

## Sisteme Distribuite - Laborator 3

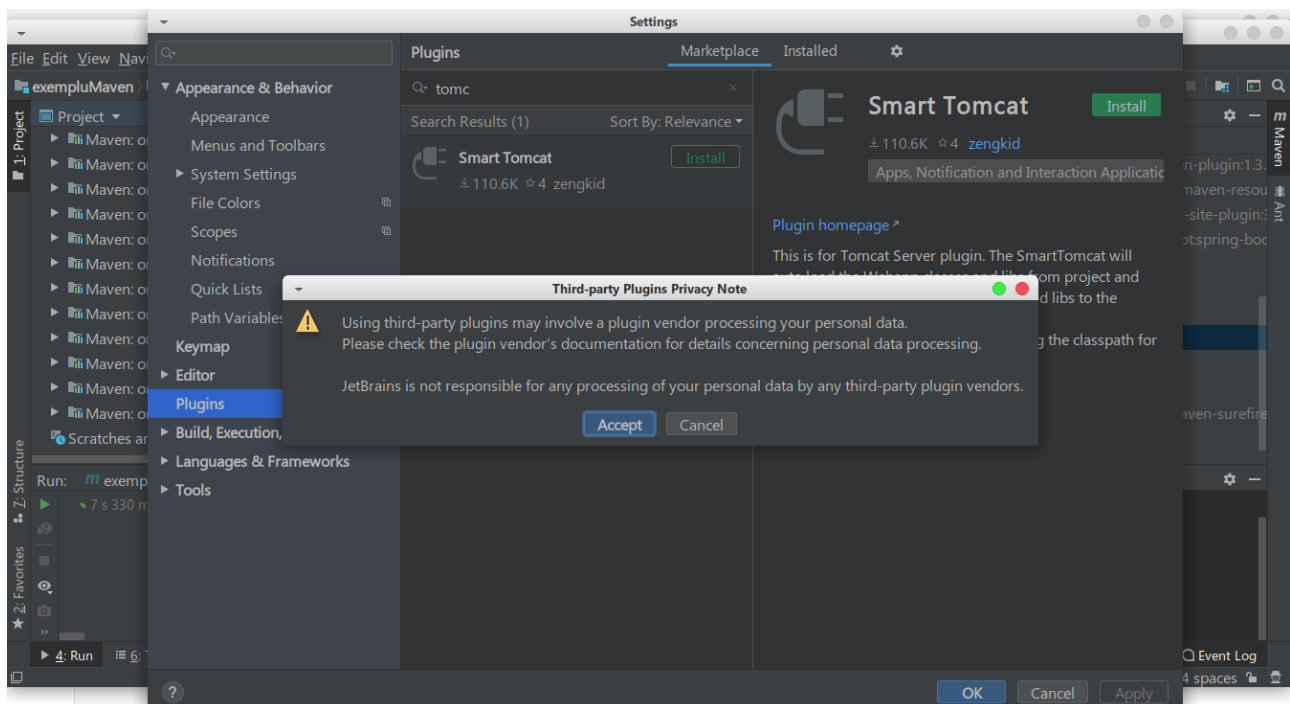
### Prima aplicație utilizând Spring Boot

Obiectivele lucrării de laborator sunt următoarele:

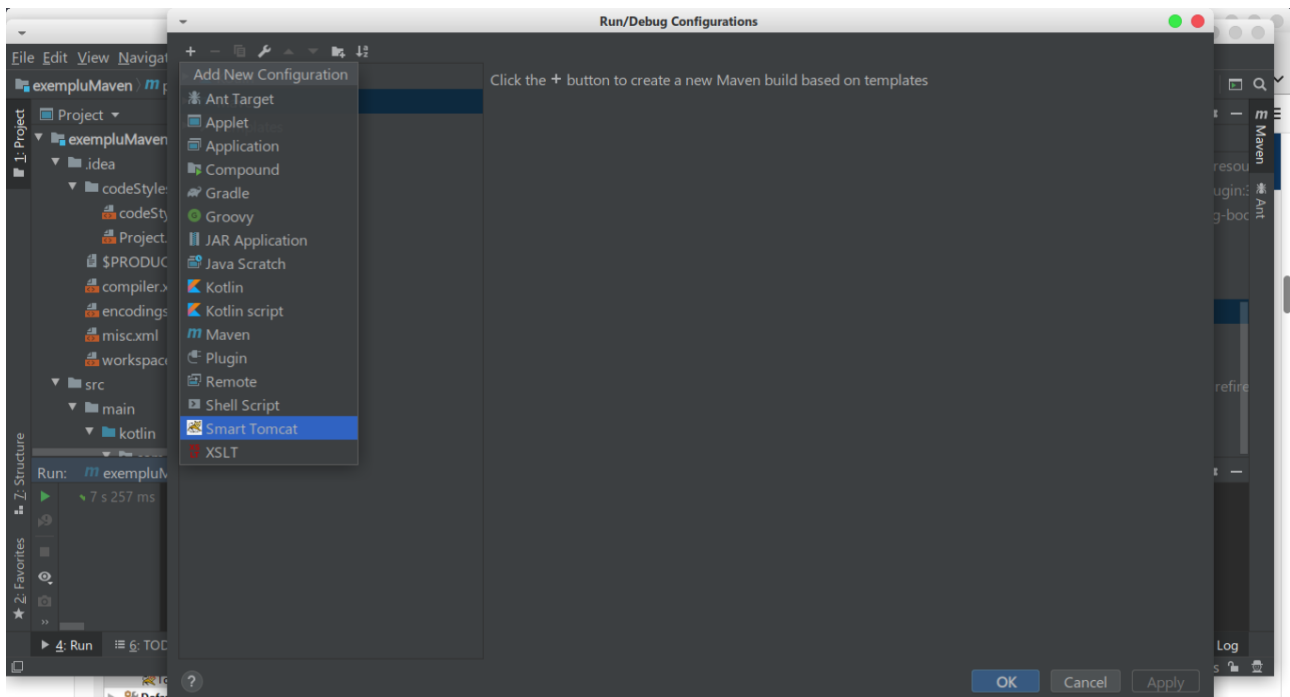
- crearea unui proiect **Spring Boot**;
  - folosind **Maven**;
  - folosind **Gradle**;
- primii pași în utilizarea expunerii serviciilor folosind *plugin*-ul **Spring Boot**;
- crearea unei aplicații **Spring** simple.

**ALT+ENTER este prietenul vostru!**

Pentru a putea executa aplicațiile specifice, trebuie instalat un server de **Tomcat** în mediul virtual al IntelliJ. **File** → **Settings** → **Plugins** și se caută și instalează *plugin*-ul „**Smart Tomcat**” (vezi figura următoare).



În continuare, se adaugă o configurație de execuție pentru a porni Smart Tomcat.



Acesta este *plugin*-ul pentru serverul de **Tomcat** care va autoîncărca clasele aplicației Web precum și bibliotecile din proiect și modulele. De asemenea, va autoconfigura și căile către clase necesare serverului Tomcat. Suportă Tomcat începând cu versiunea 6.

## 1. Crearea unui proiect Spring Boot

Spring Boot ușurează crearea de aplicații bazate pe *framework*-ul Spring, prin punerea la dispoziția utilizatorului a unui proiect Spring preconfigurat.

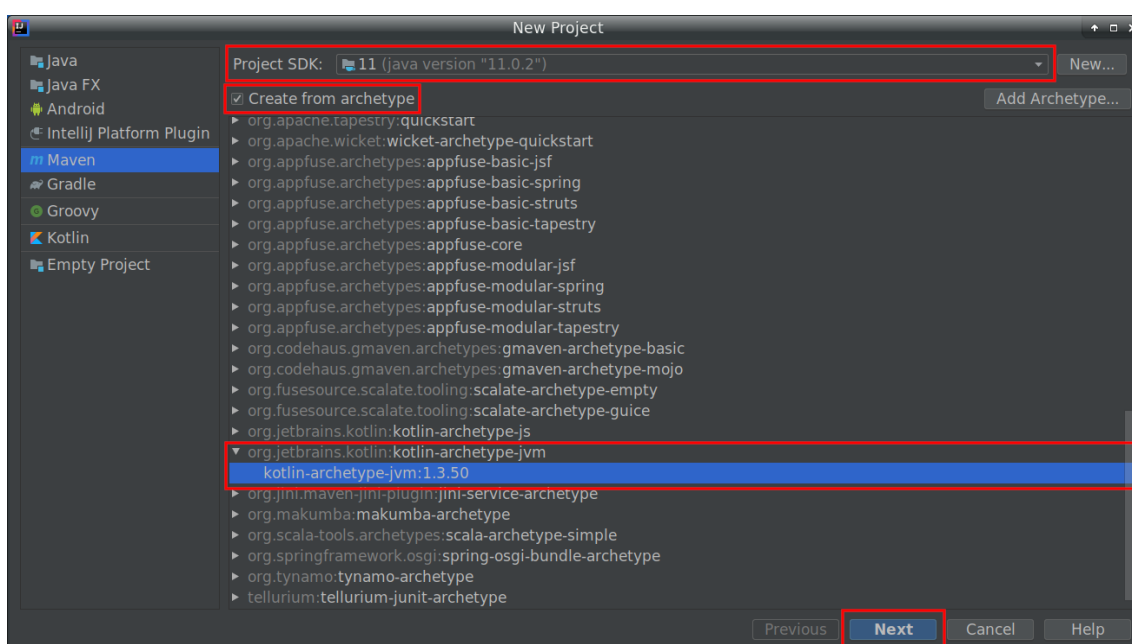
Deoarece mediul de dezvoltare IntelliJ IDEA versiunea „Community Edition” (disponibilă în laborator) nu oferă opțiunea de a crea direct un proiect de tip „Spring Boot Application”, va trebui să se urmeze pașii prezentați în cele ce urmează, în funcție de utilitarul folosit pentru management-ul proiectelor (Maven / Gradle).

### 1.1. Proiect Spring Boot folosind *Maven*

1. Deschideți aplicația **IntelliJ IDEA Community**
2. Selectați „**Create New Project**” din meniul principal



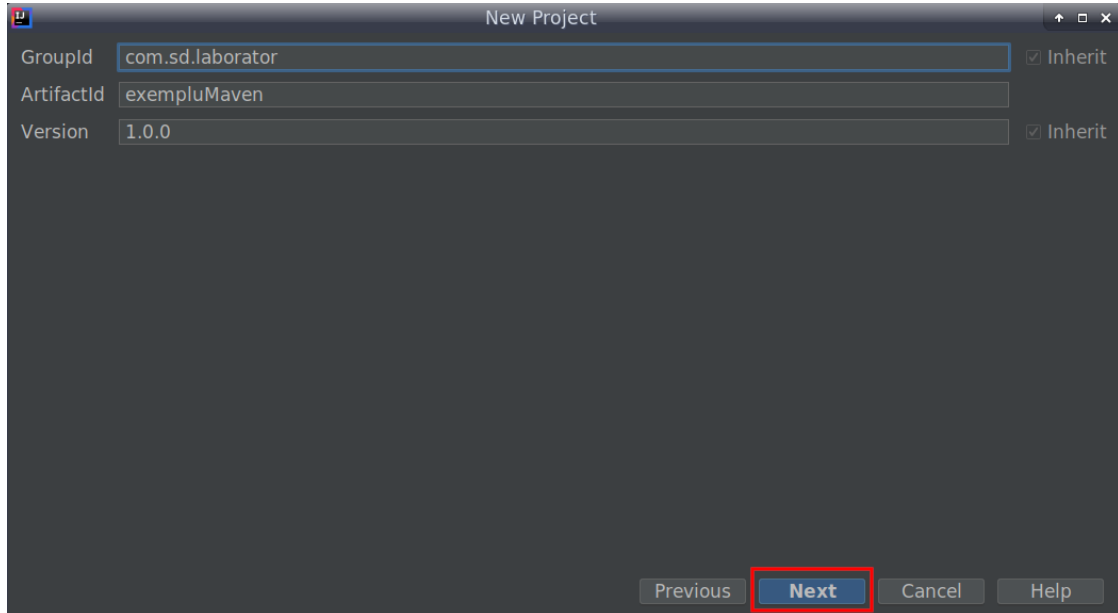
3. Alegeți ca tip de proiect „**Maven**”, selectați **versiunea 11** de **Java SDK**, apoi bifați „**Create from archetype**”. Din lista de mai jos, selectați „**org.jetbrains.kotlin:kotlin-archetype-jvm**” → „**kotlin-archetype-jvm:<versiune\_Kotlin>**”, apoi apăsați pe pe „**Next**”.



4. Completați metadatele proiectului, conform cu specificațiile Apache Maven, disponibile la următorul URL:

<https://maven.apache.org/guides/mini/guide-naming-conventions.html>

- a) groupId: **com.sd.laborator**
- b) artifactId: **exempluMaven**
- c) version: **1.0.0**

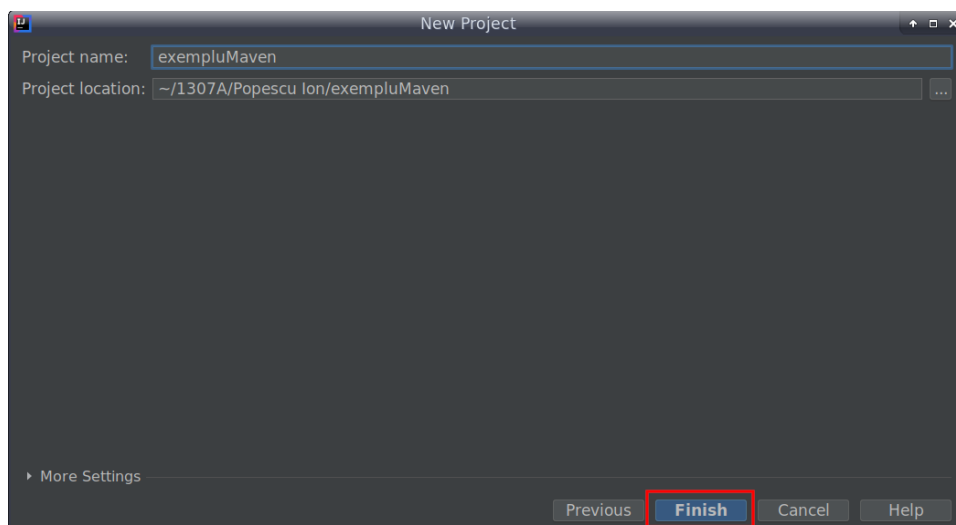


5. Se va apăsa pe „Next”, iar în secțiunea următoare nu modificați nici o setare. Se va apăsa din nou pe „Next”.
6. Completați numele proiectului și selectați locația sa pe disc, apoi apăsați „Finish”.

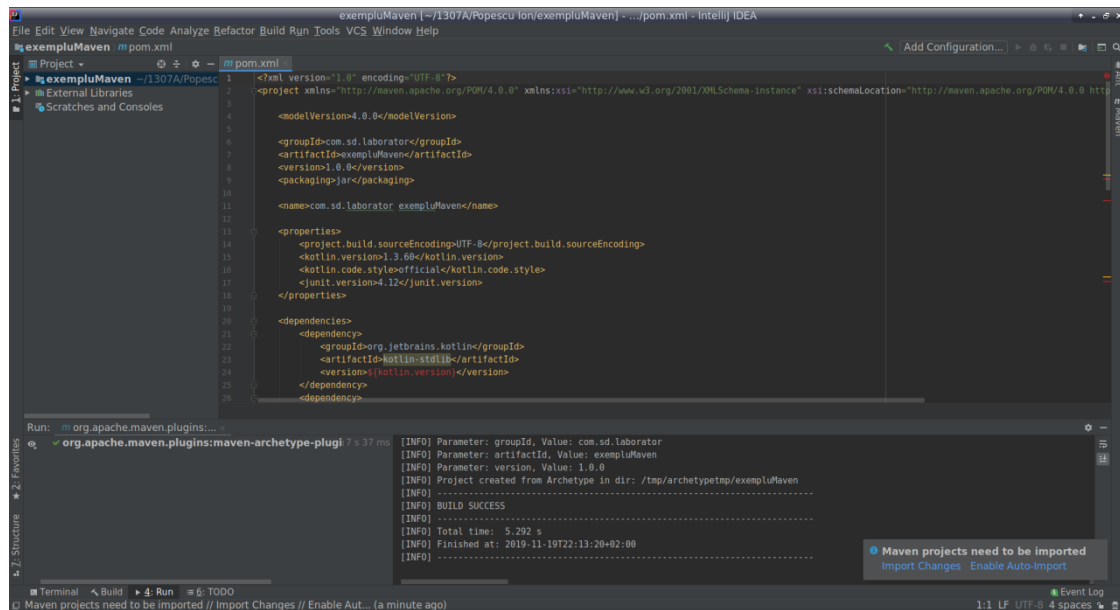
În cazul în care nu există deja, se creează un folder în locația **/home/student** cu numele grupei din care faceți parte. În folder-ul grupei, se creează un alt folder cu numele dvs., acela fiind folder-ul de bază în care se va lucra la laborator.

Exemplu: **/home/student/1307A/Popescu Ion**

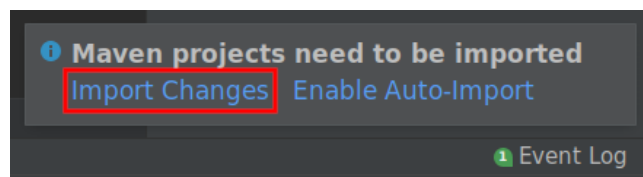
**Atenție!** Fișierele proiect vor fi plasate **direct** în ultimul folder din calea specificată în „Project location”. IntelliJ nu creează automat niciun subfolder pentru utilizator. Așadar, ar fi indicat ca locația proiectului să indice către un folder gol!



7. Se va deschide o fereastră IntelliJ, cu noul proiect Maven creat, afișând automat conținutul fișierului „**pom.xml**”. Acesta reprezintă un **Project Object Model**, un fișier XML care conține toate informațiile necesare compilării proiectului respectiv. Se poate observa că, pentru moment, acesta conține metadatele introduse la pasul 4, în *tag*-urile XML corespunzătoare.



8. În momentul în care IntelliJ afișează următorul mesaj, selectați „**Import Changes**”:

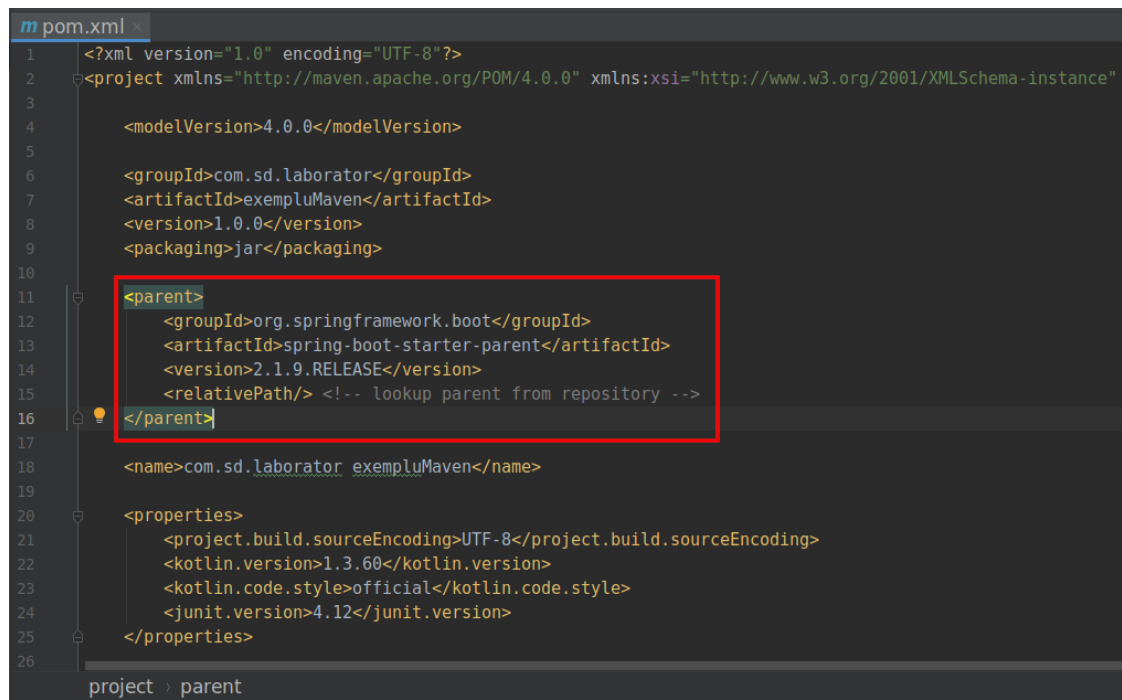


Operația va conduce la sincronizarea dependențelor specificate în fișierul de configurare **pom.xml**, în momentul în care este editat de către utilizator. Ca efect, modificarea făcută în pasul 7 de mai sus va determina ca IntelliJ să descarce automat toate dependențele specificate în interiorul *tag*-ului **<dependencies>**.

9. În fișierul POM (**pom.xml**), adăugați următoarele elemente, ca și subordonați ai *tag*-ului **<project>**:

```
<parent>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-parent</artifactId>
  <version>2.1.9.RELEASE</version>
  <relativePath/> <!-- lookup parent from repository -->
</parent>
```

*Tag*-urile adăugate mai sus determină ca proiectul să moștenească dependențele și setările unei aplicații **Spring Boot** din *repository*-urile Maven.



```

1  <?xml version="1.0" encoding="UTF-8"?>
2  <project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
3
4      <modelVersion>4.0.0</modelVersion>
5
6      <groupId>com.sd.laborator</groupId>
7      <artifactId>exempluMaven</artifactId>
8      <version>1.0.0</version>
9      <packaging>jar</packaging>
10
11     <parent>
12         <groupId>org.springframework.boot</groupId>
13         <artifactId>spring-boot-starter-parent</artifactId>
14         <version>2.1.9.RELEASE</version>
15         <relativePath/> <!-- lookup parent from repository -->
16     </parent>
17
18     <name>com.sd.laborator exempluMaven</name>
19
20     <properties>
21         <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
22         <kotlin.version>1.3.60</kotlin.version>
23         <kotlin.code.style>official</kotlin.code.style>
24         <junit.version>4.12</junit.version>
25     </properties>
26

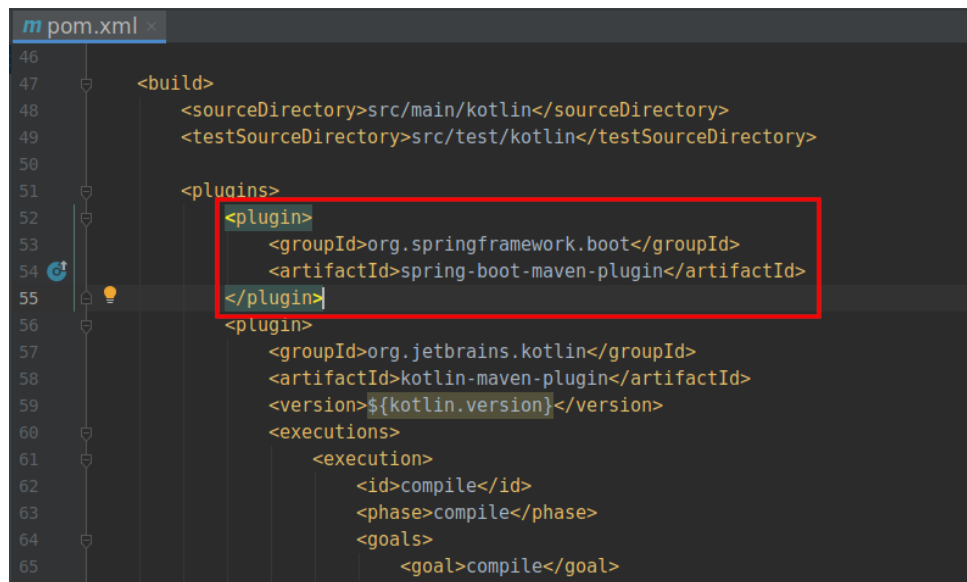
```

10. Ca subordonat al *tag*-ului **<plugins>**, adăugați **Spring Boot Plugin**:

```

<plugin>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-maven-plugin</artifactId>
</plugin>

```



```

46
47     <build>
48         <sourceDirectory>src/main/kotlin</sourceDirectory>
49         <testSourceDirectory>src/test/kotlin</testSourceDirectory>
50
51         <plugins>
52             <plugin>
53                 <groupId>org.springframework.boot</groupId>
54                 <artifactId>spring-boot-maven-plugin</artifactId>
55             </plugin>
56             <plugin>
57                 <groupId>org.jetbrains.kotlin</groupId>
58                 <artifactId>kotlin-maven-plugin</artifactId>
59                 <version>${kotlin.version}</version>
60                 <executions>
61                     <execution>
62                         <id>compile</id>
63                         <phase>compile</phase>
64                         <goals>
65                             <goal>compile</goal>

```

11. Adăugați următoarele dependențe suplimentare (elemente subordonate al *tag*-ului **<dependencies>**):

```

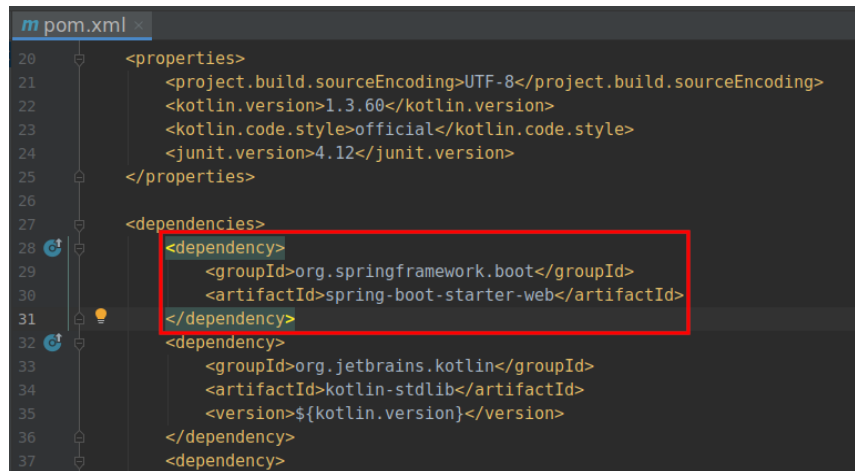
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-web</artifactId>
</dependency>
<dependency>
  <groupId>org.jetbrains.kotlin</groupId>

```

```

<artifactId>kotlin-reflect</artifactId>
<version>${kotlin.version}</version>
</dependency>

```



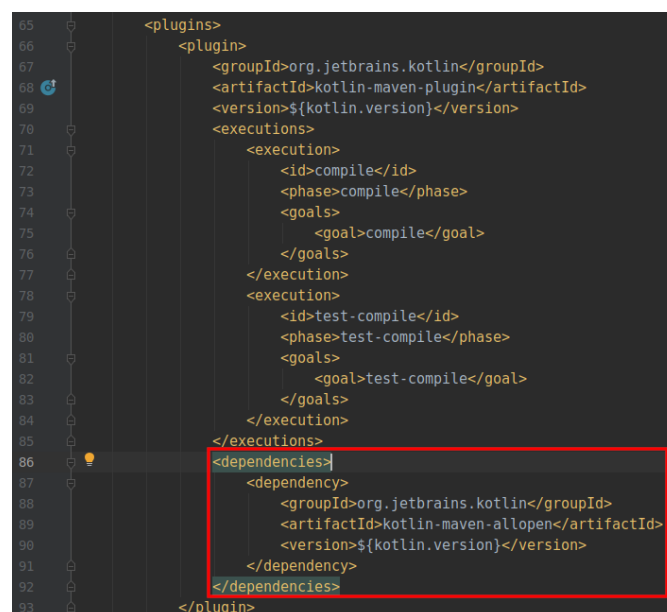
Clasele în Kotlin sunt, în mod implicit, marcate ca și **final**, deci nu se pot moșteni decât dacă dezvoltatorul le marchează explicit ca **open** (de exemplu, **open** class MyClass ...). **Spring** necesită ca acele clase ce vor primi anumite tipuri de adnotări (cum ar fi **@Component** sau **@Service**) să fie moștenibile, adică marcate cu **open**. Acest lucru este făcut automat de *plugin*-ul **kotlin-maven-allopen**, așadar îl veți adăuga ca dependență la compilare, astfel:

Adăugați următorul element de configurare în interiorul *tag*-ului **<plugin>**, corespunzător *plugin*-ului **kotlin-maven-plugin** (consultați figura de mai jos pentru locația exactă):

```

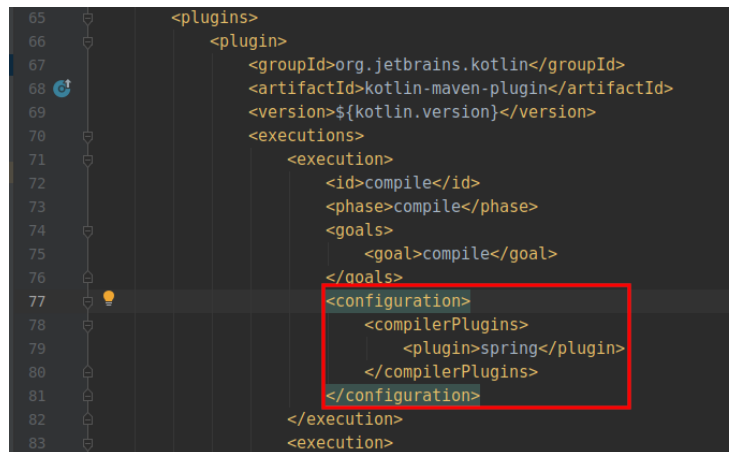
<dependencies>
  <dependency>
    <groupId>org.jetbrains.kotlin</groupId>
    <artifactId>kotlin-maven-allopen</artifactId>
    <version>${kotlin.version}</version>
  </dependency>
</dependencies>

```



Apoi, adăugați *plugin*-ul **spring** ca și dependență la faza de compilare, cu următorul element copil al tag-ului **<execution>**, pus în locația indicată în figura ce urmează.

```
<configuration>
  <compilerPlugins>
    <plugin>spring</plugin>
  </compilerPlugins>
</configuration>
```



Pentru productivitate sporită, adăugați ca dependență și **Spring Boot Developer Tools**:  
<https://mvnrepository.com/artifact/org.springframework.boot/spring-boot-devtools>

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-devtools</artifactId>
  <version>${project.parent.version}</version>
</dependency>
```

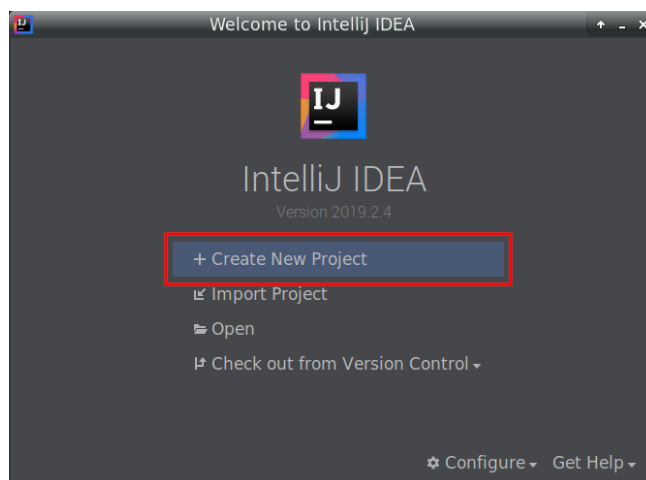
Adăugând această dependență, *plugin*-ul **Spring Boot** va reîncărca automat artefactele rezultate din compilarea surselor, atunci când acestea sunt modificate de utilizator. Deci, **odată** pornite aplicația și server-ul Tomcat (folosind *goal*-ul **spring-boot:run**), dacă modificați sursele și vreți să vedeți rezultatul modificărilor, doar le compilați cu *lifecycle*-ul Maven **compile** și atât. Aplicația vi se reîncarcă automat (dacă nu sunt erori!).

În acest moment, s-a obținut un proiect Maven creat folosind un șablon (*archetype*) pentru Kotlin, în care ați actualizat fișierul POM pentru a include ca și dependențe „**Spring Boot**”, „**Spring Boot Developer Tools**” și „**Spring Boot Starter Web**”. Cea din urmă este necesară pentru construirea de aplicații web, folosind Spring MVC.

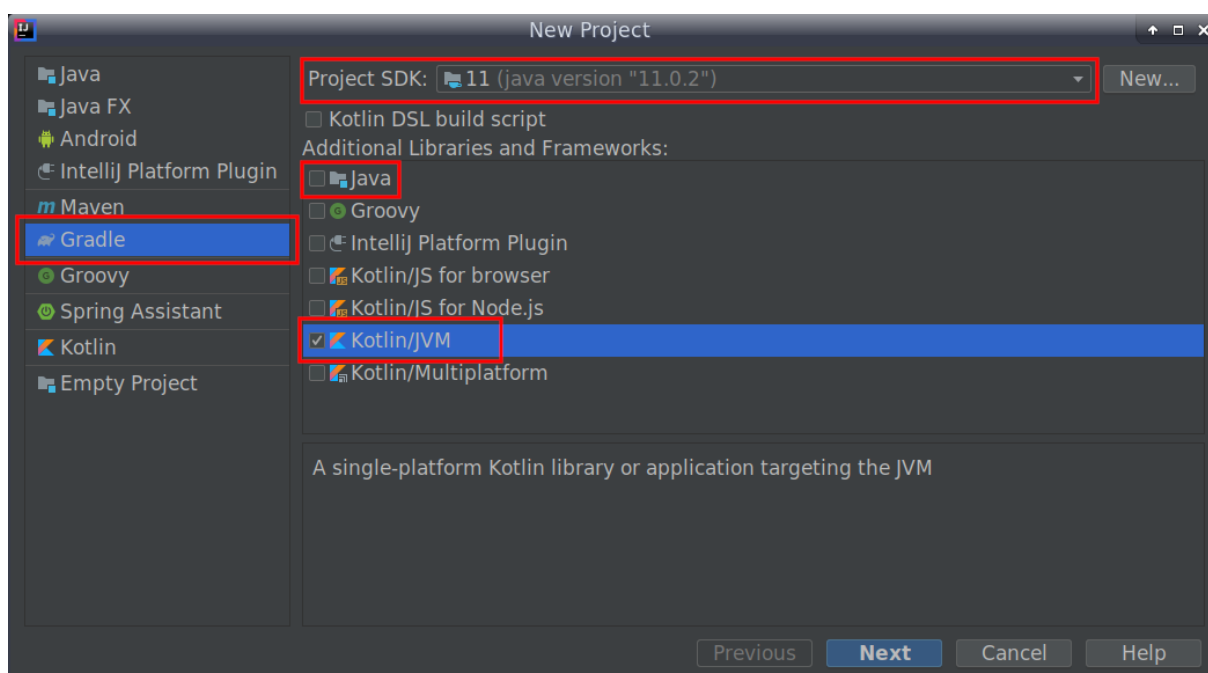


## 1.2. Proiect Spring Boot folosind **Gradle**

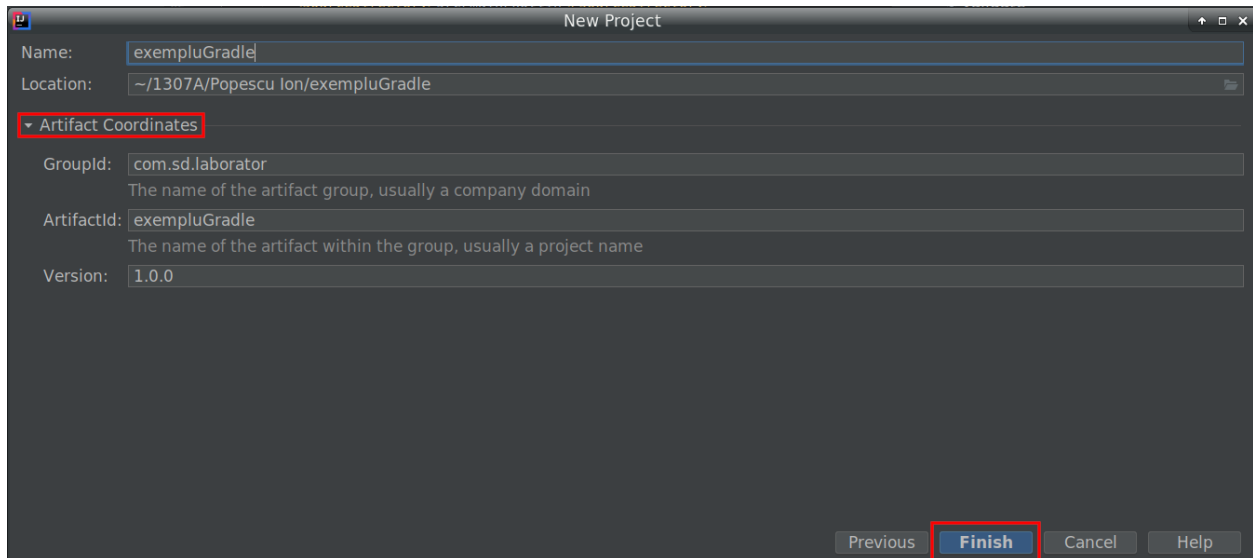
1. Deschideți aplicația **IntelliJ IDEA Community**
2. Selectați „Create New Project” din meniul principal



3. Se va alege ca tip de proiect „**Gradle**”, selectați **versiunea 11** de **Java SDK**, apoi, în secțiunea „Additional Libraries and Frameworks”, **debifați „Java”** și bifați „**Kotlin/JVM**”. Se va apăsa pe pe „Next”.



4. Completați numele proiectului și locația sa pe disc, respectiv metadatele proiectului, expandând secțiunea „**Artifact Coordinates**”:
  - a) groupId: **com.sd.laborator**
  - b) artifactId: **exempluGradle**
  - c) version: **1.0.0**



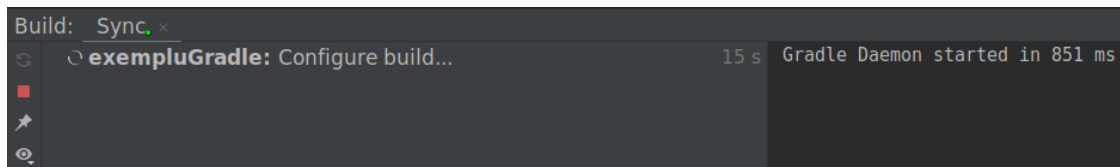
În cazul în care nu există deja, se creează un folder în locația **/home/student** cu numele grupei din care faceți parte. În folder-ul grupei, se creează un alt folder cu numele dvs., acela fiind folder-ul de bază în care se va lucra la laborator.

Exemplu: **/home/student/1307A/Popescu Ion**

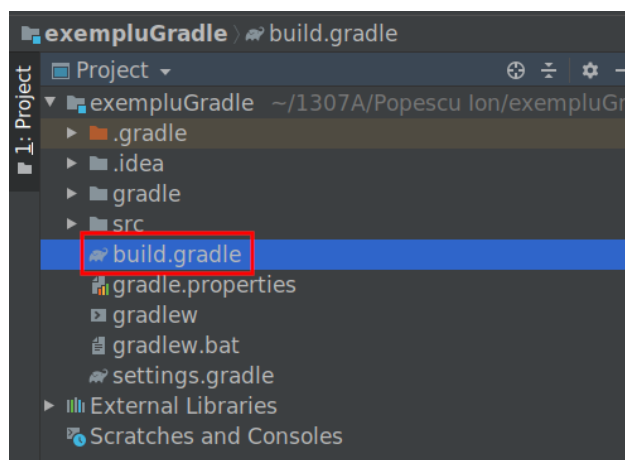
**Atenție!** Fișierele proiect vor fi plasate **direct** în ultimul folder din calea specificată în „Project location”. IntelliJ nu creează automat niciun subfolder pentru utilizator. Așadar, ar fi indicat ca locația proiectului să indice către un folder gol!

Apăsați „Finish”.

5. Așteptați ca gestionarul de proiect Gradle să configureze automat proiectul nou creat.



6. După ce proiectul a fost configurat și în partea stângă a ferestrei apare ierarhia de fișiere și foldere, deschideți fișierul „**build.gradle**”.

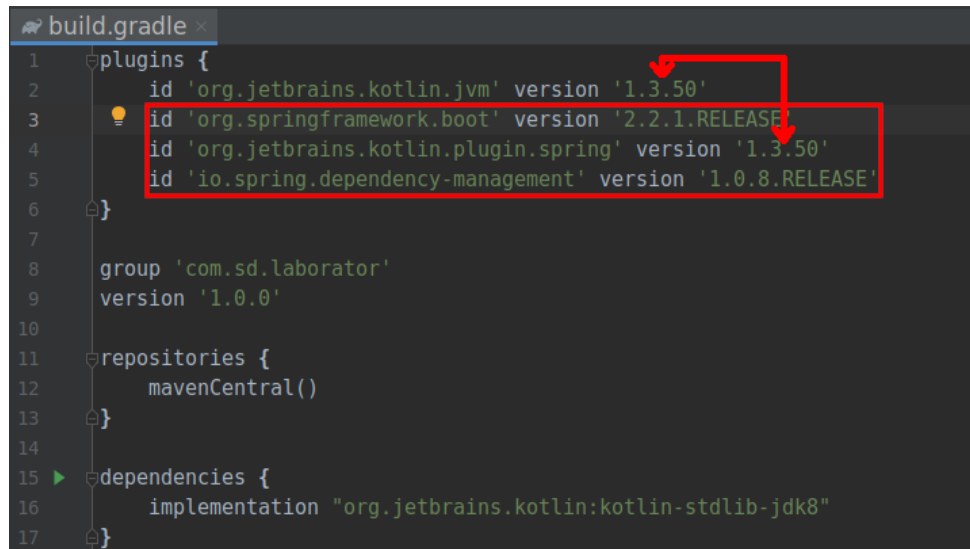


Adăugați următoarea listă de *plugin*-uri (ca subordonați în secțiunea **plugins**):

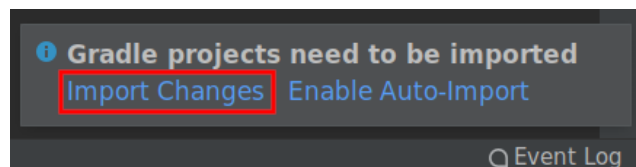
```
id 'org.springframework.boot' version '2.2.1.RELEASE'
id 'org.jetbrains.kotlin.plugin.spring' version '<VERSIUNE_KOTLIN>'
id 'io.spring.dependency-management' version '1.0.8.RELEASE'
```

Înlocuiți **<VERSIUNE\_KOTLIN>** cu versiunea de Kotlin existentă în repositories la momentul creării proiectului. Aceasta apare pe linia de mai sus, adăugată automat de IntelliJ:

```
id 'org.jetbrains.kotlin.jvm' version '1.3.50'
```



După modificarea fișierului **build.gradle**, IntelliJ va afișa un mesaj în partea din dreapta-jos a ferestrei. Selectați „**Import Changes**”.



Prin aceasta se va realiza sincronizarea automată a dependențelor specificate în fișierul de configurare **build.gradle**, în momentul în care este editat de către utilizator. Ca efect, modificarea făcută în pasul de mai sus va determina ca IntelliJ să descarce automat toate *plugin*-urile specificate în secțiunea **plugins**.

7. Adăugați **Kotlin Reflect** și **Spring Boot Web Application** ca și dependențe (elemente subordonate ale secțiunii **dependencies**):

```
implementation "org.jetbrains.kotlin:kotlin-reflect"
implementation "org.springframework.boot:spring-boot-starter-web"
```

```

8   group 'com.sd.laborator'
9   version '1.0.0'
10
11  repositories {
12      mavenCentral()
13  }
14
15  dependencies {
16      implementation "org.jetbrains.kotlin:kotlin-stdlib-jdk8"
17      implementation "org.jetbrains.kotlin:kotlin-reflect"
18      implementation "org.springframework.boot:spring-boot-starter-web"
19  }
20
21  compileKotlin {
22      kotlinOptions.jvmTarget = "1.8"
23  }

```

Pentru productivitate sporită, adăugați ca dependență și **Spring Boot Developer Tools**:  
<https://docs.spring.io/spring-boot/docs/1.5.16.RELEASE/reference/html/using-boot-devtools.html>

```
compileOnly "org.springframework.boot:spring-boot-devtools"
```

```

dependencies {
    implementation "org.jetbrains.kotlin:kotlin-stdlib-jdk8"
    implementation "org.jetbrains.kotlin:kotlin-reflect"
    implementation "org.springframework.boot:spring-boot-starter-web"
    compileOnly "org.springframework.boot:spring-boot-devtools"
}

```

Apoi, adăugați o secțiune nouă în fișierul **build.gradle**:

```

configurations {
    compileOnly
    runtimeClasspath {
        extendsFrom compileOnly
    }
}

```

```

15 configurations {
16     compileOnly
17     runtimeClasspath {
18         extendsFrom compileOnly
19     }
20 }
21
22 dependencies {
23     implementation "org.jetbrains.kotlin:kotlin-stdlib-jdk8"
24     implementation "org.jetbrains.kotlin:kotlin-reflect"
25     implementation "org.springframework.boot:spring-boot-starter-web"
26
27     compileOnly "org.springframework.boot:spring-boot-devtools"
28 }
29

```

Adăugând această dependență, *plugin-ul Spring Boot* va reîncărca automat artefactele rezultate din compilarea surselor, atunci când acestea sunt modificate de utilizator. Deci, **odată pornite aplicația și server-ul Tomcat (folosind task-ul application/bootRun), dacă modificați**

sursele și vreți să vedeți rezultatul modificărilor, doar le compilați cu *task*-ul Gradle **build/build** și atât. Aplicația vi se reîncarcă automat (dacă nu sunt erori!).

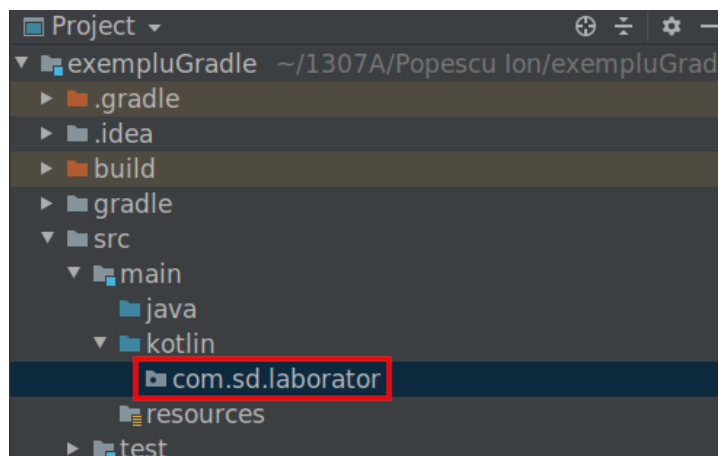
În acest moment, aveți un proiect Gradle, la care s-a adăugat librăria adițională Kotlin/JVM, proiect în care ați actualizat fișierul **build.gradle** pentru a include ca și dependențe „**Spring Boot**” și „**Spring Boot Starter Web**”. Cea din urmă este necesară pentru construirea de aplicații web, folosind Spring MVC.

## 2. Crearea unei aplicații simple

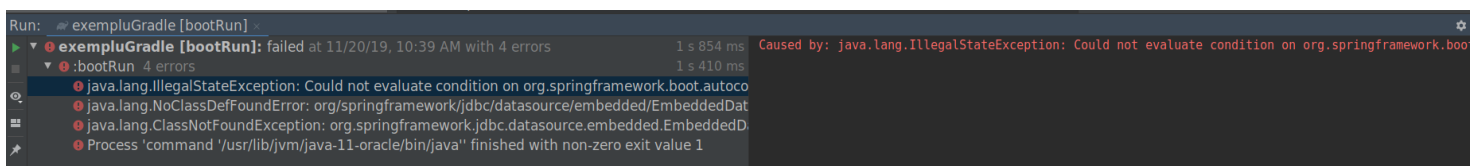
Pentru început se va crea o aplicație Spring MVC de test, care va afișa un simplu mesaj: „Hello World”. Aceasta va cuprinde un serviciu expus în calea /helloworld, care este accesibilă local prin intermediul URL-ului: <http://localhost:8080/helloworld>.

### 2.1. Adăugarea fișierelor sursă

- **pentru proiecte de tip Gradle:** se creează manual un pachet în care vor fi plasate fișierele sursă. În panoul **Project** din partea stângă apăsați dreapta pe folder-ul **kotlin** din **<Nume\_Proiect>/src/main** → **New** → **Package** → introduceți „**com.sd.laborator**” și apăsați „OK”.



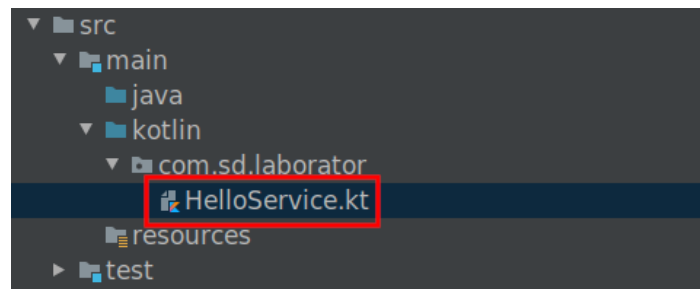
**Atenție!** Dacă nu se creează cel puțin un pachet pentru fișierele sursă și se plasează direct în folder-ul **kotlin** (adică în pachetul **default**), se va primi o excepție generată de Spring la pornirea aplicației:



- **pentru proiecte de tip Maven:** un pachet implicit va fi creat automat pe baza câmpului **GroupId** introdus ca metadată în pasul 4 de creare a unui proiect Maven. În acest caz, numele pachetului va fi **com.sd.laborator**.

În pachetul **com.sd.laborator**, se creează un fișier sursă Kotlin, denumit **HelloService.kt**.

Se va apăsa dreapta pe numele pachetului → **New** → **Kotlin File/Class** → Se va introduce „**HelloService**” ca și nume și se va lăsa selectat tipul de fișier „**File**”.



În continuare, se creează o clasă denumită **HelloService**, ce conține o funcție simplă fără parametri, denumită **getHello()**. Funcția va returna șirul de caractere „Hello World!”.

```
package com.sd.laborator

import org.springframework.stereotype.Service

@Service
class HelloService {
    fun getHello() = "Hello World!"
}
```

Pentru ca *framework*-ul Spring să recunoască această clasă ca și componentă (*bean*) și să o adauge în contextul de execuție pentru utilizarea la cerere, trebuie să adnotați clasa cu „**@Service**”. Pentru aceasta, este nevoie de import-ul interfeței **Service**, aflată în pachetul **org.springframework.stereotype**.

În mod asemănător, se creează o clasă *controller*, denumită **HelloController**, într-un fișier sursă Kotlin separat:

```
package com.sd.laborator

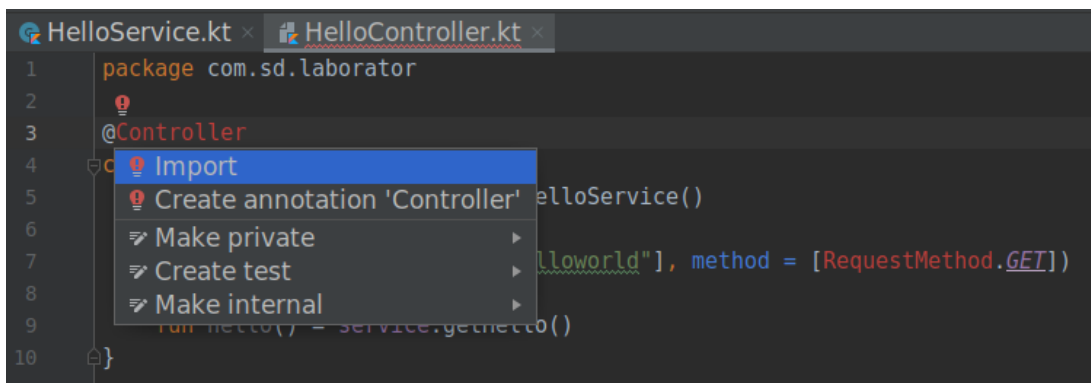
@Controller
class HelloController {
    val service: HelloService = HelloService()

    @RequestMapping(value = ["/helloworld"], method =
[RequestMethod.GET])
    @ResponseBody
    fun hello() = service.getHello()
}
```

Import-urile necesare nu au fost adăugate în mod intenționat, pentru a folosi facilitățile **IntelliSense** ale mediului de dezvoltare. IntelliJ avertizează și oferă potențiale soluții în cazul situațiilor de acest tip, spre exemplu atunci când se utilizează clase inexistente, care nu au fost importate, variabile neinițializate, specificatori de acces lipsă etc.



Se va apăsa pe primul identificator pe care IntelliJ nu îl găsește (în acest caz, nu este cunoscută interfața **Controller**), și apăsați **ALT+ENTER**.



Se va alege „**Import**” din lista de soluții propuse, iar IntelliJ va adăuga automat la începutul fișierului această linie:

```
import org.springframework.stereotype.Controller
```

Se va proceda la fel pentru toate celelalte erori raportate de mediul de dezvoltare, pentru a importa automat toate dependențele Spring de care aplicația are nevoie.

Fișierul **HelloController.kt** conține o clasă controller (adnotată cu „**@Controller**”), care expune o cale accesibilă prin HTTP: **/helloworld**. În controller, este instanțiat un obiect de tip **HelloService** prin care este accesat serviciu creat anterior (care conține metoda de interes **getHello()**). Mai concret: în momentul în care utilizatorul face o cerere HTTP de tip GET către calea **/helloworld** pe server-ul pe care se execută aplicația Spring (în acest caz, **localhost**), Spring va instanția un *bean* de tip **HelloService** în contextul de execuție, din care va apela metoda **getHello()**. Aceasta returnează un șir de caractere ce este trimis clientului apelant ca și corp de răspuns HTTP (**@ResponseBody**).

(**Maven**) Proiectul Maven va conține un fișier sursă Kotlin numit **Hello.kt**. **Ștergeți conținutul său** înainte de următorul pas.

Creați alt fișier sursă **Hello.kt** (dacă nu este deja creat), și adăugați următorul cod:

```
package com.sd.laborator

@SpringBootApplication
class Hello
```



```
fun main(args: Array<String>) {
    runApplication<Hello>(*args)
}
```

Adăugați dependențele lipsă semnalate de IntelliJ în aceeași manieră explicată mai sus.

Observați că funcția `main()` conține un apel de metodă `runApplication`, ce acceptă o clasă template, în acest caz `Hello`. `runApplication` este metoda de inițializare a unei aplicații Spring. Clasa template trimisă ca parametru trebuie adnotată cu `@SpringBootApplication` și reprezintă clasa folosită pentru configurarea aplicației (în cazul acesta, `Hello` nu conține niciun membru și nicio metodă).

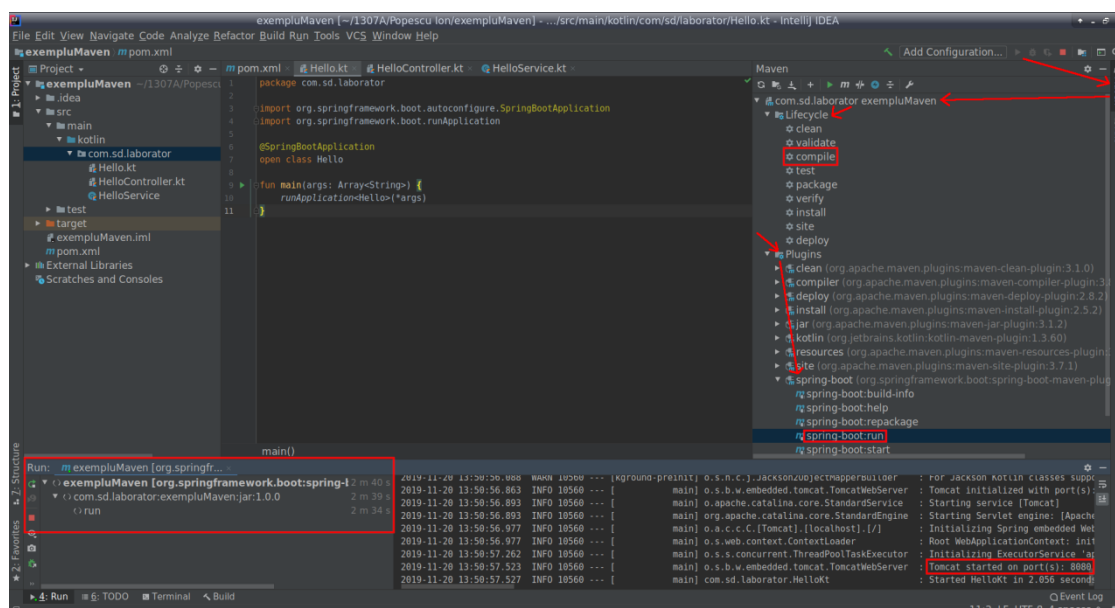
Adnotarea `@SpringBootApplication` declanșează scanarea de componente, configurarea automată și scanarea de proprietăți configurabile adiționale. Pentru detalii se poate consulta documentația, disponibilă aici:

<https://docs.spring.io/springboot/docs/current/api/org/springframework/boot/autoconfigure/SpringBootApplication.html>

## 2.2. Compilarea și execuția aplicației Spring

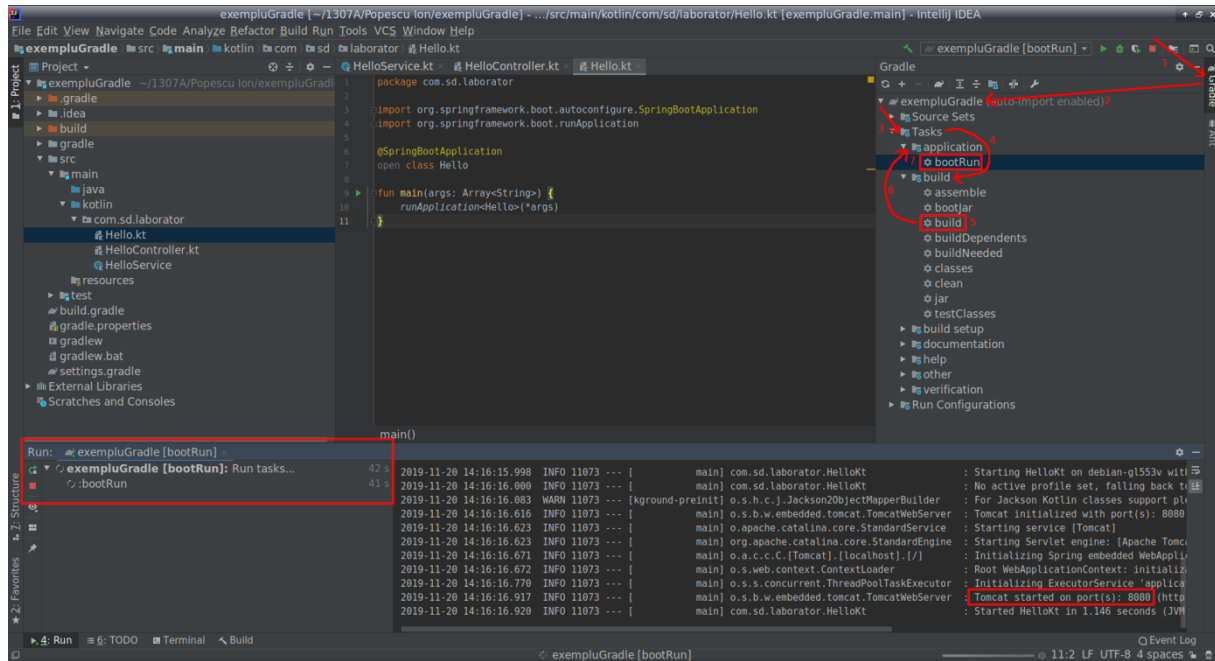
### 2.2.1. Compilare și execuție proiect *Maven*

Se va expanda meniul „Maven” din partea dreaptă a ferestrei și executați *lifecycle*-ul „**compile**”. După compilare, în caz că nu există erori, aplicația Spring se poate executa folosind un *Maven goal* denumit „**spring-boot:run**” din categoria „**Plugins** → **spring-boot**”. Se va porni automat un server **Apache Tomcat**, expus în mod implicit pe portul **8080**, care așteaptă cereri de la clienți.



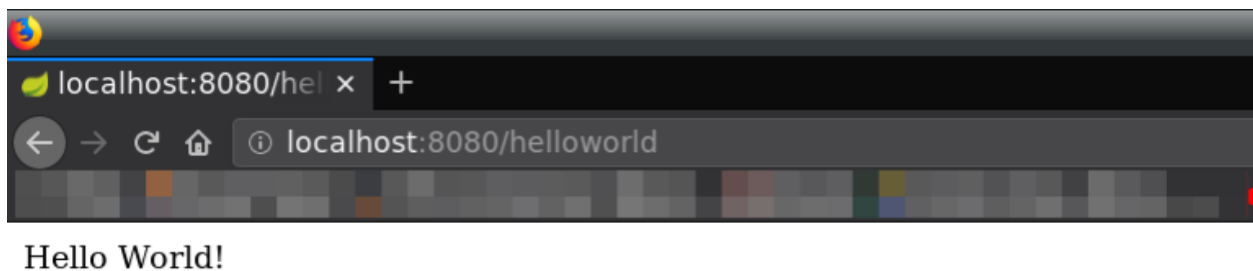
### 2.2.2. Compilare și execuție proiect *Gradle*

Se va expanda meniul „Gradle” din partea dreaptă a ferestrei și executați *task*-ul „**compile**” din categoria „**build**”. După compilare, în caz că nu există erori, aplicația Spring se poate executa folosind un *task* *Gradle* denumit „**bootRun**” din categoria „**Tasks** → **application**”. Se va porni automat un server **Apache Tomcat**, expus în mod implicit pe portul **8080**, care așteaptă cereri de la clienți.



### 2.3. Testarea funcționării aplicației Spring

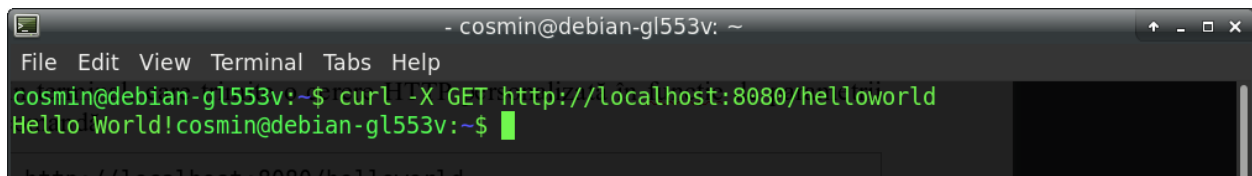
Puteți testa exemplul, navigând la adresa URL <http://localhost:8080/helloworld> cu un browser web. Ar trebui să se primească ca răspuns mesajul „Hello World!”.



De asemenea, o altă variantă de a testa funcționarea serviciului expus este folosirea comenzii „curl” din terminalul IntelliJ, care trimite o cerere HTTP personalizată în funcție de parametrii dați la linia de comandă:

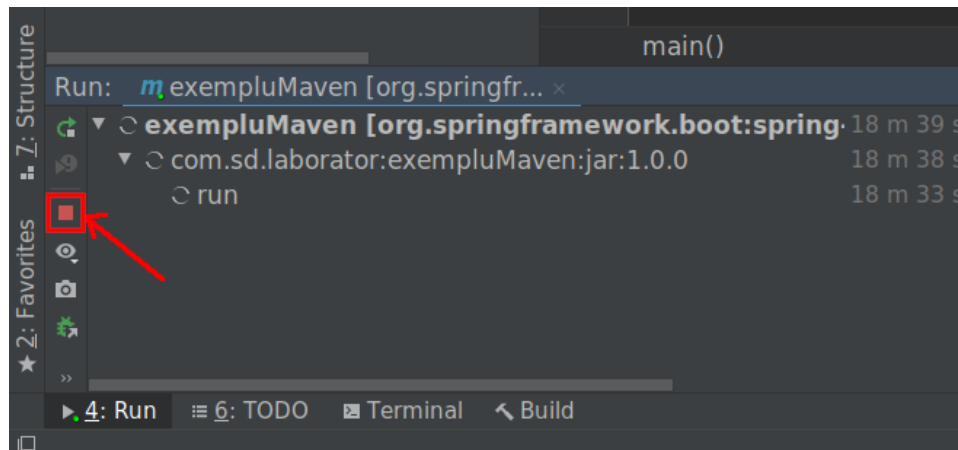
```
curl -X GET http://localhost:8080/helloworld
```

În acest exemplu, **curl** va trimite o cerere HTTP de tip **GET** către server-ul **localhost**, portul **8080**, și calea **/helloworld**.



```
- cosmin@debian-gl553v: ~  
File Edit View Terminal Tabs Help  
cosmin@debian-gl553v:~$ curl -X GET http://localhost:8080/helloworld  
Hello World!cosmin@debian-gl553v:~$
```

Aplicația își continuă execuția la infinit, deoarece server-ul funcționează în continuare, așteptând alte conexiuni din partea clienților. Pentru a opri aplicația, se va apăsa pe butonul roșu „Stop” în partea din stânga jos a ferestrei.

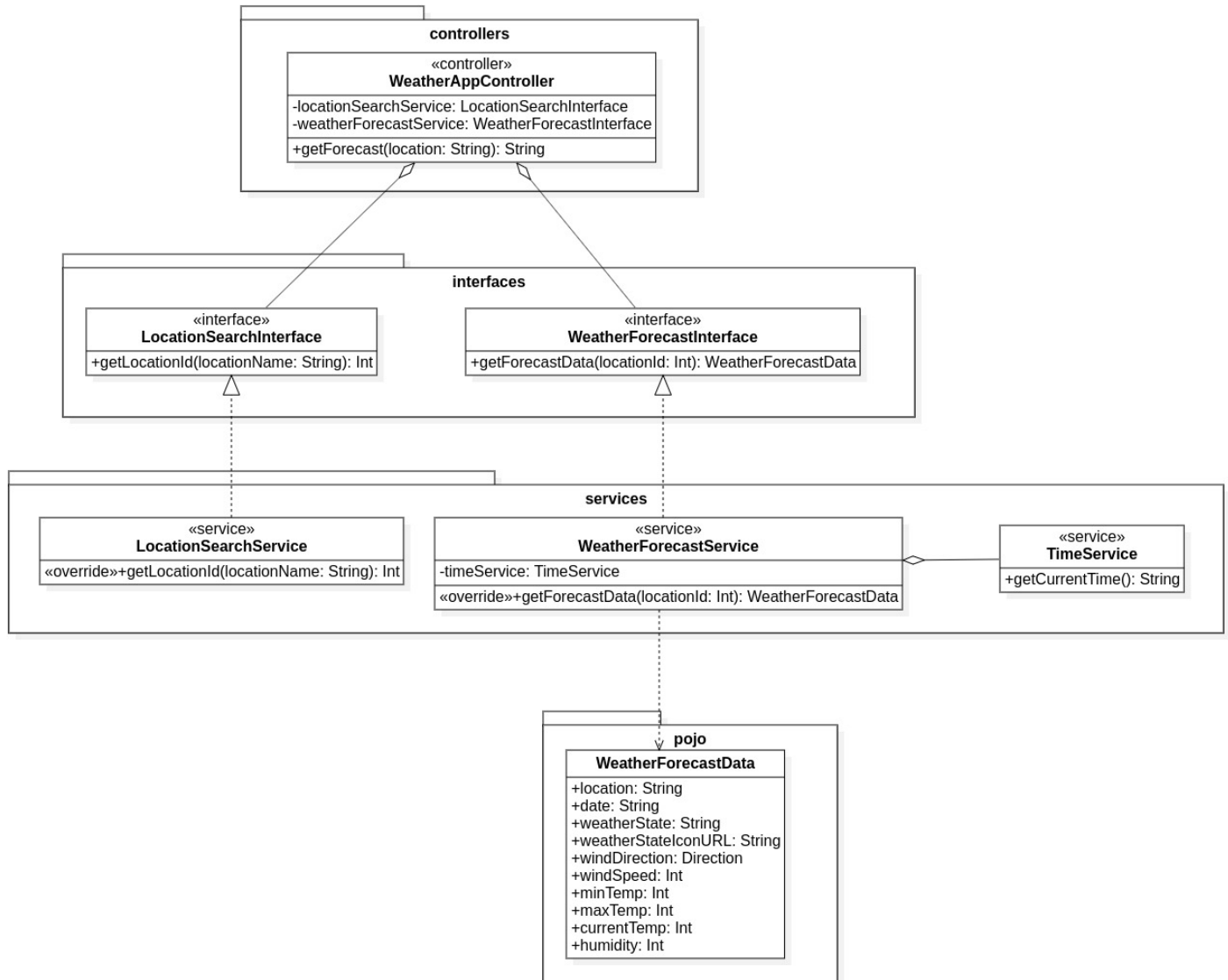


## 2.4. Modificarea surselor și reîncărcarea aplicației

Dacă modificați fișierele sursă și doriți să testați modificările făcute, datorită dependenței **Spring Boot Developer Tools**, trebuie doar să recompilați aplicația în IntelliJ (cu metoda corespunzătoare gestionarului de proiect ales), iar apoi Spring Boot va reîncărca automat artefactele modificate prin recompilare.

### 3. Aplicație meteo cu Spring Web

În următorul exemplu veți crea o aplicație meteo care preia câteva date despre vreme folosind servicii web dezvoltate cu *framework*-ul Spring. Arhitectura este reprezentată în diagrama următoare:



Clasele au fost împărțite în pachete în funcție de scopul acestora în modelarea aplicației. Spre exemplu, pentru implementare, se va folosi o abordare de tip „*bottom-up*”. Fiecare pachet, clasă, respectiv rolul în aplicație sunt explicate în cele ce urmează.

**Creați un proiect Spring Boot nou**, la care adăugați ca dependență **Spring Web** (exact cum ați procedat la aplicația demonstrativă din secțiunea anterioară), folosind un utilitar manager de proiect la alegere (Maven / Gradle). Denumiți-l, spre exemplu, **WeatherApp**.

**Nu vă speriați de blocurile de cod ce urmează. Sunt explicate pas cu pas!**

#### 3.1. Clasele POJO

Se pornește de la pachetul **pojo**, ce conține obiecte de tip *Plain Old Java Object*. Acestea sunt folosite pentru a trimite unui client datele cerute, încapsulate într-un obiect. De asemenea,

POJO-urile mai sunt utilizate în *layer*-ul de prezentare pentru afișarea datelor (de exemplu, dacă se folosește un *framework* de prezentare, cum ar fi *Thymeleaf*).

**Atenție: obiectele POJO nu procesează date.** Ele doar reprezintă rezultatul unei procesări anterioare (făcute în codul de *business*).

În acest caz, veți reprezenta datele meteo încapsulate într-un obiect POJO denumit **WeatherForecastData**. Pe lângă datele obișnuite, obiectul conține și un membru care reprezintă direcția din care bate vântul, sub formă de enumerație cu punctele cardinale.

Creați pachetul **com.sd.laborator.pojo** și adăugați clasa de mai sus sub formă de **data class**. În Kotlin, o clasă de tip **data class** are ca și scop încapsularea de date, iar datele sunt reprezentate de proprietăți mutabile sau imutabile. **Aceste clase trebuie obligatoriu să aibă un constructor primar.**

```
package com.sd.laborator.pojo

data class WeatherForecastData (
    var location: String,
    var date: String,
    var weatherState: String,
    var weatherStateIconURL: String,
    var windDirection: String,
    var windSpeed: Int, // km/h
    var minTemp: Int, // grade celsius
    var maxTemp: Int,
    var currentTemp: Int,
    var humidity: Int // procent
)
```

### 3.2. Serviciile web

În continuare, trebuie implementate serviciile web (clasele marcate cu stereotipul **service**). Pentru a ascunde și a nu depinde de implementarea serviciilor atunci când sunt utilizate de *controller*, se folosesc **interfețe** (unde este cazul). Cu excepția serviciului **TimeService**, care este utilizat de un alt serviciu, celelalte 2 derivă din interfețe.

Începeți cu serviciul **TimeService**, deoarece este cel mai simplu. Acesta expune o metodă **getCurrentTime()** ce returnează data și ora curentă sub formă de șir de caractere. Creați clasa **TimeService** într-un pachet **services** subordonat pachetului principal **com.sd.laborator**.

```
package com.sd.laborator.services

import org.springframework.stereotype.Service
import java.text.SimpleDateFormat
import java.util.*

@Service
class TimeService {
    fun getCurrentTime():String {
        val formatter = SimpleDateFormat("dd/MM/yyyy HH:mm:ss")
        return formatter.format(Date())
    }
}
```

Clasa este adnotată cu **@Service** pentru ca *framework*-ul Spring să o poată găsi și configura corespunzător sub formă de **serviciu** în timpul scanării de componente.

Pentru celelalte 2 servicii web, începeți cu interfețele. Creați un pachet **interfaces**

subordonat pachetului principal și adăugați cele 2 interfețe:

### • LocationSearchInterface

```
package com.sd.laborator.interfaces

interface LocationSearchInterface {
    fun getLocationId(locationName: String): Int
}
```

Aceasta expune o metodă pentru preluarea unui identificator de locație (**WOEID - Where On Earth ID**) ce va fi folosit mai departe la prognoza meteo. De unde a apărut acest identificator? Este utilizat de API-ul pe care îl veți folosi aici pentru a prelua datele despre vreme, și anume MetaWeather API: <https://www.metaweather.com/api/>. S-a ales acest API deoarece este ușor de folosit și **nu necesită cont și cheie API**.

### • WeatherForecastInterface

```
package com.sd.laborator.interfaces

import com.sd.laborator.pojo.WeatherForecastData

interface WeatherForecastInterface {
    fun getForecastData(locationId: Int): WeatherForecastData
}
```

Aici se expune o metodă care, pe baza unui identificator de tip **WOEID**, returnează un obiect ce încapsulează toate informațiile meteo.

Având interfețele definite, se continuă cu implementările:

### • LocationSearchService

```
package com.sd.laborator.services

import com.sd.laborator.interfaces.LocationSearchInterface
import org.springframework.stereotype.Service
import java.net.URL
import org.json.JSONObject
import java.net.URLEncoder
import java.nio.charset.StandardCharsets

@Service
class LocationSearchService : LocationSearchInterface {
    override fun getLocationId(locationName: String): Int {
        // codificare parametru URL (deoarece poate conține caractere speciale)
        val encodedLocationName = URLEncoder.encode(locationName, StandardCharsets.UTF_8.toString())

        // construire obiect de tip URL
        val locationSearchURL =
            URL("https://www.metaweather.com/api/location/search/?query=$encodedLocationName")

        // preluare raspuns HTTP (se face cerere GET și se preia conținutul răspunsului sub formă de text)
```

```

val rawResponse: String = locationSearchURL.readText()

// parsare obiect JSON
val responseRootObject = JSONObject("{\"data\": ${rawResponse}}")
val responseContentObject =
responseRootObject.getJSONArray("data").takeUnless { it.isEmpty }
    ?.getJSONObject(0)
return responseContentObject?.getInt("woeid") ?: -1
    }
}

```

**Atentie:** pentru a utiliza clasa **JSONObject**, adăugați artefactul **org.json.json** (<https://mvnrepository.com/artifact/org.json/json>) ca **dependență globală**:

- **pentru proiecte Maven:** adăugați următoarea dependență în **pom.xml**:

```

<dependency>
  <groupId>org.json</groupId>
  <artifactId>json</artifactId>
  <version>20190722</version>
</dependency>

```

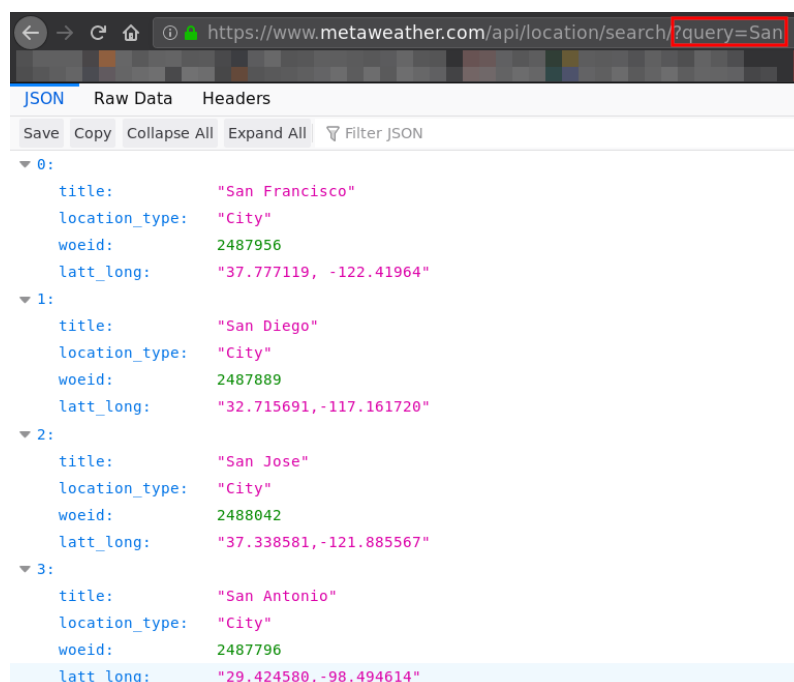
- **pentru proiecte Gradle:** adăugați următoarea dependență în interiorul secțiunii **dependencies**:

```
compile group: 'org.json', name: 'json', version: '20190722'
```

Implementarea acestui serviciu procedează în felul următor: trimite o cerere HTTP către un URL în format standard (descriș în API-ul din documentație), prin care se poate căuta o anumită locație pe baza numelui complet sau parțial. URL-ul este de forma:

[https://www.metaweather.com/api/location/search/?query=\\$locationName](https://www.metaweather.com/api/location/search/?query=$locationName)

Răspunsul returnat de serviciul MetaWeather la acest tip de interogare poate fi verificat direct din browser și este de forma:



Se observă că MetaWeather răspunde cu un obiect JSON ce conține un vector cu rezultatele căutării utilizatorului (în acest caz, s-a căutat după cuvântul cheie „**San**”). Pentru simplitate, **se va lua în considerare doar primul rezultat al căutării** (dacă există), deci primul element din vectorul returnat.

Pentru parsarea obiectului JSON, se folosește clasa **JSONObject**, care primește în constructor acel obiect sub formă de șir de caractere. Deoarece la această interogare se returnează un vector care nu este încapsulat într-un obiect JSON părinte, se adaugă manual un element rădăcină denumit **data**:

```
val responseRootObject = JSONObject("{\"data\": ${rawResponse}}")
```

Conform standardului JSON, trebuie obligatoriu să existe un element rădăcină ce încapsulează restul elementelor. Dacă nu faceți această încapsulare „la mână”, clasa **JSONObject** va genera o excepție la parsare.

În continuare, se preia obiectul de tip vector ca și elementul copil al **data: responseRootObject.getJSONArray("data")**. În cazul în care căutarea locației nu a returnat niciun rezultat, obiectul rezultat va fi un vector gol, și deci nu se dorește apelul niciunei metode asupra acestuia. Așadar, se folosește (spre exemplu), blocul **takeUnless** specific Kotlin, care, pe baza predicatului **isEmpty**, apelează înălțuit metoda **getJSONObject()** doar în cazul în care obiectul apelant satisface predicatul (adică vectorul nu este gol).

```
val responseContentObject =  
responseRootObject.getJSONArray("data").takeUnless { it.isEmpty }  
?.getJSONObject(0)
```

Blocul **takeUnless**, urmat de **getJSONObject(0)** returnează deci conținutul primului element din vectorul cu rezultate, adică ceea ce este evidențiat în figură:

▼ 0:

title:	"San Francisco"
location_type:	"City"
woeid:	2487956
latt_long:	"37.777119, -122.41964"

Reamintire: s-a folosit operatorul *safe call* (**?.**) din Kotlin, deoarece **takeUnless** returnează **null** în cazul în care predicatul trimis nu este satisfăcut.

Având acces direct la elementele de interes, se poate apela acum **get<TIP\_DE\_DATE>(nume\_proprietate)** pentru a prelua ceea ce se dorește, de exemplu câmpul **woeid**, de tip număr întreg:

```
responseContentObject?.getInt("woeid") ?: -1
```

Acest câmp va fi folosit mai departe pentru prognoza meteo a locației cu identificatorul respectiv. S-a folosit și aici operatorul *safe call*, în conjuncție cu operatorul *null coalescing* (**?:**) pentru a returna **-1** dacă nu s-a găsit nicio locație pentru cuvintele cheie căutate de utilizator.

#### • WeatherForecastService

```
package com.sd.laborator.services  
  
import com.sd.laborator.interfaces.WeatherForecastInterface  
import com.sd.laborator.pojo.WeatherForecastData  
import org.json.JSONObject
```



```

import org.springframework.stereotype.Service
import java.net.URL
import kotlin.math.roundToInt

@Service
class WeatherForecastService (private val timeService: TimeService) :
WeatherForecastInterface {
    override fun getForecastData(locationId: Int): WeatherForecastData {
        // ID-ul locației nu trebuie codificat, deoarece este numeric
        val forecastDataURL =
URL("https://www.metaweather.com/api/location/$locationId/")

        // preluare conținut răspuns HTTP la o cerere GET către URL-ul de
mai sus
        val rawResponse: String = forecastDataURL.readText()

        // parsare obiect JSON primit
        val responseObject = JSONObject(rawResponse)
        val weatherDataObject =
responseObject.getJSONArray("consolidated_weather").getJSONObject(0)

        // construire și returnare obiect POJO care încapsulează datele
meteo
        return WeatherForecastData(
            location = responseObject.getString("title"),
            date = timeService.getCurrentTime(),
            weatherState =
weatherDataObject.getString("weather_state_name"),
            weatherStateIconURL =

"https://www.metaweather.com/static/img/weather/png/${weatherDataObject.g
etString("weather_state_abbr")}.png",
            windDirection =
weatherDataObject.getString("wind_direction_compass"),
            windSpeed =
weatherDataObject.getFloat("wind_speed").roundToInt(),
            minTemp =
weatherDataObject.getFloat("min_temp").roundToInt(),
            maxTemp =
weatherDataObject.getFloat("max_temp").roundToInt(),
            currentTemp =
weatherDataObject.getFloat("the_temp").roundToInt(),
            humidity =
weatherDataObject.getFloat("humidity").roundToInt()
        )
    }
}

```

Ca și principiu de funcționare, acest serviciu este asemănător cu **LocationSearchService**. URL-ul expus de API-ul MetaWeather pentru preluarea datelor meteo este de forma:

```
https://www.metaweather.com/api/location/WOEID_PRELUAT_ANTERIOR/
```

Conținutul răspunsului HTTP este de forma:



Observați că răspunsul este un obiect JSON ce conține deja datele încapsulate într-un element rădăcină, numit **consolidated\_weather**. Ca și subordonat, se primește un vector cu date meteo de la mai mulți furnizori. MetaWeather agregă rezultate de la mai mulți furnizori meteo și le încapsulează într-un singur răspuns. Din nou, pentru simplitate, luați în considerare doar răspunsul de la primul furnizor (primul element din vectorul rezultat).

Așadar, se preia din prima obiectul conținut în primul element al vectorului respectiv:

```
val weatherDataObject =
responseRootObject.getJSONArray("consolidated_weather").getJSONObject(0)
```

Deci, variabila `weatherDataObject` încapsulează următoarele date în acest moment:

```

▼ consolidated_weather:
  ▼ 0:
    id: 5764786186878976
    weather_state_name: "Clear"
    weather_state_abbr: "c"
    wind_direction_compass: "WSW"
    created: "2020-01-13T10:20:11.665627Z"
    applicable_date: "2020-01-13"
    min_temp: -0.11499999999999999
    max_temp: 7.02
    the_temp: 4.734999999999999
    wind_speed: 5.828544966945041
    wind_direction: 245.16662387203695
    air_pressure: 1027
    humidity: 77
    visibility: 9.46037640181341
    predictability: 68
  ▼ 1:
    id: 6132782851948544
    weather_state_name: "Light Cloud"
    weather_state_abbr: "lc"
    wind_direction_compass: "WSW"
    created: "2020-01-13T10:20:14.667301Z"

```

Deoarece acum aveți acces direct (prin această variabilă) la proprietățile de interes din obiectul JSON, informațiile se pot prelua cu metodele de tipul `get<TIP_DE_DATE>(nume_proprietate)`:

- starea generală a vremii:

```
weatherDataObject.getString("weather_state_name")
```

- viteza vântului, rotunjită la întreg

```
weatherDataObject.getFloat("wind_speed").roundToInt()
```

ș.a.m.d.

Excepție face numele locației pentru care se afișează prognoza, care face parte din elementul rădăcină `consolidated_weather`, și deci se preia astfel:

```
responseRootObject.getString("title")
```

### 3.3. Clasa controller

```

package com.sd.laborator.controllers

import com.sd.laborator.interfaces.LocationSearchInterface
import com.sd.laborator.interfaces.WeatherForecastInterface
import com.sd.laborator.pojo.WeatherForecastData
import com.sd.laborator.services.TimeService
import org.springframework.beans.factory.annotation.Autowired
import org.springframework.stereotype.Controller
import org.springframework.web.bind.annotation.PathVariable
import org.springframework.web.bind.annotation.RequestMapping
import org.springframework.web.bind.annotation.RequestMethod

```

```

import org.springframework.web.bind.annotation.ResponseBody

@Controller
class WeatherAppController {
    @Autowired
    private lateinit var locationSearchService: LocationSearchInterface

    @Autowired
    private lateinit var weatherForecastService: WeatherForecastInterface

    @RequestMapping("/getforecast/{location}", method =
[RequestMethod.GET])
    @ResponseBody
    fun getForecast(@PathVariable location: String): String {
        // se incearca preluarea WOEID-ului locației primite in URL
        val locationId = locationSearchService.getLocationId(location)

        // dacă locația nu a fost găsită, răspunsul va fi corespunzător
        if (locationId == -1) {
            return "Nu s-au putut gasi date meteo pentru cuvintele cheie
\"$location\"!"
        }

        // pe baza ID-ului de locație, se interoghează al doilea serviciu
care returnează datele meteo
        // încapsulate într-un obiect POJO
        val rawForecastData: WeatherForecastData =
weatherForecastService.getForecastData(locationId)

        // fiind obiect POJO, funcția toString() este suprascrisă pentru
o afișare mai prietenoasă
        return rawForecastData.toString()
    }
}

```

Adnotarea **@Autowired** marchează faptul că proprietățile **locationSearchService**, respectiv **weatherForecastService** sunt dependente rezolvate prin facilitățile *dependency injection* ale *framework*-ului Spring. După ce Spring construiește *bean*-ul corespunzător clasei **WeatherAppController**, rezolvă și injectează automat aceste 2 dependențe (în acest caz, serviciile de care *controller*-ul are nevoie).

Observați că aceste 2 proprietăți nu sunt instanțiate nicăieri în clasă, ci sunt doar declarate. **Inițializarea este făcută automat de Spring**. De aceea a fost nevoie de specificatorul **lateinit**, deoarece Kotlin nu permite declararea unei proprietăți fără inițializarea acesteia în constructor sau imediat după declarație.

Mai mult de atât, tipurile de date al proprietăților respective corespund interfețelor serviciilor create anterior, **și nu implementărilor**. Motivul este, din nou, decuplarea *controller*-ului de implementarea efectivă a serviciilor dependente: dacă dezvoltatorul vrea să schimbe unul din servicii și să folosească o altă clasă **care respectă același contract** (aceeași interfață), atunci ar trebui modificat și recompilat *controller*-ul! În acest caz, dacă doriți să modificați complet serviciul de căutare a locației, spre exemplu, pur și simplu creați o altă clasă care implementează interfața **LocationSearchInterface** și nu modificați nimic altceva.

În *controller* se definește o metodă de tratare a cererilor HTTP de tip GET către calea **/getforecast/{location}**. **location** este un parametru trimis de utilizator direct în URL, așadar acest parametru este injectat ca parametru al metodei **getForecast()** sub formă de

variabilă de cale (*path variable*). De aici adnotarea `@PathVariable`.

În continuare, sunt utilizate serviciile disponibile pentru a:

1. Căuta WOEID-ul pe baza cuvintelor cheie primite de la utilizator
2. Returna datele meteo specifice celui WOEID sub formă de obiect POJO

### 3.4. Punctul de intrare în aplicația Spring

Având toate elementele componente definite, se creează clasa principală ce reprezintă punctul de intrare (*entrypoint*) al aplicației Spring. Aici se declară clasa de configurare, în care se pot face configurări suplimentare.

Adăugați un fișier nou `WeatherApp.kt` în pachetul `com.sd.laborator`, cu următorul conținut:

```
package com.sd.laborator

import org.springframework.boot.autoconfigure.SpringBootApplication
import org.springframework.boot.runApplication

@SpringBootApplication
class WeatherApp

fun main(args: Array<String>) {
    runApplication<WeatherApp>()
}
```

### 3.5. Testarea aplicației

Accesați următorul URL după compilarea și pornirea aplicației Spring Boot:

<http://localhost:8080/getforecast/Bucharest>

Ar trebui să primiți un răspuns ce conține datele meteo pentru locația **București**. Testați și pentru alte locații.

**Atenție:** nu sunt disponibile toate locațiile, mai ales cele din România.

```
cosmin@debian-gl553v:~$ curl -X GET http://localhost:8080/getforecast/Bucharest
WeatherForecastData(location=Bucharest, date=13/01/2020 12:10:52, weatherState=Clear, weatherStateIconURL=https://www.metaweather.com/static/img/weather/png/c.png, windDirection=WSW, windSpeed=6, minTemp=-1, maxTemp=8, currentTemp=5, humidity=78)cosmin@debian-gl553v:~$
```

Pentru a folosi spații în cuvintele cheie, adăugați codificarea corespunzătoare caracterului spațiu în URL: `%20`. Spre exemplu, pentru a se căuta „San Francisco”, URL-ul va fi de forma:

<http://localhost:8080/getforecast/San%20Francisco>

```
cosmin@debian-gl553v:~$ curl -X GET http://localhost:8080/getforecast/San%20Francisco
WeatherForecastData(location=San Francisco, date=13/01/2020 14:12:59, weatherState=Light Rain, weatherStateIconURL=https://www.metaweather.com/static/img/weather/png/lr.png, windDirection=WNW, windSpeed=7, minTemp=8, maxTemp=12, currentTemp=12, humidity=66)cosmin@debian-gl553v:~$
```

## Aplicații și teme propuse

- **Temă de laborator**

Reproiectați diagrama de clase a aplicației meteo, cu respectarea principiilor de proiectare și modificați aplicația corespunzător.

- **Temă pe acasă**

1. Reproiectați și reimplementați aplicația meteo astfel încât să utilizați înlănțuirea (engl. *chaining*) de servicii.
2. Reproiectați și reimplementați aplicația meteo astfel încât să utilizați orchestrarea (engl. *orchestration*) de servicii.

## Bibliografie

- [1] Apache Maven - <https://maven.apache.org/>
- [2] Spring Boot - <https://spring.io/projects/spring-boot>
- [3] Project Object Model (POM) - <https://maven.apache.org/guides/introduction/introduction-to-the-pom.html>
- [4] Gradle Build Tool - <https://gradle.org/>
- [5] Documentația Spring – <https://docs.spring.io>
- [6] Documentația de referință Spring Boot – <https://docs.spring.io/spring-boot/docs/current/reference/html/>
- [7] Clase de tip Data Class în Kotlin - <https://kotlinlang.org/docs/reference/data-classes.html>
- [8] Metaweather API - <https://www.metaweather.com/api/>