer, Matthieu Le Lain et Serge Dubois

Lors du développement (i.e. lors de la compilation et des tests) et de l'exécution d'une application Java complexe, il faut pouvoir définir le "classpath" listant les répertoires et les archives (jar) contenant les interfaces, les classes et les ressources (fichiers de configuration, images, ...) utilisées par l'application. Il faut également pouvoir automatiser la production de la documentation, et exécuter des tâches (création d'archives, signature des archives Jar, ...).

Dans le cadre de travaux pratiques de la ressource R4.01, nous utiliserons des bibliothèques logicielles et des cadriciels Java.

Les outils Ant, Maven et Gradle peuvent être utilisés pour faciliter et automatiser le développement et l'exécution d'applications Java complexes.

Dans le cadre de ces travaux pratiques, nous utiliserons l'outil Maven.

1. Ant

1. PRÉSENTATION DE L'OUTIL ANT

Ant est un outil développé en Java par l'ASF (*Apache Software Foundation*) pour gérer le développement d'applications Java. Cet outil permet d'effectuer des tâches telles que la compilation, la construction d'archives, l'exécution de tests. Ces tâches sont décrites dans un fichier au format XML. Ce fichier se nomme habituellement `build.xml`.

Ce fichier définit un ensemble de cibles pour construire le projet (e.g. compilation, exécution, exécution des tests, génération de la documentation). Une cible est composée de tâches qui peuvent être exécutées en parallèle ou de manière séquentielle.

Il y a des commandes usuelles écrites en Java qui peuvent être utilisées dans les tâches (javac, jar, copy, ...).

1.1 STRUCTURE DU FICHIER BUILD.XML

Les propriétés du projet peuvent être définies dans un fichier de propriétés Java, fichier qui sera référencé dans le fichier `build.xml`. Dans l'exemple ci-dessous de fichier de propriété est nommé `project.properties`. Ce fichier définit notamment les propriétés `classpath`, `src.dir`, `classes.dir` dans l'exemple ci-dessous.

```
<?xml version="1.0" encoding="UTF-8" ?>
<project name="myproj" default="compile" basedir=".">
    <description> The description of the application </description>

    <!-- Definition of properties -->
    <property file="project.properties"/>

    <!-- Compilation of Java source files -->
    <target name        = "compile"
            description = "Compiles java classes, output goes to the ${classes.dir}"
    >
       <mkdir dir = "${classes.dir}" />

       <javac destdir             = "${classes.dir}"
              debug               = "${javac.debug}"
              optimize            = "${javac.optimize}"
              includeJavaRuntime  = "${javac.include_java_runtime}"
              compiler            = "${javac.compiler}"
        >
            <classpath   path = "${classpath}"    />
            <src         path = "${src.dir}"            />
            <include     name = "**/*.java"             />
            <compilerarg line = "${javac.compile_args}" />
        </javac>

    <echo level = "verbose" message = "The java source files are compiled..." />
    </target>
</project>
```

1.2 ORDRE D'EXÉCUTION DES CIBLES

Certaines cibles peuvent dépendre d'autres cibles. Dans l'exemple ci-dessous, la cible B dépend de la cible A. Cette cible A sera exécutée avant la cible B. De la même manière, la cible C dépend de la cible B. La cible A et la cible B seront exécutées avant la cible C. L'ordre d'exécution des tâches peut également être précisé. Dans l'exemple ci-dessous, les cibles A et D seront exécutées préalablement à la cible F.

```
<target name="A"/>
<target name="B" depends="A"/>
<target name="C" depends="B"/>
<target name="D"/>
<target name="F" depends="A,D"/>
```

1.3 CONDITIONS D'EXÉCUTION

Des conditions d'exécution peuvent être précisées dans le fichier `build.xml` pour conditionner l'exécution de certaines cibles. Dans l'exemple ci-dessous, les cibles sont conditionnées au type du système d'exploitation utilisés lors de la construction de l'application.

```
<target name="build.linux" if="os.is.linux"/>
<target name="build.no.windows"  unless="os.is.windows"/>
```

1.4 LIMITES D'ANT

L'outil Ant souffre de certaines limitations. En effet, il ne définit pas de structure standard de projet. Il ne définit pas de cycle de vie d'un projet. Il faut construire un ensemble de cibles pour effectuer la compilation, les tests, etc. Il faut définir des dépendances entre les cibles pour définir un cycle de vie. Il n'y a pas de gestion des dépendances logicielles.

2. Maven

2.1 PRÉSENTATION DE L'OUTIL MAVEN

Maven est un outil développé en Java par l'ASF (*Apache Software Foundation*) pour gérer le développement d'applications Java, et répondre aux limitations de l'outil Ant. Maven propose une structure standard de projet, un cycle de vie standard de projet, et une gestion des dépendances logicielles.

L'outil Maven est un outil extensible. Des greffons peuvent être utilisés pour réaliser de nouvelles tâches. Il existe des greffons pour la compilation, les tests, la documentation, etc.

2.2 DÉPÔTS D'ARCHIVES

Pour permettre le partage et la réutilisation de librairies, de cadriciels et de boîtes à outils, Maven repose sur plusieurs dépôts:

- des dépôts globaux (http://central.maven.org/maven2/),
- des dépôts locaux à une entreprise, une organisation, etc.,
- un dépôt personnel (`~/.m2`).

Le site Web https://mvnrepository.com peut être utilisé pour rechercher des librairies, des cadriciels et des boîtes à outils.

2.3 LE MODÈLE STANDARD DE PROJET

L'outil Maven définit un modèle de projet standard (*POM: Project Object Model*). Ce modèle est une représentation XML d'un projet Maven, qui est décrite dans un fichier `pom.xml`; Un projet Maven est identifié par des coordonnées. Il définit le type paquetage qui sera produit (e.g. jar, war, bundle, ear), les dépendances logicielles tierces et les directives de constructions et de gestion du projet.

Un exemple de fichier `pom.xml` est donné ci-dessous:

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
                             http://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>
    <parent>
        <groupId>fr.univ.ubs</groupId>
        <artifactId>fr.univ.ubs.app</artifactId>
        <version>1.0.0</version>
    </parent>

    <artifactId>fr.univ.ubs.app.framework</artifactId>
    <packaging>jar</packaging>
    <name>The framework</name>
```

```
    <description> ...</description>

    <dependencies>
        <dependency>
            <groupId>org.json</groupId>
            <artifactId>json</artifactId>
            <version>20220924</version>
            <scope>provided</scope>
        </dependency>
    </dependencies>

</project>
```

2.4 IDENTIFICATION D'UN PROJET MAVEN

Un projet Maven est identifié par un triplet (`groupId`, `artifactId`, `version`). Le `groupId` permet d'identifier le groupe, l'organisation ou l'entreprise qui développe le logiciel. Par exemple pour l'outil Maven, le `groupId` est `org.apache.maven`. Cet identifiant est, comme pour Java, habituellement l'URL inversée du site Web du projet, ou du nom de domaine inversé de l'entreprise ou de l'organisation.

L'`artifactId` permet d'identifier de manière unique un développement effectué par un groupe, une organisation ou une entreprise. Enfin, un développement est identifié par son numéro de version.

2.5 FONCTIONNEMENT GÉNÉRAL DE L'OUTIL MAVEN

L'outil Maven peut être étendu à l'aide de greffons. Les greffons définissent un ensemble de buts (*goals*) permettant de réaliser des tâches (compilation, exécution des tests, etc.). Le but d'un greffon peut être exécuté de la manière suivante:

```
mvn greffon:but
```

Les paramètres de configuration d'un greffon peuvent être définis dans le fichier `pom.xml`, mais aussi sur la ligne de commande `mvn` avec l'option `-Dparam=valeur`.

Un but peut être exécuté lors d'une phase du cycle de vie du projet. Plusieurs phases sont prédéfinies dans Maven: `validate`, `compile`, `test`, `package`, `verify`, `install`, `deploy`. Ces phases peuvent être exécutées à l'aide de la ligne de commande `mvn NOM_PHASE`. Pour exécuter la phase de compilation on exécutera la commande `mvn compile`.

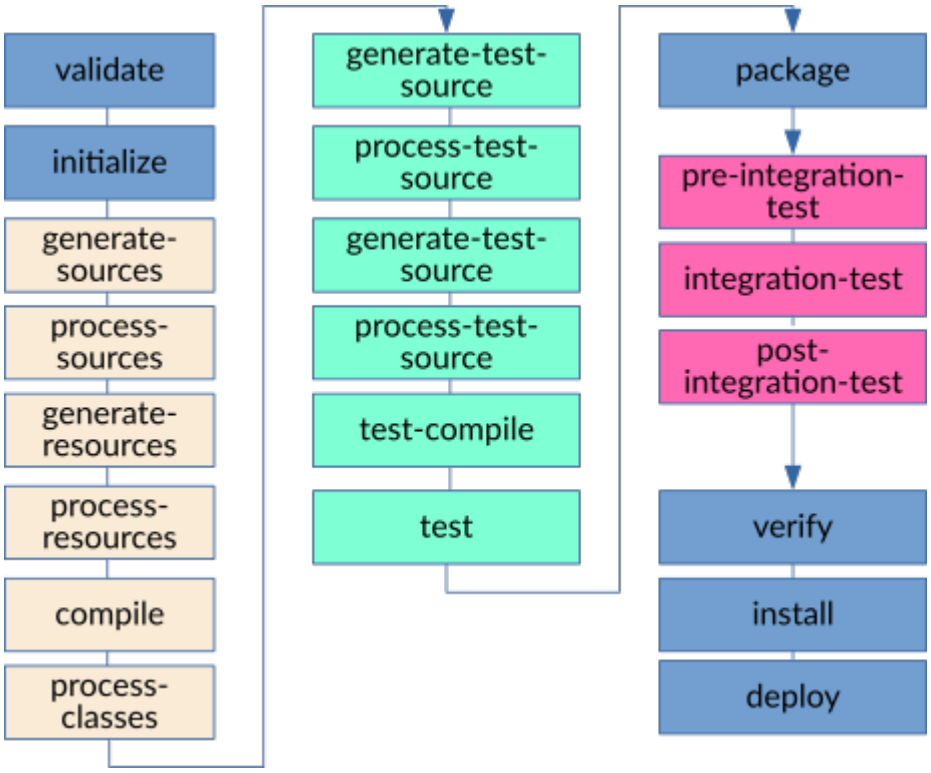
La commande `mvn greffon:but` entraîne uniquement l'exécution du but du greffon considéré.

La commande mvn phase entraîne l'exécution du (des) but(s) associé(s), et l'exécution de la phase précédente (transitivité).

À chaque phase est associé un ou plusieurs buts d'un ou plusieurs greffon(s). La phase clean entraînera l'exécution des buts suivants:

`clean = pre-clean → clean → post-clean`

Le cycle de vie de l'outil Maven est illustré dans la figure ci-dessous:



- `mvn compile` permettra de compiler les sources de l'application Java.
- `mvn test` exécutera les tests unitaires du projet. `mvn test` entraîne la phase `compile`.
- `mvn package` créera le paquetage du logiciel développé (i.e. archive jar, war, ear).
- `mvn install` installera le paquetage du logiciel dans votre dépôt local `~/.m2`, permettant ainsi sa résolution et son utilisation par d'autres projets locaux qui expriment une dépendance vis-à-vis de celui-ci.
- `mvn deploy` déploiera le paquetage sur un dépôt distant (dépôt d'entreprise, dépôt global).

2.6 DÉPENDANCES LOGICIELLES

Les dépendances concernent les artefacts et les greffons. La résolution des dépendances est transitive. Ce expression et résolution des dépendances permet de gérer le classpath pour la compiltation des sources du projet, l'exécution des tests, et éventuellement l'exécution du projet. Les dépendances sont identifiées par leur triplet (`groupId`, `artifactId`, `version`).

Un exemple d'expression de dépendances est donnée ci-dessous:

```
...
<groupId>fr.univ.ubs.app</groupId>
<artifactId>myapp</artifactId>
<version>1.0.0</version>
...
<dependencies>
<dependency>
<groupId>org.osgi</groupId>
<artifactId>compendium</artifactId>
<version>4.0.0</version>
</dependency>
</dependencies>
...
```

Les dépendances peuvent avoir une portée. Les niveaux de portée considérés sont:

portée	description
`compile`	Précise que la dépendance sera disponible dans tous les classpath (compilation, test, exécution); Elle sera transitive vers les projets dépendants (défaut)
`provided`	Indique que la dépendance sera disponible dans les classpath des phases de compilation et de test; Il n'y a pas de transitivité.
`runtime`	Indique que la dépendance sera disponible dans les classpath des phases d'exécution de l'application et de tests, mais pas à la compilation. Il n'y a pas de transitivité.
`test`	Précise que la dépendance sera disponible pour la compilation et l'exécution des tests. Il n'y a pas de transitivité.
`runtime`	La portée est similaire à `provided`, mais précise que l'archive Jar doit être fournie par le système. Elle ne sera pas téléchargée depuis un dépôt distant.

Dans l'exemple ci-dessus, l'application `myapp` dépend de (`org.osgi`, `compedium`, `4.0.0`), qui lui dépend de (`org.osgi`, `core`, `4.0.0`). Les 2 archives Jar contenant ces cadriciels seront téléchargées depuis un dépôt distant si elles ne sont pas présentes dans le dépôt local pour pouvoir être utilisées lors des phases de compilation, de tests et d'exécution.

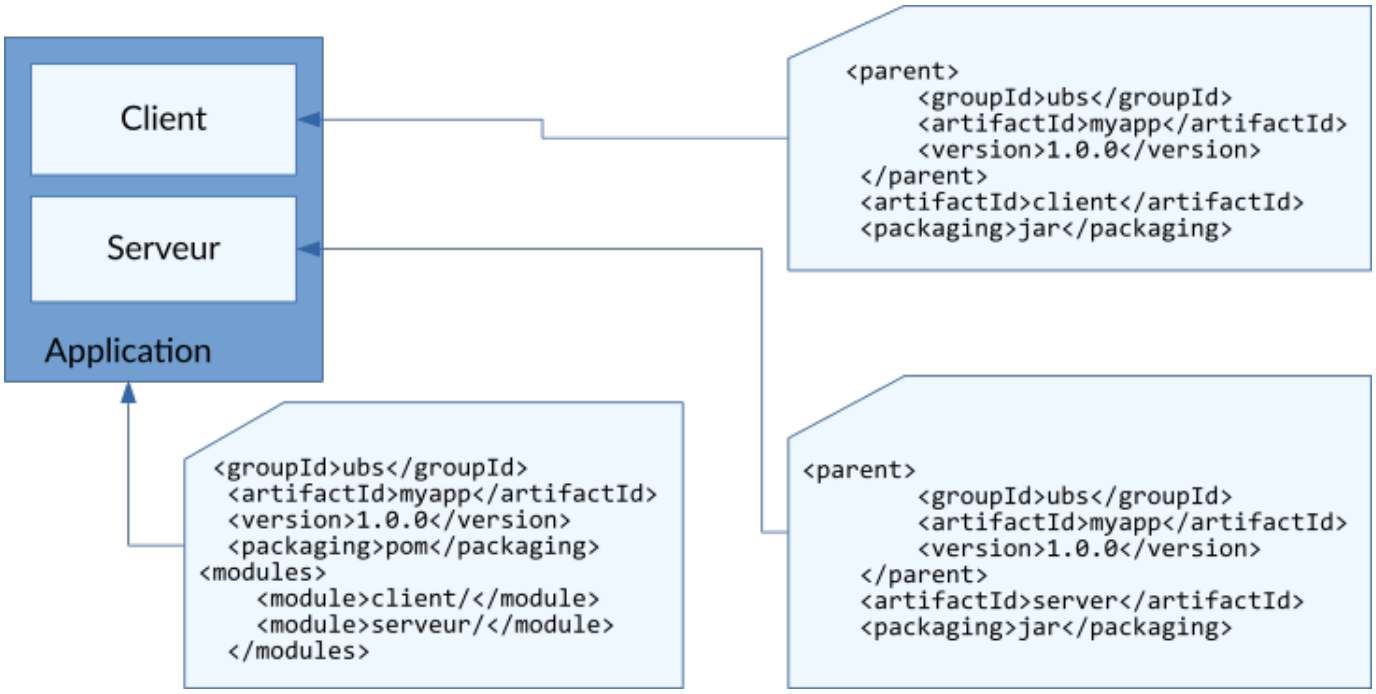
Des conditions peuvent être exprimées sur les numéros de version des dépendances.

conditions et versions	description
1.2.1	si possible 1.2.1
[1.0, 1.2]	de 1.0 à 1.2 inclus
(1.0, 1.2)	à partir de 1.0, mais avant 1.2
(, 1.2)	antérieur à 1.2
[1.0,]	à partir de 1.0
[1.0]	1.0 absolument
[1.2,)	plus grand ou égal à 1.2
(, 1.1), (1.1,)	toutes les versions sauf 1.1

2.7 DÉVELOPPEMENT MODULAIRE ET HIÉRARCHIE DE PROJETS

Une application peut être composée de plusieurs parties, comme par exemple une partie cliente et une partie serveur. Il peut être utile de regrouper ces 2 parties au sein d'un même projet, tout en permettant un développement séparé de ces deux parties (compilation, exécution des tests et des parties de manière distincte). Il peut également être intéressant de pouvoir compiler, tester et exécuter ces deux parties simultanément. C'est ce que permet de faire Maven à travers une hiérarchie de fichier `pom.xml` et la notion de modules.

La même remarque peut s'appliquer pour un cadriciel et la boîte à outils qui lui est associée.



Dans l'exemple ci-dessus, l'application est composée de 2 parties: une partie cliente et une partie serveur. La partie cliente et la partie serveur sont chacune développée dans un projet Maven spécifique. Ces 2 projets sont regroupés au sein d'un projet parent, et sont référencés à travers des modules. Le nom des modules font références au nom des répertoires contenant les projets `client` et `serveur`.

Les projets "enfants"/modules peuvent hériter d'une partie des éléments identifiant le projet parent (`groupId` et `version`). Des dépendances communes aux projets enfants peuvent être indiquées dans le projet parent.

Les projets enfant peuvent être compilés, testés et exécutés indépendamment en exécutant les commandes appropriées dans leur répertoire propre (e.g. `mvn compile`, `mvn test`).

Pour compiler, tester et exécuter l'ensemble des projets enfants, les commandes (`mvn compile`, `mvn test`, etc.) peuvent être exécuté dans le répertoire du projet parent.

3. Gradle

Gradle est un outil permettant notamment de gérer le cycle de développement d'applications Java. C'est notamment l'outil utilisé pour créer des applications mobiles Android.

Gradle permet de lever certaines limites de l'outil Maven, comme par exemple la gestion des conflits entre des versions différentes d'un même artefact, ou la difficulté d'écrire un fichier XML complexe.

Gradle utilise un DSL (*Domain Specific Language*) reposant sur le langage Groovy défini par l'ASF (*Apache Software Foundation*). À l'instar de Maven, Gradle supporte le développement de projets composés de sous projets, et permet de gérer des dépendances vis-à-vis de bibliothèques tierces.

Gradle permet de définir facilement des tâches pour exécuter un ensemble de commandes. Il repose sur des greffons pour gérer le cycle de vie d'applications Java, d'applications mobiles Android, etc. Il offre une déclaration compacte des dépendances.

Le fichier de configuration du projet est le fichier `build.gradle`.

3.1 LES TÂCHES

Les tâches peuvent être facilement décrites dans le fichier `build.gradle`. Les tâches peuvent dépendre d'autres tâches. Pour exécuter une tâche, la commande à exécuter est `gradle -q`
`LA\_TACHE`. L'option `-q` est utilisé pour supprimer les messages de journal de dégradé afin que seule la sortie de la tâche.

Dans l'exemple ci-dessous `doLast` est une fermeture d'action.

```
// groupe et version (par défaut le nom de l'artefact et le nom du répertoire)
group = "fr.ubs"
version = "1.0"

// greffons utilisés
apply plugin: 'java'
apply plugin: 'application'

// propriétés de compilation
sourceCompatibility = 1.5
targetCompatibility = 1.5

// liste des dépôts d'archives
repositories {
    mavenLocal()
    mavenCentral()
    jcenter()
}

// définition de la classe principale d'une application Java
application {
        mainClassName = 'com.Main'
}

// tâches définies par l'utiisateur
task A {
    doLast {
        println "TASK A"
    }
}

task B (dependsOn: A) {
    doLast {
        println "TASK B"
    }
}


// paramétrage de tâches existantes (clean et jar)
clean {
    delete "bin"
    delete "storage"
    delete "target"
}

// tâche définie dans le greffon java
jar {
    manifest {
        from 'MANIFEST.MF'
        attributes 'Main-Class': application.mainClassName
    }

}

// définition d'une nouvelle tâche typée
task sourceJar(type: Jar) {
    classifier = 'sources'
    from sourceSets.main.allJava
}

// définition d'une tâche typée avec dépendance
task javadocJar(type: Jar, dependsOn: javadoc) {
    classifier = 'javadoc'
    from javadoc.destinationDir
}


// dépendances
dependencies {
    compile "commons-io:commons-io:2.4"
    testCompile "junit:junit:4.10"
    runtime     files("lib/foo.jar", "lib/bar.jar")
}
```

Dans l'exemple ci-dessus, la tâche B dépend de la tâche A. Le résultat de la commande `gradle -q B` sera:

```
TASK A
TASK B
```

Les tâches définies par les greffons peuvent être exécutées avec commande `gradle`. Ainsi, la tâche de compilation définie par le greffon java peut être exécutée avec la commande `gradle compileJava`.

3.2 LES DÉPENDANCES

Les dépendances sont facilement exprimées grâce au triplet les identifiant `groupId:artifactId:version`. Les dépendances exprimées de la sorte sont téléchargées depuis un dépôt distant pour être installées localement pour utilisation (compilation, exécution des tests, exécution de l'application, etc.).

Il est également possible comme le montre l'exemple précédent de définir des dépendances vis-à-vis d'archives Jar situés localement dans un répertoire.

La gestion des dépendances est transitive. Une option peut être ajoutée pour préciser que l'on ne souhaite pas une résolution transitive des dépendances.

3.3 HIÉRARCHIE DE PROJETS

Il est possible de définir une hiérarchie de projets. Pour cela, il faut lister, à l'aide de l'instruction `include`, dans un fichier `settings.gradle` les répertoires contenant les projets enfants.

```
.
├── api
├── build.gradle
├── services
│   ├── srv1
│   └── srv2
└── settings.gradle
```

Pour l'arborescence de projets définie ci-dessus le fichier `settings.gradle` contiendra:

```
include 'api', ':services:srv1', ':services:srv2'
```

4. Travail à réaliser

Dans le cadre de ce TP, vous allez utiliser l'outil Maven pour automatiser la compilation et l'exécution de l'application sporttrack-model.

4.1. UTILISATION DE MAVEN SUR LES MACHINES FIXES DE L'UNIVERSITÉ

1. Modifiez votre fichier `~/.bashrc` afin d'ajouter l'outil Maven dans votre chemin d'accès aux exécutables (défini par la variable `$PATH`).

```
export PATH=/ubs/soft/apache-maven-3.8.7/bin:$PATH
```

> **Note**
>
> Sur les machines de l'IUT, vous compilerez votre application en procédant de la manière suivante:
>
> ```
> mvn -Dhttps.proxyHost=squidva.univ-ubs.fr -Dhttps.proxyPort=3128 clean package
> ```

4.2 INSTALLATION DE MAVEN SUR VOS MACHINES PERSONNELLES

1. Installez l'outil Maven sur votre machine personnelle. Pour une distribution reposant sur le système de paquets `apk` de Debian (Debian, Ubuntu, Linux Mint, etc.), exécutez les instructions suivantes:

```
sudo apt update; sudo apt install maven
```

Pour une distribution reposant sur le système `rpm` de Red Hat (Fedora, etc.), exécutez les instructions suivantes:

```
sudo dnf upgrade --refresh -y; sudo dnf install maven
```

4.3 DÉVELOPPEMENT DE L'APPLICATION

1. Téléchargez l'archive de l'application sporttrack-model, et créez à la racine du répertoire `sporttrack-model` un fichier *pom.xml* décrivant votre application (`groupId`, `artifactId`, `version`, ...) et permettant de créer un fichier *jar* exécutable. Ajoutez une dépendance Maven vers la librairie `org.json` version `20220924`.
2. Complétez les classes `JSONFileReader`, `JSONFileWriter` et `Main`.

3. Compilez et testez l'exécution en faisant

```
java -jar target/sporttrack-model.jar
```

> **Note**
>
> Vous trouverez un exemple de fichier JSON dans le répertoire `src/main/resources`.

4. Ajoutez le plugin Maven `exec-maven-plugin` dans votre fichier *pom.xml* et configurez le afin d'indiquer la classe principale à exécuter. Vérifiez le bon fonctionnement en faisant `mvn exec:java`
5. Trouvez un plugin Maven permettant de réaliser une archive *jar* contenant les classes de votre application et celles de l'API Json. L'objectif est de produire une archive Jar (fat jar) contenant l'ensemble des classes nécessaires à l'exécution de l'application, et ainsi d'éviter à l'utilisateur de devoir définir lui-même le classpath pour que l'exécution puisse être réalisée sans problème de dépendances.

5. Rendu

Vous devrez déposer sur Moodle une archive `tar.gz` contenant le code source de votre application (fichier *pom.xml* inclu).

3.3 Patron architectural MVVM et application Web

Nicolas Le Sommer, Matthieu Le Lain et Serge Dubois

Dans ce travail, vous allez développer une application Web en Java qui repose sur le cadriciel ZK, et qui utilise les classes `JSONFileReader` et `JSONFileWriter` que vous avez écrites lors du travail pratique précédent. Cette application devra être compilée et assemblée en utilisant l'outil Maven.

1. Téléchargement de l'application et test

1. Téléchargez l'archive sporttrack-webapp.tar.xz, extrayez le contenu de celle-ci dans votre répertoire de travail, et testez celle-ci en exécutant la commande suivante dans le répertoire `sporttrack-webapp`.

```
mvn clean package jetty:run
```

2. Connectez-vous à l'URL http://localhost:8080/sporttrack pour vérifier le bon fonctionnement de cette application Web.

> **Note**
>
> C'est le serveur Web jetty qui est utilisé pour délivrer votre application Web.

2. Ajout de la dépendance vers le modèle de données

1. Dans le travail pratique précédent, vous avez développé des classes Java permettant de lire et écrire des données d'activités depuis et dans un fichier JSON. Afin de rendre ces classes utilisables dans votre application, vous allez installer l'archive dans votre dépôt Maven local (`~/.m2`). Pour ce faire, exécutez la commande suivante dans votre répertoire `sporttrack-model`.

```
mvn clean install
```

2. Ajoutez une dépendance vis-à-vis de cette archive Jar dans le fichier `pom.xml` de votre application Web `sporttrack-webapp`.

1. Testez le bon fonctionnement en exécutant la commande suivante:

```
mvn clean package
```

3. La vue-modèle

1. En utilisant les classes définies `sporttrack-model`, écrivez dans le répertoire `src/main/java/fr/ubs/sporttrack/webapp` une classe `ActivityService` implantant une méthode `findAll()` qui retourne l'ensemble des activités contenues dans le fichier JSON contenu dans le répertoire `src/main/resources/data.json`.
2. Ajoutez à la classe `ActivityService` une fonction `search()` qui retourne une liste d'activités contenant le mot clé indiqué en paramètre de la méthode `search()`.
3. En utilisant les classes définies `sporttrack-model` et la classe `ActivityService`, modifiez la classe `MyViewModel` contenue dans le répertoire `src/main/java/fr/ubs/sporttrack/webapp` afin d'ajouter à celle-ci un attribut `keyword` de type `String`, un attribut `activities` de type `List<Activity>`. Vous ajouterez également une méthode `search()` permettant de rechercher dans la liste d'activités des activités contenant un mot clé donné. Vous utiliserez la classe `ActivityService` pour rechercher des activités et initialiser la liste d'activités présentée à l'utilisateur.

Note

Vous pouvez vous référer à la documentation ci-dessous pour cette étape.

https://www.zkoss.org/wiki/ZK_Getting_Started/Get_ZK_Up_and_Running_with_MVVM

4. L'interface utilisateur

1. Dans ZK, les interfaces utilisateurs sont décrites dans des fichiers XML. Modifiez le fichier `index.zul` contenu dans le répertoire `src/main/webapp/` afin d'afficher la liste des activités, un champ de recherche et un bouton pour démarrer la recherche.
2. Ajoutez les liaisons dynamiques dans l'interface utilisateur pour faire référence aux attributs et méthodes de la classe `MyViewModel`.
3. Testez le bon fonctionnement de votre application en exécutant la commande suivante

```
mvn clean package jetty:run
```

5. Rendu

Vous devrez rendre sur Moodle une archive tar.gz contenant le code de votre application Web, ainsi que le code de votre modèle de données (`sporttrack-model`). Vous veillerez à avoir exécuté la commande `mvn clean` dans chaque répertoire afin de ne rendre que le code source.

3.4 Micro-service REST Spring Boot

Nicolas Le Sommer, Matthieu Le Lain et Serge Dubois

Dans ce travail, vous allez développer un micro-service REST permettant de manipuler les données d'activités sportives. Vous utiliserez pour ce faire le travail que vous avez réalisé lors du premier travail pratique.

Plusieurs cadriciels peuvent être utilisés pour développer des micro-services (e.g. Spring Boot, Micronaut, Javalin, Vertx, OSGi). Spring Boot est le cadriciel le plus populaire dans l'industrie. Vous utiliserez donc ce cadriciel dans ce travail et dans les travaux pratiques à venir.

1. Création d'un micro-service REST avec Spring Boot

1. Créez un nouveau projet Maven à l'aide de la commande suivante:

```
mvn archetype:generate -DgroupId=fr.ubs.sporttrack -DartifactId=sporttrack-spring-boot -DinteractiveMode=false
```

2. Ajoutez une référence parent vers l'artefact `spring-boot-starter-parent` dans le fichier `pom.xml` du projet que vous venez de créer:

```
  <parent>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-parent</artifactId>
    <version>3.0.0</version>
  </parent>
```

3. Ajoutez une dépendance logicielle vers l'artefact `spring-boot-starter-web` dans le fichier `pom.xml` du projet:

```
    <dependency>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-starter-web</artifactId>
      <version>3.0.0</version>
    </dependency>
```

4. Configurez l'artefact `spring-boot-maven-plugin` pour préciser quelle est la classe principale `fr.ubs.sporttrack.SportTrackApp` de l'application Web.

```
<build>
    <finalName>sporttrack-spring-boot</finalName>
    <plugins>
      <plugin>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-maven-plugin</artifactId>

        <configuration>
          <mainClass>fr.ubs.sporttrack.SportTrackApp</mainClass>
        </configuration>

        <executions>
          <execution>
            <goals>
              <goal>repackage</goal>
            </goals>
          </execution>
        </executions>
      </plugin>
    </plugins>
  </build>
```

5. Créez la classe `fr.ubs.sporttrack.SportTrackApp` et annotez la pour en faire une application Spring Boot (annotation `@SpringBootApplication`).

```
package fr.ubs.sporttrack;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication
public class SportTrackApp {
    public static void main(String[] args){
        SpringApplication.run(SportTrackApp.class, args);
    }
}
```

6. Créez dans le répertoire `fr.ubs.sporttrack.controller` une classe `ActivityController.java`, et annotez cette classe avec les annotations `@RestController` et `@RequestMapping("/activities")` pour afficher la liste des activités. Ajoutez dans cette classe un méthode `findAll()` pour retourner une liste des activités. Annotez cette méthode avec l'annotation `@GetMapping("/")`.
7. Testez le bon fonctionnement de votre micro-service en compilant et en exécutant celui-ci à l'aide de l'instruction suivante. Vous pourrez tester le fonctionnement en vous connectant à l'URL http://localhost:8080/activities/.

   ```
   mvn clean spring-boot:run
   ```

   Vous pouvez également utiliser la commande `cURL` pour tester votre micro-service:

   ```
   curl -X GET "http://localhost:8080/activities/"
   ```

   Vous pouvez également produire une archive jar contenant votre micro-service avec la commande `mvn package`, et exécuter ce micro-service avec la commande ci-dessous.

   ```
   java -jar target/sporttrack-spring-boot.jar
   ```

8. Modifiez la classe `ActivityController` afin d'ajouter une méthode `findByKeyword()` permettant d'obtenir toutes les activités ayant le mot clé passé en paramètre de la méthode `findByKeyword()` dans leur description. Vous annoterez cette méthode avec l'annotation `@GetMapping("/{keyword}")`. Vous n'oublierez pas également d'ajouter l'annotation `@PathVariable("keyword")` devant le paramètre de la méthode `findByKeyword()`. Testez le bon fonctionnement de celle-ci.
9. Modifiez la classe `ActivityController` afin d'ajouter une méthode `addActivity()` pour ajouter une nouvelle activité dans votre fichier de données JSON. Vous annoterez cette méthode avec l'annotation `@PostMapping(path="/", consumes="application/json", produces="application/json")`. Vous ajouterez l'annotation `@RequestBody` devant le paramètre `activity` de la méthode `addActivity()`, paramètre qui doit être de type `Activity`.

2. Travail à rendre

Vous devrez déposer sur Moodle une archive tar.gz contenant le code source de votre application. Cette archive devra contenir également un fichier README.md précisant ce que vous avez réussi à développer, et les tests que vous avez effectués. Vous donnerez pour ce faire les commandes `cURL` que vous avez utilisées pour vos tests. Vous veillerez à exécuter la commande `mvn clean` avant de construire votre archive.

3.5 API REST avec OpenAPI

Nicolas Le Sommer, Matthieu Le Lain et Serge Dubois

Dans ce travail, vous allez commenter le micro-service REST que vous avez développé lors du précédent travail pratique pour que ce micro-service réponde à la spécification OpenAPI.

1. Création d'une API REST conforme à la spécification OpenAPI

1. Ajoutez dans le fichier pom.xml de votre projet une dépendance vers l'artefact `springdoc-openapi-ui`.

```
<dependency>
  <groupId>org.springdoc</groupId>
  <artifactId>springdoc-openapi-starter-webmvc-ui</artifactId>
  <version>2.0.2</version>
</dependency>
```

2. Ajoutez un fichier `application.properties` contenant les propriétés ci-dessous dans le répertoire `src/main/resources/`. Ainsi, vous pourrez préciser quel est le paquetage à explorer, et définir un chemin spécifique pour `swagger-ui` et `api-docs`.

```
springdoc.api-docs.path=/openapi/api-docs
springdoc.swagger-ui.path=/openapi/index.html
springdoc.swagger-ui.operationsSorter=method
springdoc.packagesToScan=fr.ubs.sporttrack.controller
```

3. Produisez un package pour votre application et consultez la documentation en vous connectant aux URLs suivantes:

- http://localhost:8080/openapi/index.html
- http://localhost:8080/openapi/api-docs

4. Complétez la documentation de votre modèle de données (i.e. les classes `Activity` et `Data`) avec les annotations prévues dans la JSR 380 pour la validation d'API. Pour ce faire, ajoutez dans le fichier `pom.xml` de votre projet `fr.ubs.sporttrack.model` une dépendance vis-à-vis de la version `3.0.2` de l'artefact `jakarta.validation-api` du groupe `jakarta.validation`. Ces annotations vous permettent de définir pour un attribut par exemple une obligation de présence (`@NotNull`), une valeur minimale (`@Min`), une valeur maximale (`@Max`) ou un intervalle de valeurs (`@Size=(min=-90, max=90)`). Après avoir annoté votre modèle de données, compilez votre projet et installez-le dans votre dépôt Maven local.

5. Complétez la documentation de votre API (i.e. de votre contrôleur `ActivityController`) avec les annotations `@Operation`, `@ApiResponse`, `@ApiResponses`, `@Content` et `@Schema`, afin de préciser ce que fait la méthode (`@Operation`) et les données retournées (`@ApiResonse` et `@ApiResponses`). Vous ajouterez une dépendance dans votre projet vis-à-vis de la version `2.2.7` de l'artefact `swagger-annotations` du groupe `io.swagger.core.v3`. Vous vous référerez à la documentation de Swagger pour ce faire. Vous compilerez votre projet et vous le testerez en exécutant la commande `mvn clean spring-boot:run`.

6. Afin de valider les données ajoutées dans votre fichier JSON de données via une requête HTTP POST, utilisez les classes `Validator`, `Validation`, `ValidatorFactory` et `ConstraintViolation` d'Hibernate. Pour ce faire, ajoutez le fichier `pom.xml` de votre application springboot une dépendance vis-à-vis de la version `8.0.0.Final` de l'artefact `hibernate-validator`. Le `groupId` de cet artefact est `org.hibernate.validator`. Référez vous à la documentation d'`hibernate-validator` pour la validation des contraintes.

2. Travail à rendre

Vous devrez déposer sur Moodle une archive tar.gz contenant le code source de votre application. Cette archive devra contenir également un fichier README.md précisant ce que vous avez réussi à développer, et les tests que vous avez effectués. Vous donnerez pour ce faire les commandes `cURL` que vous avez utilisées pour vos tests. Vous veillerez à exécuter la commande `mvn clean` avant de construire votre archive.

3.6 Micro-services

Nicolas Le Sommer, Matthieu Le Lain et Serge Dubois

Dans ce travail, vous allez développer, en vous inspirant des travaux précédents, un micro-service permettant de gérer des utilisateurs. Ce micro-service devra permettre de lister les utilisateurs, d'ajouter un nouvel utilisateur et d'obtenir des informations sur un utilisateur particulier. Ce micro-service devra également permettre de vérifier l'identifiant et le mot de passe d'un utilisateur. Dans ce travail, vous allez également développer une application avec le cadriciel `Vue.js` exploitant les micro-services "utilisateurs" et "activités".

1. Création d'un micro-service de gestion des utilisateurs

1. Créez le modèle de données et annotez-le.
2. Créez le contrôleur permettant de gérer les utilisateurs (ajout, liste, etc.), et annotez-le.
3. Testez le bon fonctionnement de votre service.

2. Création d'une application

1. Avec le cadriciel `Vue.js`, créez une vue pour afficher la liste des activités. Cette vue devra exploiter le micro-service que vous avez développé.
2. Créez une vue pour afficher la liste des utilisateurs. Cette vue utilisera le micro-service que vous avez développé.
3. Créez une vue permettant d'inscrire un nouvel utilisateur. Cette vue devra utiliser le micro-service "utilisateur".
4. Créez une vue permettant à un utilisateur de se connecter à votre application. Cette vue devra utiliser le micro-service "utilisateur".
5. Modifiez la vue listant les activités afin de ne les afficher que lorsque l'utilisateur est connecté. Pour des raisons de simplicité, on ne distinguera pas l'appartenance des activités.

3. Travail à rendre

Vous devrez déposer sur Moodle une archive tar.gz contenant le code source de votre micro-service et de votre application. Cette archive devra contenir également un fichier README.md précisant ce que vous avez réussi à développer, et les tests que vous avez effectués. Vous veillerez à exécuter la commande `mvn clean` avant de construire votre archive.

3.7 Sécurité et API REST

Nicolas Le Sommer, Matthieu Le Lain et Serge Dubois

Dans ce travail, vous allez sécuriser l'accès à votre service de gestion des activités sportives.

1. Sécurisation par couple identifiant/mot de passe

1. Pour sécuriser l'accès à votre service de gestion d'activités sportives, vous devez ajouter dans le fichier `pom.xml` de votre projet une dépendance vis-à-vis de la version 3.0.1 de l'artefact `spring-boot-starter-security`, dont le `groupId` est `org.springframework.boot`.
2. Définissez dans le `src/main/resources/application.properties` les propriétés `security.user.username` et `security.user.password`. Ces propriétés prendront toutes les deux la valeur "sporttrack". Elles représentent respectivement l'identifiant et le mot de passe à utiliser pour accéder à votre micro-service. Ces propriétés seront utilisées par votre application.
3. Ajoutez une classe `SecurityConfiguration` dans le paquet Java `fr.ubs.sporttrack.security` de votre application. Cette classe devra être annotée avec l'annotation `@Configuration` afin que les propriétés `security.user.username` et `security.user.password` contenues dans le fichier `application.properties` puissent être injectées dans les attributs `username` et `password` que vous définirez dans cette classe `SecurityConfiguration`. Pour permettre cette injection/auto-configuration, annotez les attributs `username` et `password` avec les annotations `@Value("${security.user.username}")` et `@Value("${security.user.password}")`.
4. Afin gérer les utilisateurs pouvant se connecter à votre micro-service, définissez une méthode `userDetailsService()` retournant un objet de type `org.springframework.security.provisioning.InMemoryUserDetailsManager`. Vous annoterez cette méthode avec l'annotation `@Bean` pour qu'elle puisse être considérée comme un élément de contexte d'exécution du cadriciel Spring. À l'aide de la classe `org.springframework.security.core.userdetails.User`, créez un objet de type `org.springframework.security.core.userdetails.UserDetails`. Vous pourrez définir l'identifiant, le mot de passe et le rôle de l'utilisateur autorisé à accéder à votre micro-service avec les méthodes `username()`, `password()` et `roles()`.
5. Afin de filtrer les méthodes HTTP adressées à votre micro-service, définissez une méthode `filterChain()` dont la signature est :

```
public SecurityFilterChain filterChain(HttpSecurity http) throws Exception
```

   Annotez cette méthode avec l'annotation `@Bean`. En utilisant l'objet `http` (de type `HttpSecurity`) passé en paramètre de cette méthode, faites en sorte que la documentation de votre micro-service puisse être consultée sans restriction (`"/openapi"`), mais que l'utilisation du micro-service est elle conditionnée à une authentification.
6. Testez le bon fonctionnement de votre politique de sécurité en connectant à l'URL suivante: http://localhost:8080/openapi/index.html. Essayez d'exécuter depuis l'interface Web Swagger une requête HTTP `GET` `/activities/`. Si tout se passe bien, un identifiant et un mot de passe vous sont demandés. Vous pouvez saisir comme identifiant ceux définis dans le fichier `application.properties`.
7. Le mot de passe d'accès à votre micro-service est stocké en clair dans le fichier `application.properties`. Nous voulons chiffrer ce mot de passe. Pour ce faire, définissez dans la classe `SecurityConfiguration` une méthode `passwordEncoder()` qui retourne un objet de type `org.springframework.security.crypto.bcrypt.BCryptPasswordEncoder`. Cette méthode devra être annotée avec l'annotation `@Bean`. Trouvez un moyen pour chiffrer, à l'aide de cette classe le mot de passe `sporttrack`, et stocker ce mot de passe chiffré dans le fichier `application.properties`. Testez le bon fonctionnement en recompilant et redémarrant votre micro-service, et en vous connectant à celui-ci.

2. Travail à rendre

Vous devrez déposer sur Moodle une archive tar.gz contenant le code source de votre application. Cette archive devra contenir également un fichier README.md précisant ce que vous avez réussi à développer, et les tests que vous avez effectués. Vous donnerez pour ce faire les commandes `cURL` que vous avez utilisées pour vos tests. Vous veillerez à exécuter la commande `mvn clean` avant de construire votre archive.