Packaging Java Applications (.jar)

Contents

Overview

Maven allows us to package Java applications into neat containers. These are known as **Java Archives**, and use the **.jar** file extension.

(*note: there are also Web Archives, which are used for Java-based Web-apps (.war files), but we will not be using them in this tutorial.*)

Tutorial

Given that Maven is a build tool, it can be used to build out applications into easily-runnable (*repeatable*) executables. Much like the **.exe** extension, **.jar** files allow for us to run Java applications *platform-independently* - after all, 6 billion devices run Java; not all of them are going to have a compatible IDE installed!

This allows us to run them from a *command-line- interface (CLI)* without needing to open them up inside an IDE.

Let's test out how Maven does this by using a premade project.

Clone this Git repository before continuing with this tutorial.

Open up a command line inside the folder where you cloned this repo.

The aim here is that, when the program is run from a command line, it will give us two messages - a test message, and a message containing a nice fancy random integer.

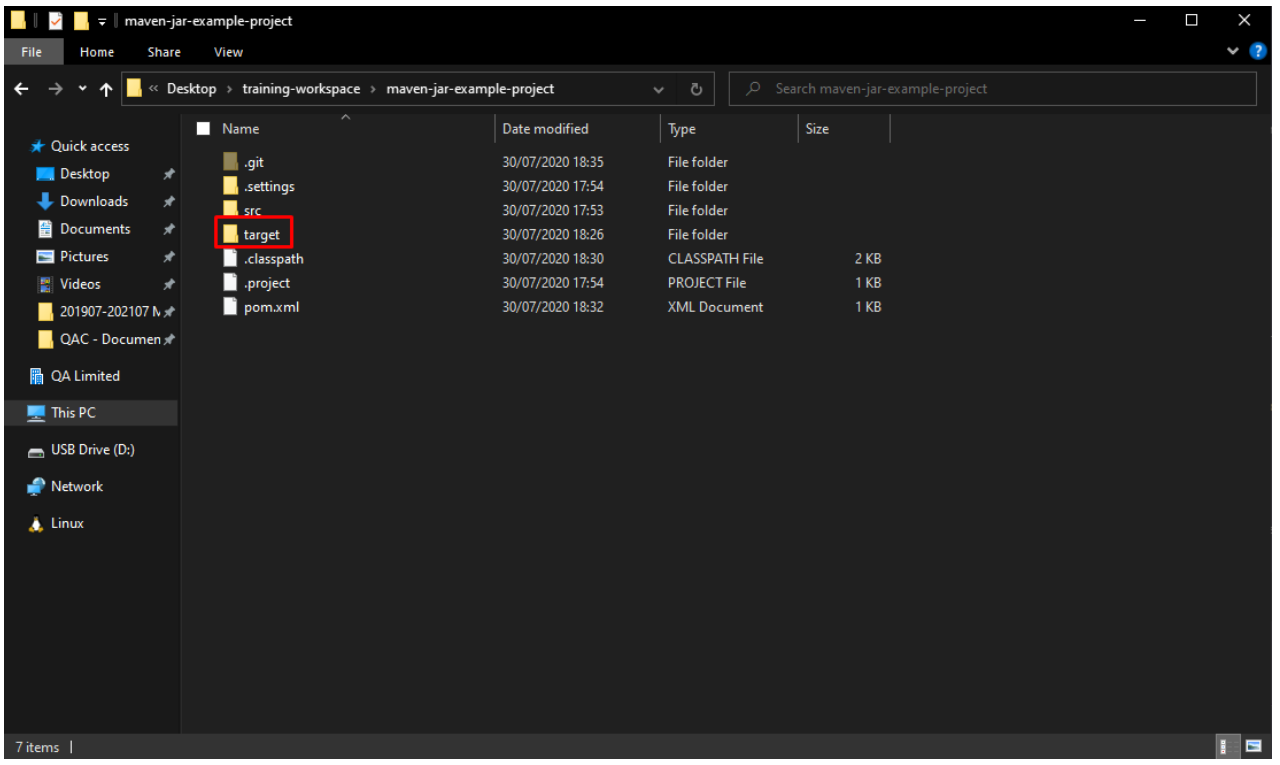
It should look a bit like this:

```
This should be printed no matter what!
This number should be printed if we build a fat-.jar!: 1200303342
```

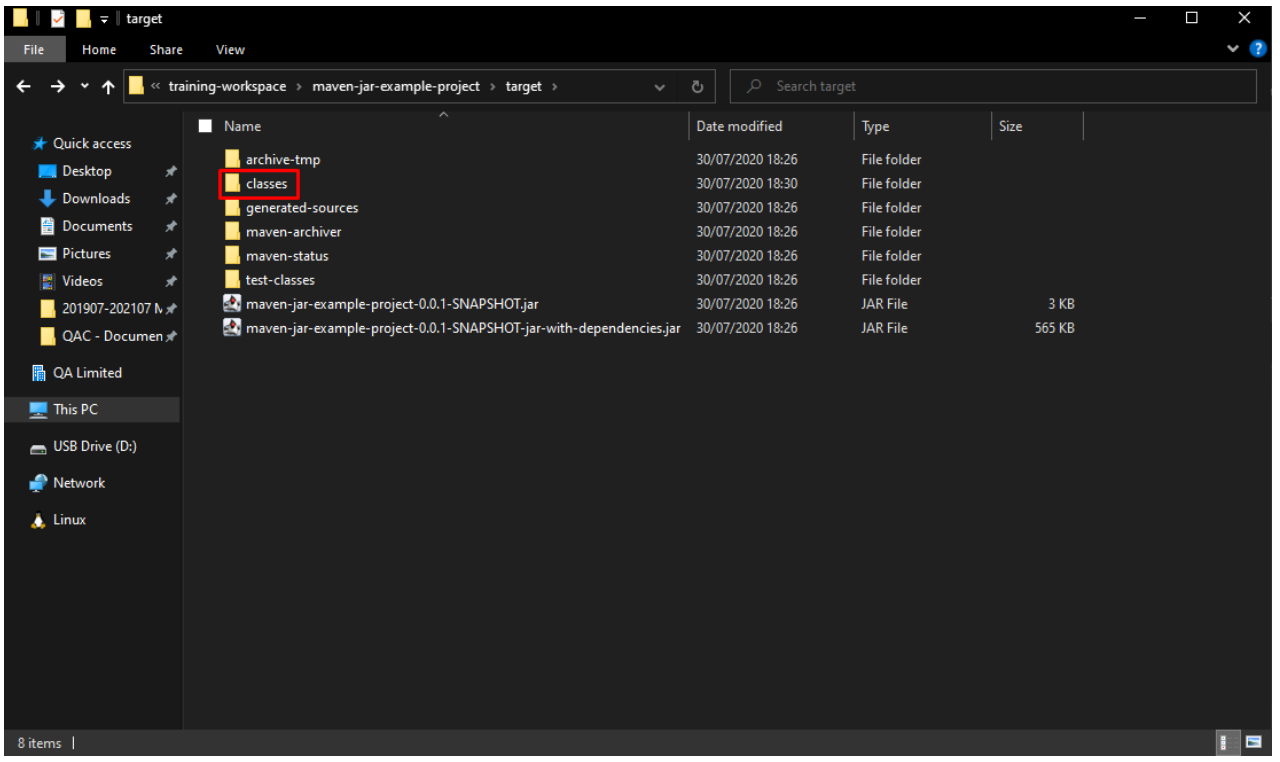
Let's see if we can get it working.

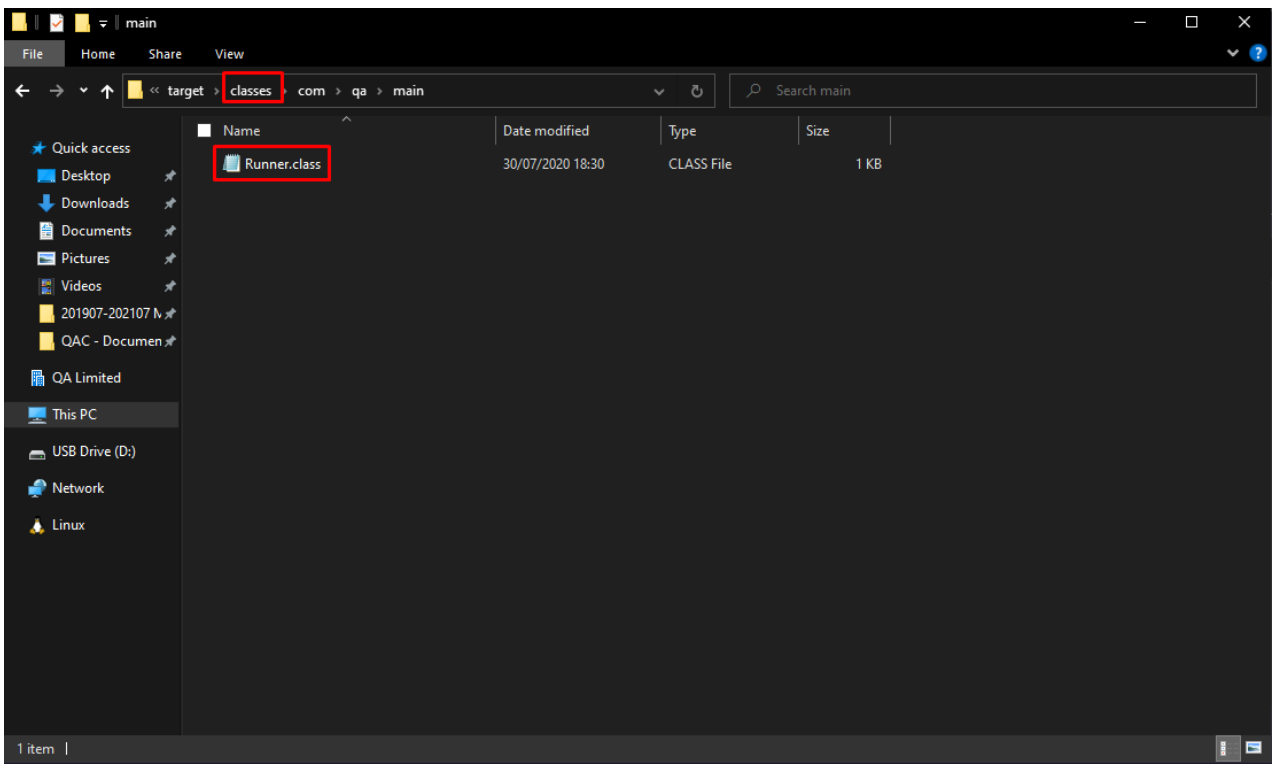
`mvn clean`

When using a Maven project inside Eclipse, you will notice that, when running the application you've made, a `/target/` folder will be created:



This is the Java Virtual Machine (JVM) converting your classes into machine-readable code - it converts your `.java` files into `.class` files and stores them inside that folder:

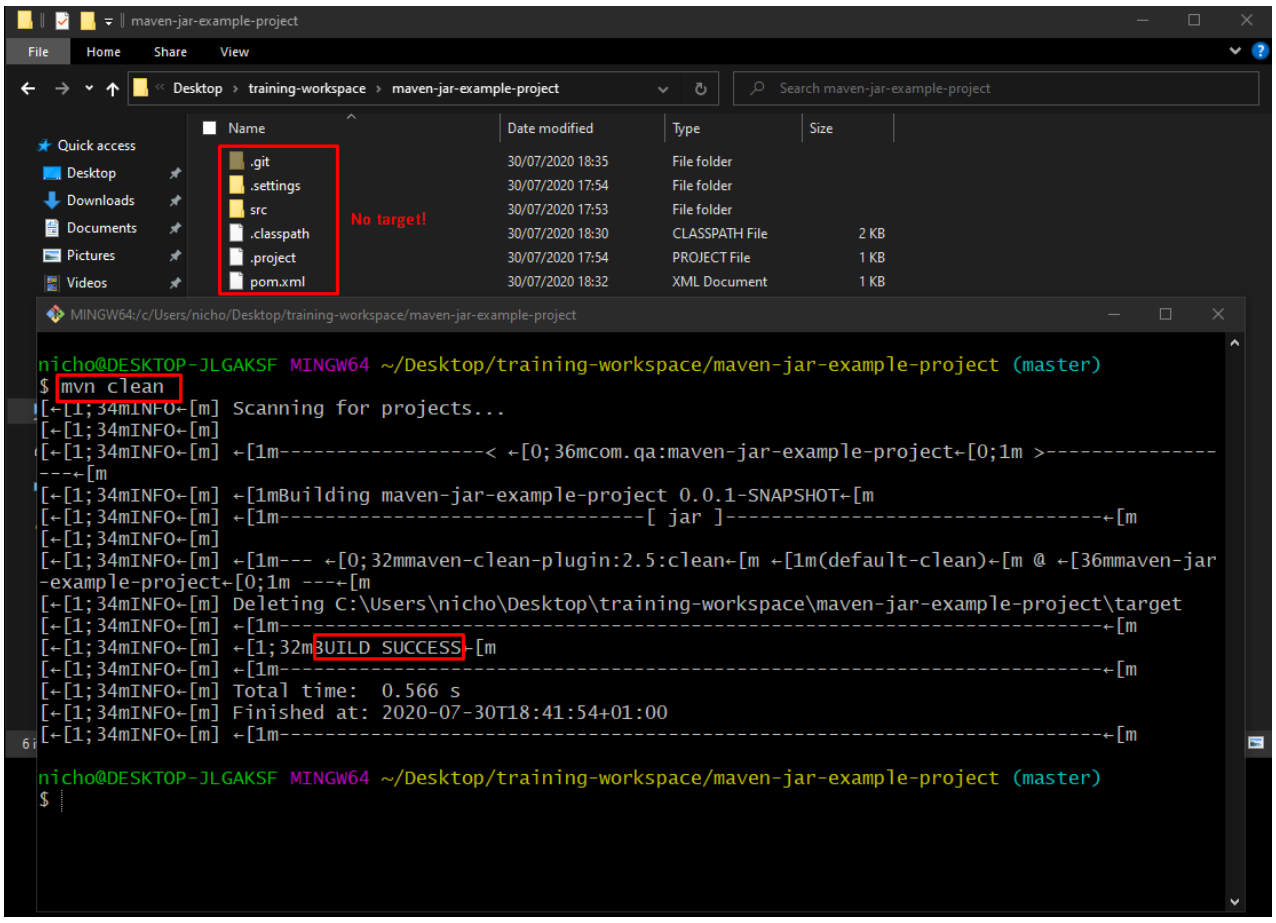




When packaging up an application, that `/target/` folder is where the `.jar` is saved.

If we open a command-line inside the project folder, we can get Maven to clean the project folder of IDE-generated files for us, which deletes the `/target/` folder for us:

```
mvn clean
```



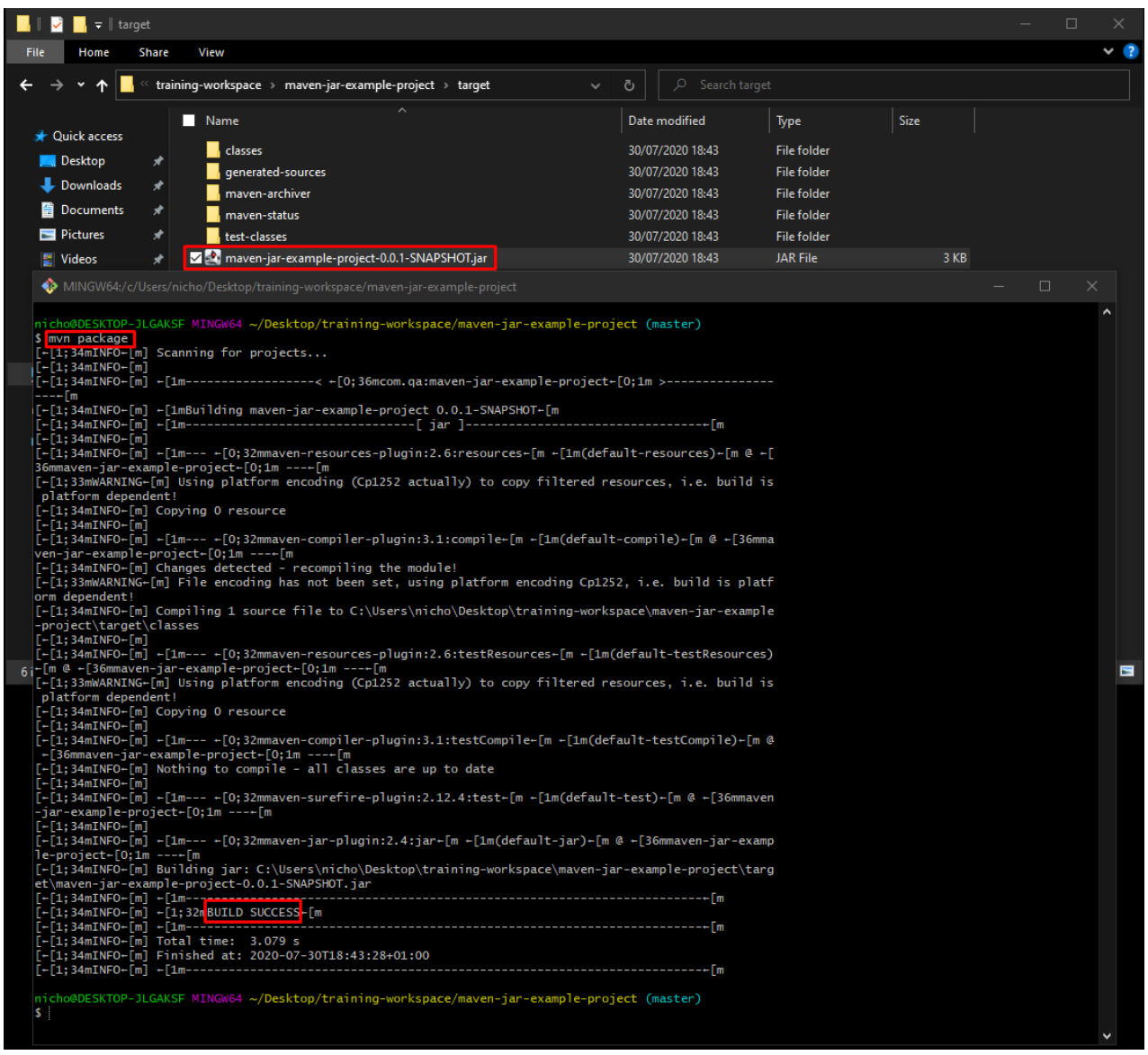
We do this so that we can remove old builds of our code before rebuilding it again. Generally speaking, you should always run an `mvn clean` before building your code.

`mvn package`

To package our application into the platform-independent `.jar`, we run the following command:

```
mvn package
```

This will `package` all the source code in our Java project into the `.jar` file:



We can then try to run it using the following command-line Java command:

```
java -jar maven-jar-example-project-0.0.1-SNAPSHOT.jar
```

It won't work - because we haven't added the dependency to allow Maven to build things into `.jar`s:

```
no main manifest attribute, in maven-jar-example-project-0.0.1-SNAPSHOT.jar
```

Making a `.jar`

We can solve this by using the **`maven-jar-plugin`**. This will build the project into a `.jar` for us.

Update your `pom.xml` with the following code - this should go **after** your `</dependencies>` close-tag:

```
<build>
    <plugins>
        <plugin>
            <groupId>org.apache.maven.plugins</groupId>
            <artifactId>maven-jar-plugin</artifactId>
            <version>3.2.0</version>
            <configuration>
                <archive>
                    <manifest>
                        <mainClass>com.qa.main.Runner</mainClass>
                    </manifest>
                </archive>
            </configuration>
        </plugin>
    </plugins>
</build>
```

This plugin allows us to package all our source code into a `.jar`.

Run `mvn clean package` again, then the `java -jar ...` command again, and we should see this:

```
This should be printed no matter what!
Exception in thread "main" java.lang.NoClassDefFoundError:
org/apache/commons/lang3/RandomUtils
        at com.qa.main.Runner.main(Runner.java:9)
Caused by: java.lang.ClassNotFoundException:
org.apache.commons.lang3.RandomUtils
        at java.base/jdk.internal.loader.BuiltinClassLoader.loadClass(Unknown
Source)
        at
java.base/jdk.internal.loader.ClassLoaders$AppClassLoader.loadClass(Unknown
Source)
        at java.base/java.lang.ClassLoader.loadClass(Unknown Source)
        ... 1 more
```

Well, it works on the first print statement... but it's not running our fancy integer-generating code!

This is because the plugin we've put in our `pom.xml` only packages up our own source code, **not** any external libraries.

This is great for small applications, or ones where you're writing all your own code from scratch, but it won't do for what we've got here.

Building a 'fat-`.jar`'

We need to use the **`maven-assembly-plugin`** instead - this creates a `.jar` containing **both** our source code and any other dependencies it might have.

Add this inside your `<plugins>` tag in your `pom.xml`:

```xml
<plugin>
    <groupId>org.apache.maven.plugins</groupId>
    <artifactId>maven-assembly-plugin</artifactId>
    <executions>
        <execution>
        <phase>package</phase>
        <goals>
            <goal>single</goal>
        </goals>
        <configuration>
            <archive>
                <manifest>
                    <mainClass>com.qa.main.Runner</mainClass>
                </manifest>
            </archive>
            <descriptorRefs>
                <descriptorRef>jar-with-dependencies</descriptorRef>
            </descriptorRefs>
        </configuration>
        </execution>
    </executions>
</plugin>
```

This is slightly different from the `maven-jar-plugin` - instead of only packaging your source code, it'll package up **everything** - including external libraries.

Given how much larger these are from regular `.jar` files, these are known as '**fat-`.jar`s**'.

We can help ourselves to identify which `.jar` is the one which includes the external dependencies by using the `<descriptorRef>` "`jar-with-dependencies`".

Final attempt

We can now build out the application and run it properly.

Handily, Maven allows us to chain its phases together:

```
mvn clean package
java -jar maven-jar-example-project-0.0.1-SNAPSHOT-jar-with-dependencies.jar
```

You should be greeted with the code working as intended:

```
This should be printed no matter what!
This number should be printed if we build a fat-.jar!: 1200303342
```

Exercises

There is no exercise for this module.