Ćwiczenie wstępne

Projekt Maven (dla XML-RPC)

Author: Mariusz Fraś

1 Objectives of the exercise

The purpose of the exercise is:

1. Familiarization with the development of Maven projects.
2. Understanding the general architecture of the XML-RPC application.

2 Maven (*content is based mainly on apache.org website*)

Maven – a tool (mechanism) that is used for building and managing Java-based projects – among others: building/compiling the projects, defining components of what the project consists of and its dependences, publishing project information, sharing JARs across several projects. Maven is **a plugin execution framework**. All work is done by plugins.

POM (Project Object Model) is the fundamental unit of work in Maven. It is an XML file that contains information about the project and configuration details used by Maven to build the project. POM file is an **essential element of maven project**.
Some of basic elements specified in POM are:
– project dependencies (packages to be used),
– the plugins or goals that can be executed,
and many others.
A plugin **goal** is a specific task which contributes to the building and managing of a project.

The default POM of new project in IntelliJ.IDEA shows minimal requirements for POM:
- **project** – root element,
- **modelVersion** – version of the object model this POM is using – should be set to 4.0.0,
- **groupId** – identifier of the organization or group that created the project (typically based on the fully qualified domain name of organization),
- **artifactId** – the unique base name of the primary artifact being generated by this project – typically project module name (the primary artifact for a project is typically a JAR file),
- **version** – the version of the artifact under the specified group.

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
        ...>
    <modelVersion>4.0.0</modelVersion>

    <groupId>organization.domain.name</groupId>
    <artifactId>unique_name</artifactId>
    <version>1.0-SNAPSHOT</version>
    <packaging>pom</packaging>

</project>
```

An **artifact** is an element that a project can either use or produce. Maven artifacts may include five key elements, **groupId**, **artifactId**, **version**, **packaging**, and **classifier**. Any component must be specified by at least three of them: **groupId**, **artifactId**, and **version**.

The default POM file also contains properties section/node. It contains value placeholders accessible anywhere within a POM. E.g. it may be contain java compiler version constants or version number of some package/library – e.g.:

```
<project ...>
    ...
    <properties>
        <maven.compiler.source>17</maven.compiler.source>
        <xmlrpc.version>3.1.3</xmlrpc.version>
        ...
    </properties>
</project>
```

Properties are accessible using the notation **${property_name}**

Dependency management is crucial feature of maven projects. It permits define (and finally import) components (e.g. given packages with used classes) the project depends on. Dependency management is based on some concepts, among others:

- repositories that stores artifacts – local and remote (default is Maven repository),
- inheritance (from parent components – e.g. project modules inherit from main POM),
- scope – allows to specify in which context use a dependency (*compile* scope is default).

Dependences are defined in **<dependency>** node in **<dependencies>** section. In below example we include in the project library for building Netty server based application:

```
<project ...>
    ...
    <dependencies>
        <dependency>
            <groupId>io.netty</groupId>
            <artifactId>netty-all</artifactId>
            <version>4.1.89.Final</version>
        </dependency>
    </dependencies>
</project>
```

Build lifecycle and phases.
Maven build follows a specified lifecycle to compile, deploy, distribute, etc. a particular artifact (target project). The three built-in lifecycles are:

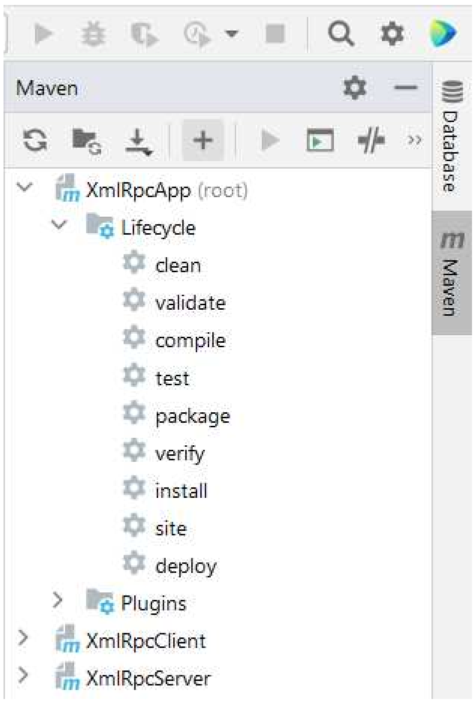
- - **clean** – to clean the project and remove all generated files,
- - **default** – the main lifecycle responsible for project deployment,
- - **site** – to create the project's documentation.

A **lifecycle** consists of a sequence of **phases**. The default build lifecycle consists of 23 phases. A **phase** represents a stage in the Maven build lifecycle. Each phase is responsible for a specific task (sequence of **goal**s).

Some important phases in the default build lifecycle are:

- **validate** – validate the project is correct and all necessary information is available
- **compile** – compile the source code
- **test** – run unit tests
- **package** – package compiled source code into the distributable format (jar, war, …)
- **verify** – run checks on results of integration tests to ensure quality criteria are met
- **install** – install the package to a local repository
- **deploy** – copy the package to the remote repository

These tasks are available in Maven tab of IntelliJ.IDEA platform (from the right).



Plugins and goals

To customize the build for a Maven project you may add or customize a plugin. A Maven plugin is a group of goals. The goals aren't necessarily all bound to the same phase.

The plugins (also its configuration) and goals ale declared in **<plugin>** node in **<build>** section of POM. The goals (and also configuration) are placed in the **<goal>** node in the **<execution>** part.

Here is an example of declaring (1) MFT Maven plugin that formats a project's Java Code following google-java-format, and (2) Maven Failsafe plugin, which is responsible for running integration tests:

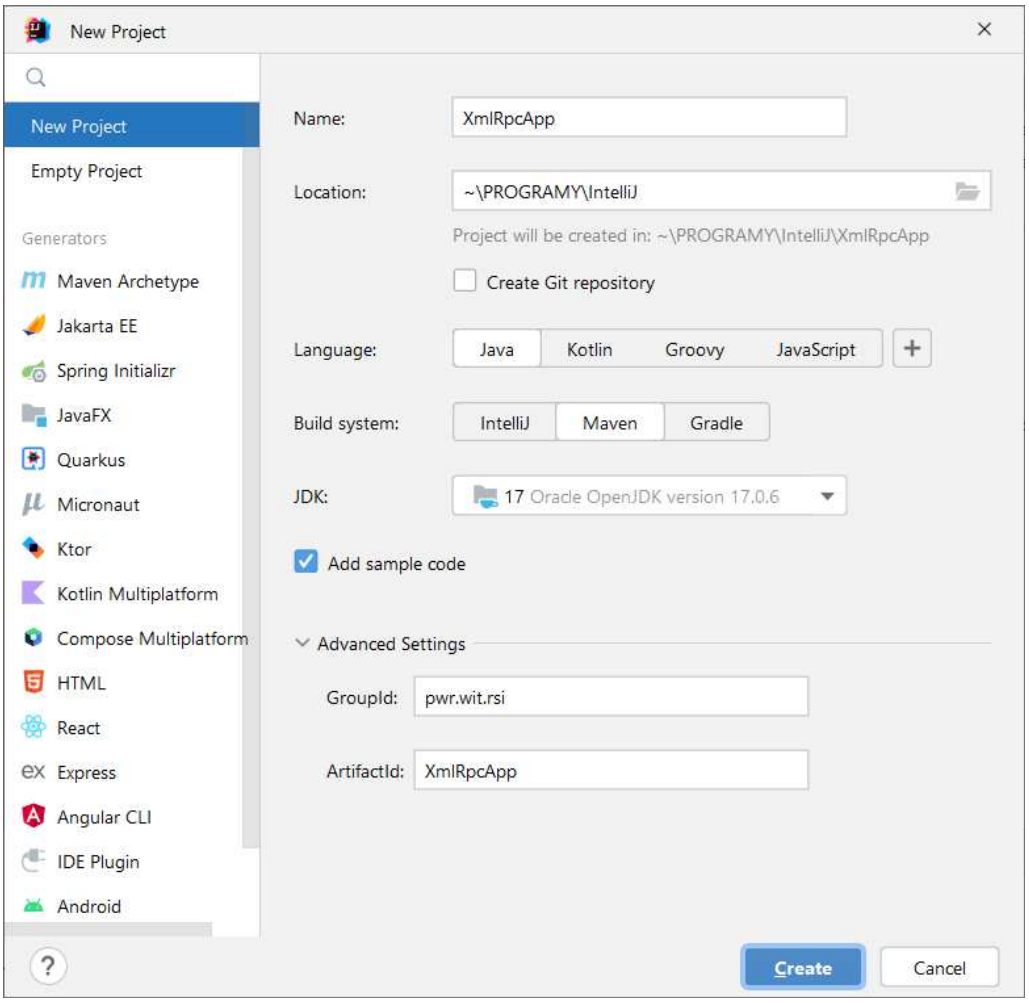
```
</project ...>
    ...
    <build>
        <plugins>
            <plugin>
                <groupId>com.spotify.fmt</groupId>
                <artifactId>fmt-maven-plugin</artifactId>
                <version>2.19</version>
                <executions>
                    <execution>
                        <goals>
                            <goal>format</goal>
                        </goals>
                    </execution>
                </executions>
            </plugin>

            <plugin>
                <artifactId>maven-failsafe-plugin</artifactId>
                <version>${maven.failsafe.version}</version>
                <executions>
                    <execution>
                        <goals>
                            <goal>integration-test</goal>
                            <goal>verify</goal>
                        </goals>
                    </execution>
                </executions>
            </plugin>
        </plugins>
    </build>

</project>
```

3 Exercise – Part I
Example Maven project for XML-RPC application

Here we build **XmlRpcApp** application project that consists of two modules: XML-RPC server and XML-RPC client. The app uses Apache XML-RPC lib.

3.1 Maven project

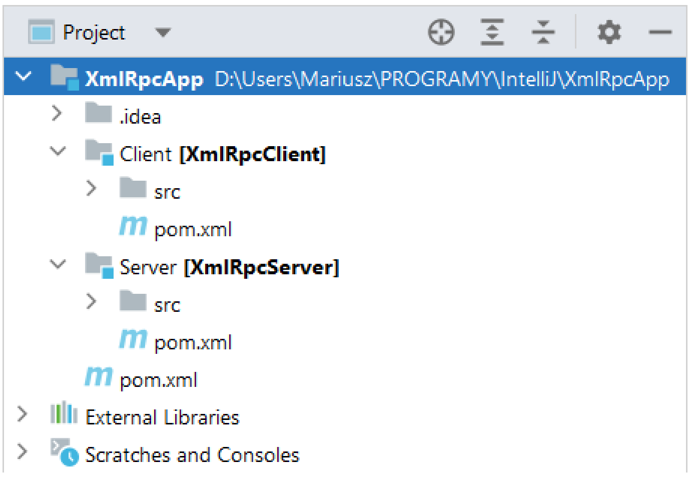
Maven is integrated in IntelliJ.IDEA development platform. To create maven project set *Maven* option for *Build system* in *New Project* window. You may also set here *GroupId* and *ArtifactId* for the project.

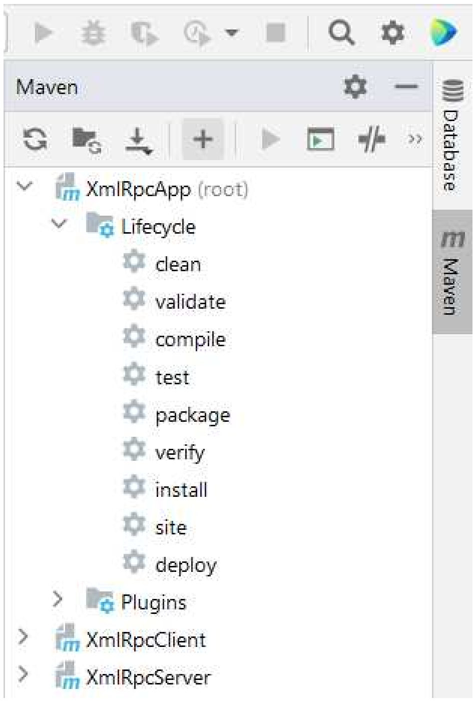


Name: XmlRpcApp
Location: ~\PROGRAMY\IntelliJ
Project will be created in: ~\PROGRAMY\IntelliJ\XmlRpcApp
Create Git repository
Language: Java Kotlin Groovy JavaScript
Build system: IntelliJ Maven Gradle
JDK: 17 Oracle OpenJDK version 17.0.6
Add sample code
Advanced Settings
GroupId: pwr.wit.rsi
ArtifactId: XmlRpcApp

3.1.1 General project structure

The project is multi-module maven project. It consist of **XmlRpcServer**, and **XmlRpcClient** modules as presented in the figure below (on the left side: project structure). It contains one shared POM file and three POMs of each module. In shared POM are defined shared components/dependences/goals…





- Create the maven project composed of mentioned modules.
  Note: one of possible way is to create new maven project (here named **XmlRpcApp**) with default module, next remove src node/folder (leave **pom.xml** file), and then add two mentioned child modules **Server**, and **Client** (their parent is XmlRpcApp) – using context menu or from main menu ***File*** → ***New*** → ***Module…*** :

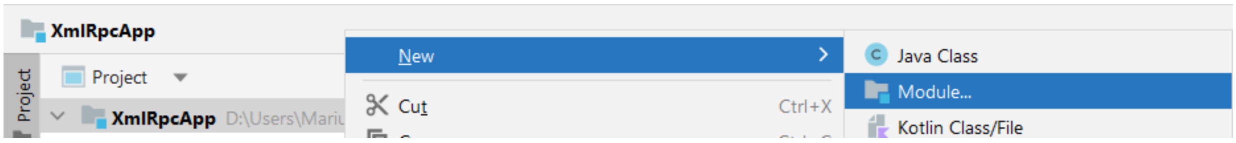


Fig. adding module to the project.

3.1.2 POMs

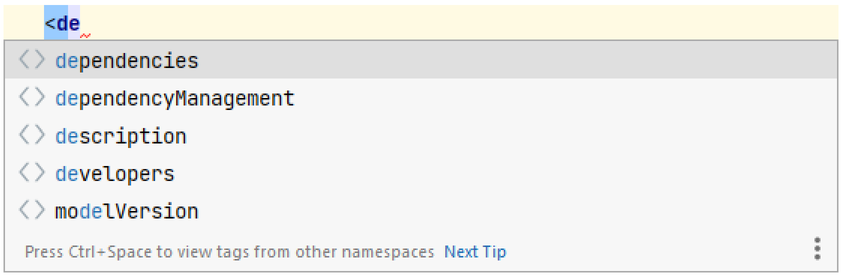
- In the main POM (pom.xml) define the following:
  – two component modules (shoul be added by the IDE automatically)

```
<modules>
    <module>XMLRpcServer</module>
    <module>XMLRpcClient</module>
</modules>
```

  – necessary dependences – packages used to build application – here one: Apache XML RPC Common Library (**org.apache.xmlrpc - xmlrpc-common**) that contains some classes necessary to operate on XML-RPC messages:

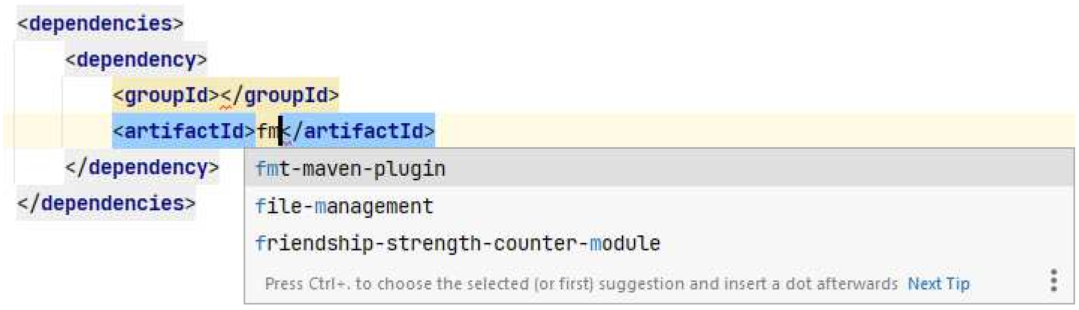
```
<dependency>
    <groupId>org.apache.xmlrpc</groupId>
    <artifactId>xmlrpc-common</artifactId>
    <version>3.1.3</version>
</dependency>
```

Pay attention how the IDE supports entering the code – see figures below.



```
<de
<> dependencies
<> dependencyManagement
<> description
<> developers
<> modelVersion
Press Ctrl+Space to view tags from other namespaces  Next Tip
```

Fig. Entering the <dependences> node.



```
<dependencies>
    <dependency>
        <groupId></groupId>
        <artifactId>fm</artifactId>
    </dependency>
</dependencies>
```

```
fmt-maven-plugin
file-management
friendship-strength-counter-module
Press Ctrl+. to choose the selected (or first) suggestion and insert a dot afterwards  Next Tip
```

Fig. Entering the fmt-maven-plugin.

- In the **Server** POM define the following:
  - parent module (should be added by development platform during previous project configuration),

```
<project ...>
    <parent>
        <artifactId>XmlRpcExample</artifactId>
        <groupId>pwr.wit.rsi</groupId>
        <version>1.0-SNAPSHOT</version>
    </parent>
    ...
</project>
```

  - necessary dependences – packages used to build application – here one: Apache XML RPC Server Library (**org.apache.xmlrpc - xmlrpc-server**) that contains classes necessary to build XML-RPC server:

```
<dependency>
    <groupId>org.apache.xmlrpc</groupId>
    <artifactId>xmlrpc-server</artifactId>
    <version>3.1.3</version>
</dependency>
```

- In the **client** POM define the following:
  - parent module as above (should be added by development platform during previous project configuration),
  - necessary dependences – here: Apache XML RPC Client Library (**org.apache.xmlrpc - xmlrpc-client**) that contains classes necessary to build XML-RPC client and Apache HttpComponents Client (**org.apache.httpcomponents.client5 - httpclient5**):

```xml
        <dependencies>
            <dependency>
                <groupId>org.apache.xmlrpc</groupId>
                <artifactId>xmlrpc-client</artifactId>
                <version>3.1.3</version>
            </dependency>
            <dependency>
                <groupId>org.apache.httpcomponents.client5</groupId>
                <artifactId>httpclient5</artifactId>
                <version>5.2.1</version>
            </dependency>
        </dependencies>
```

***Note**: Some used libs are a bit old and NOT vulnerability free. Instead, you may use newer libs forked from org.apache.xml*:

```xml
<groupId>com.evolvedbinary.thirdparty.org.apache.xmlrpc</groupId>
<artifactId>xmlrpc-common</artifactId>
<version>5.0.0</version>

<groupId>com.evolvedbinary.thirdparty.org.apache.xmlrpc</groupId>
<artifactId>xmlrpc-server</artifactId>
<version>5.0.0</version>

<groupId>com.evolvedbinary.thirdparty.org.apache.xmlrpc</groupId>
<artifactId>xmlrpc-client</artifactId>
<version>5.0.0</version>
```

- Define additionally in main POM the plugin (and goal) that formats a project's Java Code following google-java-format.
  Include in the main POM's **<build><plugins>** section the plugin and goal:

```xml
<build>
    <plugins>
        <plugin>
            <groupId>com.spotify.fmt</groupId>
            <artifactId>fmt-maven-plugin</artifactId>
            <version>2.19</version>
            <executions>
                <execution>
                    <goals>
                        <goal>format</goal>
                    </goals>
                </execution>
            </executions>
        </plugin>
    </plugins>
</build>
```

3.2 Application code

3.2.1 Server code

In server java code file (here **XmlRpcServer** class) you must minimally define:

– the class implementing services – containing remote procedures code called from remote client – here two procedures:
  (1) called synchronously: `public String echo(String message)`
  and (2) called asynchronously:

```
public String echoWithDelay(String message)
```

In the main method:

- create **server** object working on given port – the instance of **org.apache.xmlrpc.webserver.WebServer** class
- get XML-RPC **stream server** (org.apache.xmlrpc.server.**XmlRpcStreamServer**)
- create **handler mapping object** (for mapping handlers to services) and add handler to defined service implementation (here build service class object).
- set handler mapping for stream server
- start and finally down the server.

So:

- Add in the Server module and define the service class (here named **MessageService**):

```
public class MessageService {
  private static final int DELAY_SEC = 3;
  public String echo(String message) {
    System.out.println("Message received:", message));
    System.out.println("Response: sending echo message..."));
    return "Echo: " + message;
  }
  public String echoWithDelay(String message) throws InterruptedException {
    System.out.println("Message received:", message));
    System.out.println("Waiting for %d seconds before sending response...");
    Thread.sleep(DELAY_SEC * 1000);
    System.out.println("Response: sending echo message..."));
    return "Echo after delay: " + message;
  }
}
```

- In the server class in **main** method:

```
public class Server {
  private static final int port = 8080;
  public static void main(String[] args) {
    var webServer = new WebServer(port);
    var xmlRpcServer = webServer.getXmlRpcServer();
    var handler = new PropertyHandlerMapping();
    try {
      handler.addHandler("MessagesService", MessageService.class);
      xmlRpcServer.setHandlerMapping(handler);
      webServer.start();
      System.out.println(" Server started...");
    } catch (Exception e) {
      System.out.println("Something went wrong!");
    }
  }
}
```

3.2.2 Client code

In client java code file (here **XmlRpcClient** class) you must:

- define **configuration** for the client (defining e.g. working port, timeouts, etc.),
- create client object (instance of **org.apache.xmlrpc.client.XmlRpcClient** class),
- set/define the **transport** for the client (here standard http transport),
- set configuration for the client,
- create a **callback objects** (instances of **TimingOutCallback** class) that can wait up to a specified amount of time for the XML-RPC responses,
- call the remote procedures (sync and async),
- wait (call callback's **waitForResponse** method) and print the results – here separate **getAsyncResponse(TimingOutCallback)** method is defined for this purpouse.

- In the client class enter the code:

```
public static void main(String[] args) throws Throwable {
  // create configuration
  var config = new XmlRpcClientConfigImpl();
  config.setServerURL(new URL("http://127.0.0.1:8080/xmlrpc"));
  config.setEnabledForExtensions(true);
  config.setConnectionTimeout(60 * 1000);
  config.setReplyTimeout(60 * 1000);

  // create client and set settings
  var client = new XmlRpcClient();
  client.setTransportFactory(new XmlRpcSunHttpTransportFactory(client));
  client.setConfig(config);

  // synchronous call
  System.out.println("Executing synchronous call...");
  var result = client.execute("MessagesService.echo", List.of("Message One [1]"));
  System.out.println(result);

  // create callbacks
  var callback1 = new TimingOutCallback(1000);
  var callback2 = new TimingOutCallback(5 * 1000);

  // asynchronous calls
  System.out.println("Executing asynchronous call...");
  client.executeAsync("MessagesService.echoWithDelay", List.of("Message Two [2]"),
                      callback1);

  System.out.println("Executing asynchronous call...");
  client.executeAsync("MessagesService.echoWithDelay",
                      List.of("Message Three [3]"), callback2);

  System.out.println(getAsyncResponse(timeoutCallback));
  System.out.println(getAsyncResponse(correctCallback));
}

// method to wait for responses and print fails
public static Optional<Object> getAsyncResponse(TimingOutCallback callback) {
  try {
    return Optional.ofNullable(callback.waitForResponse());
  } catch (TimingOutCallback.TimeoutException e) {
    logMessage("No response from server before timeout.");
  } catch (Throwable e) {
    logMessage("Server returned an error message.");
  }
  return Optional.empty();
}
```

3.3 Running and testing application

- Run the application and check the operation:
  - run the server first,
  - next run the client

  The server should show similar to the following:

```
2023-03-08T02:30:24.911635Z Server started...
Listening on port: 8080
Press <ENTER> to stop the server.

2023-03-08T02:30:45.331575Z Message received: My message One [1]
2023-03-08T02:30:45.375561500Z Message received: My message Three [3]
Waiting for 3 seconds before sending response...
2023-03-08T02:30:45.375561500Z Message received: My message Two [2]
Waiting for 3 seconds before sending response...

Process finished with exit code 0
```

  The client should show similar to the following:

```
"C:\Program Files\Java\jdk-17\bin\java.exe" …

2023-03-08T02:30:45.241604100Z Executing synchronous call...
2023-03-08T02:30:45.369562900Z Echo: My message One [1]
2023-03-08T02:30:45.369562900Z Executing asynchronous call...
2023-03-08T02:30:45.370562600Z Executing asynchronous call...
2023-03-08T02:30:46.372266400Z No response from server before
timeout.
2023-03-08T02:30:46.372266400Z Optional.empty
2023-03-08T02:30:48.392661700Z Optional[Echo after delay: My message
Three [3]]

Process finished with exit code 0
```