UNIVERSITET OF UPPSALA

MDConverter Manual

User & developer manual

Michael Pecher

3/4/2016

This document contains a user manual and also a developer manual for the MDConverter prototype framework. This framework is used to convert structure and topology input data of different molecular dynamic simulation tools. The framework is extensible with plugins to support further tools.

Introduction

MD computer simulation is a method to simulate the interactions and trajectories of atoms and molecules during a pre-defined timespan. To run such simulations, specific input files are needed. These input files differ from one simulation tool to the other, and even if the coordinate and/or the topology of the planned simulation already exists for a simulation tool other than the one which will be used, it needs to be converted. (Nair & Miners, 2014) (Rapaport, 2004)

Coordinate input files contain the structural information about the system which is going to be simulated. This information consists of 3D coordinates of each atom of the system and sometimes the connections between them (Nair & Miners, 2014) (Rapaport, 2004).

Topology input files contain the information about each atom defined in the coordinate input file. This information consists of data about the atom, like for example the charge, the mass and other values of the atoms. Furthermore it contains the bonds between the defined atoms, the manner of calculation of the bonds and also some more information. This information of topology input files are the forces which are to be simulated, also called Force Field (FF). FFs are not compatible between different simulation tools because each tool has to support different calculation algorithms to simulate different kinds of FFs. (Nair & Miners, 2014) (Rapaport, 2004) The main goal of the project is to support an interconversion process of the mentioned input files for different simulation tools, to avoid the great overhead costs for generating these files manually for each simulation tool. The developed prototype framework, called MDConverter, is written in Java and is a pilot project to determine if conversions can be done successfully for these file types. The project should also determine if such a framework could be flexible enough to provide a relatively simple way to write plug-ins for the individually preferred simulation tool and publish it to the community. (Nair & Miners, 2014) (Rapaport, 2004)

The requirements such a solution has to fulfill, are:

1. Many simulation tools uses their own file formats for input data, which is the reason why there is no common format which all simulation tools support. This circumstance necessitates the possibility to attach new file format translators to the framework when required. It is impracticable to be forced to write converters for all existing simulation tools individually and maintain all changes which may occur in the developmental process of the simulation tools. To fulfill this requirement the framework has to be easily extendable with third party plug-ins for different simulation tools in order to be generally applicable. (Abraham et al., 2016) (Berman et al., 2003) (Case et al., 2015) (Sousa da Silva & Vranken, 2012)
2. In order to provide automatic conversions of coordinate input files all kinds of simulation tools and their formats have to fit into a specific data model defined by the framework. The conversion of the topology files faces a similar problem. To convert coordinate and topology-input files, two data models have to be developed. (Abraham et al., 2016) (Berman et al., 2003) (Case et al., 2015) (Sousa da Silva & Vranken, 2012)
3. Each MD simulation tool uses different units of measurement for internal calculations and hence for their input files. The framework has to be able to process these differences and automatically convert these units. (Abraham et al., 2016) (Berman et al., 2003) (Caseet al., 2015) (Sousa da Silva & Vranken, 2012)
4. Different programming languages are another requirement in this scenario because the target groups for this framework are chemists, biologists and physicists. Therefore support of an additional script language would increase the acceptance of the framework. (Abraham et al., 2016) (Berman et al., 2003) (Case et al., 2015) (Sousa da Silva & Vranken, 2012)

Vranken, 2012) User Manual

This part of the manual describes how to use the framework and which possibilities are available to configure the framework.

What does the framework do?

The MDConverter framework is an open source tool, which provides the possibility to convert molecular dynamic simulation input files. The framework supports two different file types, first structure files and second topology files. Depending on the available plugins the framework is able to convert between different simulation tools or formats.

The basic framework supports:

1) Structure data:
    a. Read:
        i. Amber (*.mol2)
        ii. Gromacs (*.gro)
    b. Write:
        i. PDB-format (*.pdb)
2) Topology data:
    a. Read:
        i. Amber (*.prmtop)
        ii. Gromacs (*.top)
    b. Write:
        i. Gromacs (*.top)

The framework only converts within the two file types, which means that it is only possible to convert into another format of structure files and not from structure files to topology files.

How to add a new plugin to my local installation?

Depending on the installation path in your system, there are four folders parallel to the "MDConverter-x.x.jar" file.

1) StructureReader
2) StructureWriter
3) TopologyReader
4) TopologyWriter

The plugin you want to add is designed for you of these four tasks and depending on that you have to copy the given plugin (it should be a jar-File) into the designated folder.

If you are not sure which kind of plugin you should use, try to open the jar-File with a compression program like "WinRaR" or a similar one and look for the "manifest.json" file to see the configuration of the plugin.

How to use the Framework?

The current version 0.1 has the following available options to control the framework.

Usage:

java –jar MDConverter-[version].jar [options]

Unspecified options (unspecified options will stop the application)

Available options:

Option	Description	Default values	Required
--help, -h, -?	*Show usage*	*false*	*No*
-R	*FileReader you want to use*	*None*	*No*
-W	*FileWriter you want to use*	*None*	*No*
-r	*Dynamic parameters for the FileReader* *Syntax: -rkey:value (defined by plugin)*	*{}*	*No*
-w	*Dynamic parameters for the FileWriter* *Syntax: -wkey:value (defined by plugin)*	*{}*	*No*
-S	*Define for structure type conversion* *only required if topology type is not defined*	*None*	*No*
-T	*Define for topology type conversion* *only required if structure type is not defined*	*None*	*No*
-i, -in	*Input file you want to convert*	*None*	*Yes*
-o, -out	*Output goes to this file* *if not defined, output goes to console*	*None*	*No*

How to show plugin usage?

To get the usage of available plugins to see configuration possibilities you have to define the "-h" option and run the application. Then the framework will show the general usage and forces you to select the file type and afterwards the reader and writer plugin. After that, the framework will show the usage of the selected reader and writer plugin.

If no input file is defined the framework will stop because of a missing option. If everything is well configured the framework will go on with the conversion process.

Selecting plugins during runtime?

If you do not know which plugins are available just run the application without defining them and you will be forced to select the plugins from a list depending on the file type. If no file type is defined you will be asked for a file type too.

Developer Manual

The developer manual explains how to write your own plugins for the MDConverter framework.

How to configure my plugin?

Each plugin has to have a "manifest.json" file which will be loaded by the PluginLoader and will be parsed to a PluginManifest. The manifest contains a general configuration section and a section to define the measurement units for the plugin.

Description of available parameters:

Key	Description
pluginName	Display name for your plugin (identifier for command line option too)
className	The main class which implements AbstractReader or AbstractWriter
version	The version of your plugin (independent from framework)
fileExtension	The file extension which is supported by your plugin
fileType	Which kind of model will be used ("structure" or "topology")
pluginType	Either a "reader" or a "writer" plugin, depending on implemented abstract class
script	Uses a script language instead of Java ("true" or "false")
scriptType	"none" or "jython"
modelVersion	Which topology ModelVersion is supported, only necessary if fileType = "topology" Framework checks if ReaderPlugin and WriterPlugin are compatible → cancel if the ModelVersion of the reader is lower than the one of the writer or inform the user about possible data loss if not
measurementUnits	Defines the necessary units to convert the models from ReaderPlugin to the WriterPlugin The complete unit configuration has to look as in the example except for the values (multiplication in units must be written as a dot "·") For unit definition see JScience JavaDoc.

Special cases:

- "global" are general unit definitions for the simulation tool and thought as a single line replacement in future releases as for ex. ("atom/atomTypes": { "none": { "c1": "charge"....) and charge will be replaced with the global definition. So, please define them. At the moment only the length unit is used by the framework for the Structure model but not as a replacement.
- Sometimes something like for ex. ("c1_1": "kJ·nm^6/mol", "c2_1": "kJ·nm^12/mol","c1": "nm", "c2": "kJ/mol") exists. c1 is the default value and c1_1 is a replacement if the topology meta model class "Default.class" has set "combRule" to 1. In this situation the framework will use the alternative unit definition from c1_1.
- At the moment it is not possible to define custom units (units which are not standard in the JScience API). Custom definitions is a planned feature for future releases. The following special units are defined (see abstract class "Convert" for details).

```
private final double cal = 4.1868;
UnitFormat.getInstance().label(NonSI.E.divide(18.222615), "efact");
UnitFormat.getInstance().label(KILO(JOULE.times(cal)), "kcal");
UnitFormat.getInstance().label(JOULE.times(cal), "cal");
```

- Last but not least, sometimes you need values from the runtime object. For that purpose you can do something like ("orientationRestraint": { "1": { "c1": "U·nm^'getAlpha' "....), where 'getAlpha' returns the alpha value from the runtime object (apostrophes are the indicator for a method name

which will be called via reflection). If the actual object of OrientationRestraintImpl with a funcType set to 1 and alpha set to 10 will be converted by the framework, value c1 will be converted with a unit depending on the actual object. This means the given example will end in U·nm^10

This is a full example of a configuration (just copy part 1 and the part 2and adopt the values to your needs):

Part 1

```
{
 "pluginName": "gromacs",
 "className":
"org.mdconverter.gromacstr.main.GromacsTReader",
 "version": "1.0",
 "fileExtension": "top",
 "fileType": "topology",
 "pluginType": "reader",
 "script": false,
 "scriptType": "none",
 "modelVersion": "V1",
 "measurementUnits": {
  "global": {
   "standard": {
    "length": "nm",
    "mass": "u",
    "time": "ps",
    "charge": "e",
    "temperature": "K",
    "energy": "kJ/mol",
    "force": "kJ·nm/mol",
    "pressure": "kJ·nm^3/mol",
    "velocity": "nm/ps",
    "dipolemoment": "e·nm"
   }
  },
  "atom/atomTypes": {
   "none": {
    "c1": "e",
    "c2": "u",
    "c3_1": "kJ·nm^6/mol",
    "c4_1": "kJ·nm^12/mol",
    "c3": "nm",
    "c4": "kJ/mol"
   }
  },
  "bond/bondTypes": {
   "1": {
    "c1": "nm",
    "c2": "kJ·nm^-2/mol"
   },
   "2": {
    "c1": "nm",
    "c2": "kJ·nm^-4/mol4"
   },
   "3": {
    "c1": "nm",
    "c2": "kJ/mol",
    "c3": "1/nm"
   },
   "4": {
    "c1": "nm",
    "c2": "kJ·nm^-2/mol",
    "c3": "kJ·nm^-3/mol"
   },
   "6": {
    "c1": "nm",
    "c2": "kJ·nm^-2/mol"
   },
   "7": {
```

Part 2

```
   "dihedral/dihedralTypes": {
    "1": {
     "c1": "°",
     "c2": "kJ/mol"
    },
    "2": {
     "c1": "°",
     "c2": "kJ·rad^-2/mol"
    },
    "3": {
     "c1": "kJ/mol",
     "c2": "kJ/mol",
     "c3": "kJ/mol",
     "c4": "kJ/mol",
     "c5": "kJ/mol",
     "c6": "kJ/mol"
    },
    "4": {
     "c1": "°",
     "c2": "kJ/mol"
    },
    "5": {
     "c1": "kJ/mol",
     "c2": "kJ/mol",
     "c3": "kJ/mol",
     "c4": "kJ/mol"
    },
    "8": {
     "c2": "kJ/mol"
    },
    "9": {
     "c1": "°",
     "c2": "kJ/mol"
    },
    "10": {
     "c1": "°",
     "c2": "kJ/mol"
    },
    "11": {
     "c1": "kJ/mol",
     "c2": "kJ/mol",
     "c3": "kJ/mol",
     "c4": "kJ/mol",
     "c5": "kJ/mol"
    }
   },
   "constraint/constraintTypes": {
    "1": {
     "c1": "nm"
    },
    "2": {
     "c1": "nm"
    }
   },
   "nonbond_params": {
    "1": {
     "c1_1": "kJ·nm^6/mol",
     "c2_1": "kJ·nm^12/mol",
     "c1": "nm",
     "c2": "kJ/mol"
```

```
      "c1": "nm",
      "c2": "kJ·nm^-2/mol"
     },
     "8": {
      "c2": "kJ/mol"
     },
     "9": {
      "c2": "kJ/mol"
     },
     "10": {
      "c1": "nm",
      "c2": "nm",
      "c3": "nm",
      "c4": "kJ·nm^-2/mol"
     }
    },
    "pair/pairTypes": {
     "1": {
      "c1_1": "kJ·nm^6/mol",
      "c2_1": "kJ·nm^12/mol",
      "c1": "nm",
      "c2": "kJ/mol"
     },
     "2": {
      "c2": "e",
      "c3": "e",
      "c4_1": "kJ·nm^6/mol",
      "c5_1": "kJ·nm^12/mol",
      "c4": "nm",
      "c5": "kJ/mol"
     }
    },
    "angle/angleTypes": {
     "1": {
      "c1": "°",
      "c2": "kJ·rad^-2/mol"
     },
     "2": {
      "c1": "°",
      "c2": "kJ/mol"
     },
     "3": {
      "c1": "nm",
      "c2": "nm",
      "c3": "kJ·nm^-2/mol"
     },
     "4": {
      "c1": "nm",
      "c2": "nm",
      "c3": "nm",
      "c4": "kJ·nm^-2/mol"
     },
     "5": {
      "c1": "°",
      "c2": "kJ·rad^-2/mol",
      "c3": "nm",
      "c4": "kJ·nm^-2/mol"
     },
     "6": {
      "c1": "°",
      "c2": "kJ·rad^-0/mol",
      "c3": "kJ·rad^-1/mol",
      "c4": "kJ·rad^-2/mol",
      "c5": "kJ·rad^-3/mol",
      "c6": "kJ·rad^-4/mol"
     },
     "8": {
      "c2": "kJ/mol"
     },
     "10": {
      "c1": "°",
      "c2": "kJ/mol"
     },
     "2": {
      "c1": "kJ/mol",
      "c2": "1/nm",
      "c3": "kJ·nm^6/mol"
     }
    },
    "settle": {
     "1": {
      "c1": "nm",
      "c2": "nm"
     }
    },
    "positionRestraint": {
     "1": {
      "c1": "kJ·nm^-2/mol",
      "c2": "kJ·nm^-2/mol",
      "c3": "kJ·nm^-2/mol"
     },
     "2": {
      "c1": "nm",
      "c2": "nm",
      "c3": "kJl·nm^-2/mo"
     }
    },
    "distanceRestraint": {
     "1": {
      "c1": "nm",
      "c2": "nm",
      "c3": "nm"
     }
    },
    "dihedralRestraint": {
     "1": {
      "c1": "°",
      "c2": "°"
     }
    },
    "orientationRestraint": {
     "1": {
      "c1": "U·nm^'getAlpha'",
      "c2": "U",
      "c3": "1/U"
     }
    },
    "angleRestraint": {
     "1": {
      "c1": "°",
      "c2": "kJ/mol"
     }
    },
    "angleRestraintZ": {
     "1": {
      "c1": "°",
      "c2": "kJ/mol"
     }
    },
    "pairNB": {
     "1": {
      "c1": "e",
      "c2": "e",
      "c3_1": "kJ·nm^6/mol",
      "c4_1": "kJ·nm^12/mol",
      "c3": "nm",
      "c4": "kJ/mol"
     }
    }
   }
}
```

} },	

How to write a new Plugin?

The Framework is a Maven project and provides following libraries for the plugins:

Library	Version
com.google.guava:guava	19.0
org.jscience:jscience	4.3.1
com.beust:jcommander	1.48
com.google.inject:guice	4.0
com.google.code.gson:gson	2.6.2
org.slf4j:slf4j-api	1.7.12
org.biojava:biojava-structure	4.1.0
org.python:jython-standalone	2.7.1b2
org.apache.commons:commons-lang3	3.4
commons-io:commons-io	2.4

If your plugin needs one of these dependencies just do something like:

<dependency>
<groupId>org.biojava</groupId>
<artifactId>biojava-structure</artifactId>
<version>4.1.0</version> (only needed if you have a standalone maven project)
<scope>provided</scope>
</dependency>

PS.: This dependency is needed if you are writing a Reader-/WriterPlugin with FileType “structure”.

Write it as a standalone maven project:

The Following maven configuration is needed to generate a jar-File of your plugin:

```
<?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>

    <groupId>org.mdconverter</groupId>
    <artifactId>GromacsTopologyReader</artifactId> <!– customize that -->
    <version>1.0-SNAPSHOT</version> <!– customize that -->

    <dependencies> <!– customize that -->
        <dependency>
            <groupId>org.mdconverter</groupId>
            <artifactId>MDConverterAPI</artifactId>
            <version>0.1</version>
            <scope>provided</scope>
        </dependency>
    </dependencies>

    <properties>
        <build.dir>${project.basedir}/../build</build.dir>
        <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
        <project.reporting.outputEncoding>
                UTF-8
        </project.reporting.outputEncoding>
    </properties>
```

```
    <build>
        <plugins>
            <plugin>
                <groupId>org.apache.maven.plugins</groupId>
                <artifactId>maven-compiler-plugin</artifactId>
                <version>2.3.2</version>
                <configuration>
                    <source>1.8</source>
                    <target>1.8</target>
                </configuration>
            </plugin>
            <plugin>
                <groupId>org.apache.maven.plugins</groupId>
                <artifactId>maven-jar-plugin</artifactId>
                <version>2.3.1</version>
                <configuration>
                    <outputDirectory>${build.dir}</outputDirectory>
                </configuration>
            </plugin>
            <plugin>
                <groupId>org.apache.maven.plugins</groupId>
                <artifactId>maven-shade-plugin</artifactId>
                <version>2.3</version>
                <executions>
                    <execution>
                        <phase>package</phase>
                        <goals>
                            <goal>shade</goal>
                        </goals>
                    </execution>
                </executions>
                <configuration>
                    <finalName>GromacsTR</finalName> <!— customize that -->
                    <outputDirectory>
                       ${build.dir}/TopologyReader <!— customize that -->
                    </outputDirectory>
                </configuration>
            </plugin>
        </plugins>
        <directory>target</directory>
        <resources>
            <resource>
                <directory>src/main/resources</directory>
                <filtering>true</filtering>
            </resource>
        </resources>
    </build>
</project>
```

Write it with the whole MDConverter maven project:

```
<?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">
    <parent>
        <artifactId>MDConverterCollection</artifactId>
        <groupId>org.mdconverter</groupId>
        <version>0.1</version>
```

```
    </parent>
    <modelVersion>4.0.0</modelVersion>

    <artifactId>GromacsStructureReader</artifactId> <!– customize that -->

    <dependencies> <!– customize that -->
        <dependency>
            <groupId>org.mdconverter</groupId>
            <artifactId>MDConverterAPI</artifactId>
            <scope>provided</scope>
        </dependency>
    </dependencies>

    <build>
        <plugins>
            <plugin>
                <groupId>org.apache.maven.plugins</groupId>
                <artifactId>maven-compiler-plugin</artifactId>
                <version>2.3.2</version>
                <configuration>
                    <source>1.8</source>
                    <target>1.8</target>
                </configuration>
            </plugin>
            <plugin>
                <groupId>org.apache.maven.plugins</groupId>
                <artifactId>maven-shade-plugin</artifactId>
                <version>2.3</version>
                <executions>
                    <execution>
                        <phase>package</phase>
                        <goals>
                            <goal>shade</goal>
                        </goals>
                    </execution>
                </executions>
                <configuration>
                    <minimizeJar>true</minimizeJar>
                    <finalName>GromacsSR</finalName> <!– customize that -->
                    <outputDirectory>
                       ${plugin.output.reader.structure}
                    </outputDirectory>
                </configuration>
            </plugin>
        </plugins>
    </build>
</project>
```

How to write a Java plugin?

Follow:

- How to configure my plugin?
- How to write a new Plugin?

and the last thing you have to regard is depending on your configuration to implement either the AbstractReader or AbstractWriter class and override the following methods:

- Reader
    - getMetaModel()
    - getDescription()

    - getUsage()
- Writer
    - getOutput()
    - getDescription()
    - getUsage()

How to write a Python plugin?

```
<?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">
    <parent>
        <artifactId>MDConverterCollection</artifactId>
        <groupId>org.mdconverter</groupId>
        <version>0.1</version>
    </parent>
    <modelVersion>4.0.0</modelVersion>

    <artifactId>AmberStructureReader</artifactId> <!— customize that -->

    <dependencies> <!— customize that -->
        <dependency>
            <groupId>org.mdconverter</groupId>
            <artifactId>MDConverterAPI</artifactId>
            <version>0.1</version>
        </dependency>
    </dependencies>

    <build>
        <plugins>
            <plugin>
                <groupId>org.apache.maven.plugins</groupId>
                <artifactId>maven-compiler-plugin</artifactId>
                <version>2.3.2</version>
                <configuration>
                    <source>1.8</source>
                    <target>1.8</target>
                </configuration>
            </plugin>
            <plugin>
                <groupId>org.apache.maven.plugins</groupId>
                <artifactId>maven-source-plugin</artifactId>
                <version>2.1.2</version>
            </plugin>
            <plugin>
                <groupId>net.sf.mavenjython</groupId>
                <artifactId>jython-compile-maven-plugin</artifactId>
                <version>1.2</version>
                <executions>
                    <execution>
                        <phase>package</phase>
                        <goals>
                            <goal>jython</goal>
                        </goals>
                    </execution>
                </executions>
            </plugin>
            <plugin>
                <groupId>org.apache.maven.plugins</groupId>
                <artifactId>maven-shade-plugin</artifactId>
```

```
                <version>2.3</version>
                <executions>
                    <execution>
                        <phase>package</phase>
                        <goals>
                            <goal>shade</goal>
                        </goals>
                    </execution>
                </executions>
                <configuration>
                    <minimizeJar>true</minimizeJar>
                    <finalName>AmberSR</finalName> <!— customize that -->
                    <outputDirectory>
                       ${plugin.output.reader.structure}
                    </outputDirectory>
                </configuration>
            </plugin>
        </plugins>
    </build>
</project>
```

The example Maven configuration is for a plugin which relies on the complete framework. To write a standalone Python plugin, combine the maven configuration with that from “How to write a new Plugin? - Write it as a standalone maven project”.

A small part has to be written in Java because the Python script will be compiled by a library called “Jython”. This library generates Java code out of a Python script and allows the usage of Java classes in Python. For this purpose you have to follow “How to write a Java plugin?” until you start with the implementation of “getMeatModel()” or “getOutput()”, there you have to create an Interface which will be implemented by a Python script.

Here a small example for a structure reader:

```
public interface ReadStructure {

    Structure readFileToStructure(String file, Structure structure);
}
```

This Interface has to be implemented by Python script as below.

```
import org.mdconverter.api.ReadStructure as ReadStructure
import java.lang.Object as Object
import org.mdconverter.api.plugin.InvalidInputException as Invalid

class ReadFile(ReadStructure, Object):
        your code!!!!!
```

After you have finished your implementation you have to add the following code snippet into your “getMeatModel()” or “getOutput()” implementation depending on your plugin and adopt it to your package hierarchy and naming.

```
//get the object factory from the framework
JythonObjectFactory jof = getJythonObjectFactory();
//generate a Java implementation out of the Python script which implements
the defined interface ReadStructure.class
ReadStructure readFile = (ReadStructure)
jof.createObject(ReadStructure.class, "ReadFile", "Lib.processfile.");
```

```
//get the Structure from the Java implementation
Structure structure =
readFile.readFileToStructure(getInputFile().toString(), (Structure)
getStructure());
//inform the user about progress
getConsoleWriter().printInfoln(structure.getName() + " successfully read
in");
return structure;
```

The Python scripts have to be placed in the resources of the Java project structure. In this example, the hierarchy is Lib.processfile.ReadFile.py. Each of the subfolders has to include a __init__.py file which defines the existing package hierarchy.

In the example Lib contains a file with

```
import Lib
import Lib.processfile
```

and processfile has a file with the following code

```
import Lib.processfile.ReadFile

__all__ = ['ReadFile']
```

How to distribute my plugin?

To distribute your plugin for the community you have to provide the compiled jar-File via a file sharing platform. Maybe there will be a centralized platform in the future where you will be allowed to distribute your plugins.

References

Abraham, M., van der Spoel, D., Lindahl, E., Hess, B., & the GROMACS development team, 2016. GROMACS User Manual version 5.1.2: www.gromacs.org.

Berman, H., Henrick, K. & Nakamura, H., 2003. Announcing the worldwide protein data bank. Nature Structural & Molecular Biology, 10, p.980.

Case, D., Berryman, J., Betz, R., Cerutti, D., Cheatham, T., Darden, T., Duke, R., Giese, T., Gohlke, H., Goetz, A., Homeyer, N., Izadi, S., Janowski, P., Kaus, J., Kovalenko, A., LeGrand, T. L. S., Li, P., Luchko, T., Luo, R., Madej, B., Merz, K., Monard, G., Needham, P., Nguyen, H., Nguyen, H., Omelyan, I., Onufriev, A., Roe, D., Roitberg, A., Salomon-Ferrer, R., Simmerling, C., Smith, W., Swails, J., Walker, R., Wang, J., Wolf, R., Wu, X., York, D. & Kollman, P., 2015. AMBER 2015. University of California, San Francisco: University of California.

Nair, P. C. & Miners, J. O., 2014. Molecular dynamics simulations: from structure function relationships to drug discovery. In Silico Pharmacology, 2:4.

Rapaport, D. C., 2004. The Art of Molecular Dynamics Simulation: Cambridge University Press.

Sousa da Silva, A. W. & Vranken, W. F., 2012. Acpype - antechamber python parser interface.BMC Research Notes, 5(1), pp.1–8.