w6_build

Build Tools

Part 1: Building C code from source

Extract it with `tar zxvf FILENAME`. The `z` means decompress it with `gunzip` first, `x` is for extract, `v` is for verbose (list the files it is expanding) and `f` is for read the archive from a file as opposed to standard input.

You can see a file called `INSTALL` which you can open in a text editor to find the standard instructions:

> Briefly, the shell commands ./configure; make; make install should configure, build, and install this package.

Installing

If you want to, you can now type `make install` to copy the executable to `/usr/local/bin`; but you'll probably find it fails with permissions issues. Turns out
most user's aren't allowed to install software for everyone.

- You could install using the `root` user who is allowed to install to the system directories: `sudo make install`
- You could install it somewhere else. Clean out your build and try rerunning `./configure` with `-prefix=${HOME}/.local/` first!

What to do if it doesn't build?

What do you do if it says it can't find a `.h` file, or can't link it to a library file (a `.so`)? C predates modern languages with package managers, so it probably means you haven't installed a library the code depends on. Luckily `apt-file` on Debian-based systems can be really helpful here: run `apt-file search <name of file>` to find out which package provides the file you're missing and install it.

Makefile

```
CC?=gcc
CFLAGS?=-Wall

.DEFAULT:
        echo "default"
rm:
        rm -rf driver library.o
driver: driver.c library.o
        ${CC} ${CFLAGS} driver.c library.o
library.o: library.c
        ${CC} ${CFLAGS} -c library.c -o library.o
install: driver
        install -m 0755 -o root -g root -s driver "$PREFIX/driver"
```

CC as a variable and shift to another compiler:

```
$ make CC=clang driver
```

Patternrule:

```
CC?=gcc
CFLAGS?=-Wall

.DEFAULT:
        echo "default"
%.o: %.c
        ${CC} ${CFLAGS} -c -$^ -o $@
%: %.c %.o
        ${CC} ${CFLAGS} -$^ -o $@

library.o: library.c
driver: driver.c library.o
```

Python

The Python programming language comes with a package manager called `pip`. Find the package that provides it and install it (**hint**: how did we find a missing library in the C build tools?). We are going to practice installing the mistletoe module, which renders markdown into HTML.

- In python, try the line `import mistletoe` and notice that you get `ModuleNotFoundError: No module named 'mistletoe'`.
- Quit python again (Control-d) and try `pip3 install --user mistletoe`. You should get a success message (and possibly a warning, explained below).
- Open python again and repeat `import mistletoe`. This produces no output, so the module was loaded.

```
$ python3
Python 3.11.2 (main, Mar 13 2023, 12:18:29) [GCC 12.2.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> import mistletoe
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ModuleNotFoundError: No module named 'mistletoe'

$ sudo apt install python3-pip

$ pip3 install --user mistletoe
error: externally-managed-environment

× This environment is externally managed
╰─> To install Python packages system-wide, try apt install
    python3-xyz, where xyz is the package you are trying to
    install.

    If you wish to install a non-Debian-packaged Python package,
    create a virtual environment using python3 -m venv path/to/venv.
    Then use path/to/venv/bin/python and path/to/venv/bin/pip. Make
    sure you have python3-full installed.

    If you wish to install a non-Debian packaged Python application,
    it may be easiest to use pipx install xyz, which will manage a
    virtual environment for you. Make sure you have pipx installed.

    See /usr/share/doc/python3.11/README.venv for more information.

note: If you believe this is a mistake, please contact your Python installation or OS distribution
provider. You can override this, at the risk of breaking your Python installation or OS, by passin
g --break-system-packages.
hint: See PEP 668 for the detailed specification.

$ pip3 install --user mistletoe --break-system-packages
Collecting mistletoe
  Downloading mistletoe-1.4.0-py3-none-any.whl (51 kB)
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 51.3/51.3 kB 4.7 MB/s eta 0:00:00
Installing collected packages: mistletoe
  WARNING: The script mistletoe is installed in '/home/vagrant/.local/bin' which is not on PATH.
  Consider adding this directory to PATH or, if you prefer to suppress this warning, use --no-warn
-script-location.
Successfully installed mistletoe-1.4.0

$ python3
Python 3.11.2 (main, Sep 14 2024, 03:00:30) [GCC 12.2.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> import mistletoe
>>>
```

Create a small sample markdown file as follows, called `hello.md` for example:

```
# Markdown Example

Markdown is a *markup* language.
```

Open python again and type the following. You need to indent the last line (four spaces is usual) and press ENTER twice at the end.

```
import mistletoe
with open('hello.md', 'r') as file:
    mistletoe.markdown(file)
```

This should print the markdown rendered to HTML, e.g.

```
<h1>Markdown Example</h1>\n<p>Markdown is a <em>markup</em> language.</p>
```

Sometimes you'll see things telling you to install a package with `sudo pip install` but don't do that! It will break things horribly eventually. You can use pip without sudo, by passing the `--user` option which installs packages into a folder in your home directory (`~/.local`) instead of in `/usr` which normally requires root permissions.

Python used to manage your OS should be run by the system designers; Python used for your dev work should be managed by you. And never the twain shall meet.

Avoiding sudo

Python comes with a mechanism called venv which lets you create a virtual python install that is owned by a user: you can alter the libraries in that without `sudo` and without fear of mucking up your host system. Read the docs and get used to using it—it'll save you a world of pain later!

```
vagrant@debian12:~$ sudo apt install python3-venv -y
vagrant@debian12:~$ python3 -m venv myenv
vagrant@debian12:~$ source myenv/bin/activate
(myenv) vagrant@debian12:~$ pip install --user mistletoe
ERROR: Can not perform a '--user' install. User site-packages are not visible in this virtualenv.
(myenv) vagrant@debian12:~$ pip install mistletoe
Collecting mistletoe
  Using cached mistletoe-1.4.0-py3-none-any.whl (51 kB)
Installing collected packages: mistletoe
Successfully installed mistletoe-1.4.0
```

- `pip freeze | tee requirements.txt` will list all the packages your using and what version they are and save them in a file called `requirements.txt`.
- `pip install -r requirements.txt` will install them again!

  ```
  (myenv) vagrant@debian12:~$ pip freeze
  mistletoe==1.4.0
  (myenv) vagrant@debian12:~$ pip freeze | tee requirements.txt
  mistletoe==1.4.0
  (myenv) vagrant@debian12:~$ pip install -r requirements.txt
  Requirement already satisfied: mistletoe==1.4.0 in ./myenv/lib/python3.11/site-packages (from -r r
  equirements.txt (line 1)) (1.4.0)
  ```

This makes it *super easy* to ensure that someone looking at your code has all the right dependencies without having to reel off a list of *go install these libraries* (and will make anyone whoever has to mark your code happy and more inclined to give you marks).

Java

In the Java world,

- The `javac` compiler turns source files (`.java`) into `.class` files;
- The `jar` tool packs class files into `.jar` files;
- The `java` command runs class files or jar files.

A Java Runtime Environment (JRE) contains only the `java` command, which is all you need to run java applications if you don't want to do any development. Many operating systems allow you to double-click jar files (at least ones containing a special file called a `manifest`) to run them in a JRE. A Java Development Kit (JDK) contains the `javac` and `jar` tools as well as a JRE. This is what you need to develop in java. `maven` is a Java package manager and build tool. It is not part of the Java distribution, so you will need to install it separately.

Installing on Debian

On Debian, install the `openjdk-17-jdk` and `maven` packages. This should set things up so you're ready to go but if you have *multiple versions* of Java installed you may need to set the `JAVA_HOME` and `PATH` variables to point to your install.

For example:

```
export JAVA_HOME='/usr/lib/jvm/java-17-openjdk'export PATH="${PATH}:${JAVA_HOME}/bin"
```

|||advanced Debian also has a special command called `update-alternatives` that can help manage alternative development environments for you. Read the manual page! |||

```
vagrant@debian12:~$ sudo apt install openjdk-17-jdk
vagrant@debian12:~$ echo $JAVA_HOME

vagrant@debian12:~$ echo $PATH
/home/vagrant/.local/bin:/usr/local/bin:/usr/bin:/bin:/usr/local/games:/usr/games

vagrant@debian12:~$ java -version
openjdk version "17.0.13" 2024-10-15
OpenJDK Runtime Environment (build 17.0.13+11-Debian-2deb12u1)
OpenJDK 64-Bit Server VM (build 17.0.13+11-Debian-2deb12u1, mixed mode, sharing)

vagrant@debian12:~$ sudo update-alternatives --config java
There is 1 choice for the alternative java (providing /usr/bin/java).

  Selection    Path                                         Priority   Status
------------------------------------------------------------
* 0            /usr/lib/jvm/java-17-openjdk-amd64/bin/java   1711      auto mode
  1            /usr/lib/jvm/java-17-openjdk-amd64/bin/java   1711      manual mode

Press <enter> to keep the current choice[*], or type selection number: 0

vagrant@debian12:~$ export JAVA_HOME=/usr/lib/jvm/java-17-openjdk-amd64/
vagrant@debian12:~$ export PATH=$JAVA_HOME/bin:$PATH
vagrant@debian12:~$ source ~/.bashrc
vagrant@debian12:~$ echo $JAVA_HOME
/usr/lib/jvm/java-17-openjdk-amd64/
vagrant@debian12:~$ echo $PATH
/usr/lib/jvm/java-17-openjdk-amd64//bin:/home/vagrant/.local/bin:/usr/local/bin:/usr/bin:/bin:/usr/lo
cal/games:/usr/games
vagrant@debian12:~$ sudo apt install maven
Reading package lists... Done
Building dependency tree... Done
Reading state information... Done
maven is already the newest version (3.8.7-1).
0 upgraded, 0 newly installed, 0 to remove and 77 not upgraded.
```

Running maven

Open a shell and type `mvn archetype:generate`. This lets you *generate an artifact from an archetype*, which is maven-speak for create a new folder with a maven file.

If you get a "not found" error, then most likely the maven `bin` folder is not on your path. If you're on a POSIX system and have used your package manager, this should be set up automatically, but if you've downloaded and unzipped maven then you have to `export PATH="$PATH:..."` where you replace the three dots with the path to the folder, and preferably put that line in your `~/.profile` too.

The first time you run it, maven will download a lot of libraries.

Maven will first show a list of all archetypes known to humankind (3046 at the time of counting) but you can just press ENTER to use the default, 2098 ("quickstart"). Maven now asks you for the version to use, press ENTER again.

You now have to enter the triple of (groupId, artifactId, version) for your project - it doesn't really matter but I suggest the following:

```
groupId: org.example
artifactId: project
version: 0.1
```

Just press ENTER again for the following questions, until you get a success message.

Maven has created a folder named after your artifactId, but you can move and rename it if you want and maven won't mind as long as you run it from inside the folder. Use `cd project` or whatever you called it to go inside the folder.

If you're in a POSIX shell, then `find .` should show everything in the folder (in Windows, `start .` opens it in Explorer instead):

```
.
./src
./src/main
./src/main/java
./src/main/java/org
./src/main/java/org/example
./src/main/java/org/example/App.java
./src/test
./src/test/java
./src/test/java/org
./src/test/java/org/example
./src/test/java/org/example/AppTest.java
./pom.xml
```

This is the standard maven folder structure. Your java sources live under `src/main/java`, and the default package name is `org.example` or whatever you put as your groupId so the main file is currently `src/main/java/org/example/App.java`. Since it's common to develop Java from inside an IDE or an editor with "folding" for paths (such as VS code), this folder structure is not a problem, although it's a bit clunky on the terminal.

The POM file

Have a look at `pom.xml` in an editor. The important parts you need to know about are:

The artifact's identifier (group id, artifact id, version):

```
<groupId>org.example</groupId>
<artifactId>project</artifactId>
<version>0.1</version>
```

The build properties determine what version of Java to compile against (by passing a flag to the compiler). Unfortunately, the default maven template seems to go with version 7, but version 8 was released back in 2014 which is stable enough for us, so please change the 1.7 to 1.8:

```
<properties>
    <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
    <maven.compiler.source>1.8</maven.compiler.source>
    <maven.compiler.target>1.8</maven.compiler.target>
</properties>
```

The dependencies section is where you add libraries you want to use. By default, your project uses `junit`, a unit testing framework - note that this is declared with `<scope>test</scope>` to say that it's only used for tests, not the project itself. You do not add this line when declaring your project's real dependencies.

```
<dependencies>
    <dependency>
        <groupId>junit</groupId>
        <artifactId>junit</artifactId>
        <version>4.11</version>
        <scope>test</scope>
    </dependency>
</dependencies>
```

The `<plugins>` section contains the plugins that maven uses to compile and build your project. This section isn't mandatory, but it's included to "lock" the plugins to a particular version so that if a new version of a plugin is released, that doesn't change how your build works.

The one thing you should add here is the `exec-maven-plugin` as follows, so that you can actually run your project:

```
<plugin>
    <groupId>org.codehaus.mojo</groupId>
    <artifactId>exec-maven-plugin</artifactId>
    <version>3.0.0</version>
    <configuration>
        <mainClass>org.example.App</mainClass>
    </configuration>
</plugin>
```

The important line is the `mainClass` which you set to the full name (with path components) of your class with the `main()` function.

```
vagrant@debian12:~/project$ cat pom.xml
<?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-instan
ce"
  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.example</groupId>
  <artifactId>project</artifactId>
  <version>0.1</version>

  <name>project</name>
  <!-- FIXME change it to the project's website -->
  <url>http://www.example.com</url>

  <properties>
    <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
    <maven.compiler.release>17</maven.compiler.release>
  </properties>

  <dependencyManagement>
    <dependencies>
      <dependency>
        <groupId>org.junit</groupId>
        <artifactId>junit-bom</artifactId>
        <version>5.11.0</version>
        <type>pom</type>
        <scope>import</scope>
      </dependency>
    </dependencies>
  </dependencyManagement>

  <dependencies>
    <dependency>
      <groupId>org.junit.jupiter</groupId>
      <artifactId>junit-jupiter-api</artifactId>
      <scope>test</scope>
    </dependency>
    <!-- Optionally: parameterized tests support -->
    <dependency>
      <groupId>org.junit.jupiter</groupId>
      <artifactId>junit-jupiter-params</artifactId>
      <scope>test</scope>
    </dependency>
  </dependencies>

  <build>
    <pluginManagement><!-- lock down plugins versions to avoid using Maven defaults (may be moved to
parent pom) -->
      <plugins>
        <!-- clean lifecycle, see https://maven.apache.org/ref/current/maven-core/lifecycles.html#cle
```

```
an_Lifecycle -->
        <plugin>
          <artifactId>maven-clean-plugin</artifactId>
          <version>3.4.0</version>
        </plugin>
        <!-- default lifecycle, jar packaging: see https://maven.apache.org/ref/current/maven-core/de
fault-bindings.html#Plugin_bindings_for_jar_packaging -->
        <plugin>
          <artifactId>maven-resources-plugin</artifactId>
          <version>3.3.1</version>
        </plugin>
        <plugin>
          <artifactId>maven-compiler-plugin</artifactId>
          <version>3.13.0</version>
        </plugin>
        <plugin>
          <artifactId>maven-surefire-plugin</artifactId>
          <version>3.3.0</version>
        </plugin>
        <plugin>
          <artifactId>maven-jar-plugin</artifactId>
          <version>3.4.2</version>
        </plugin>
        <plugin>
          <artifactId>maven-install-plugin</artifactId>
          <version>3.1.2</version>
        </plugin>
        <plugin>
          <artifactId>maven-deploy-plugin</artifactId>
          <version>3.1.2</version>
        </plugin>
        <!-- site lifecycle, see https://maven.apache.org/ref/current/maven-core/lifecycles.html#site
_Lifecycle -->
        <plugin>
          <artifactId>maven-site-plugin</artifactId>
          <version>3.12.1</version>
        </plugin>
        <plugin>
          <artifactId>maven-project-info-reports-plugin</artifactId>
          <version>3.6.1</version>
        </plugin>
        <plugin>
          <groupId>org.codehaus.mojo</groupId>
          <artifactId>exec-maven-plugin</artifactId>
          <version>3.0.0</version>
          <configuration>
            <mainClass>org.example.App</mainClass>
          </configuration>
        </plugin>
      </plugins>
    </pluginManagement>
  </build>
</project>
```

Compile, run and develop

`mvn compile` compiles the project. The very first time you do this, it will download a lot of plugins, after that it will be pretty fast. Like `make`, it only compiles files that have changed since the last run, but if this ever gets out of sync (for example because you cancelled a compile halfway through) then `mvn clean` will remove all compiled files so the next compile will rebuild everything.

```
vagrant@debian12:~/project$ mvn compile
```

The `App.java` file contains a basic “Hello World!” program (have a look at this file). You can run the compiled project with `mvn exec:java` if you’ve set up the plugin as above. After you’ve run it the first time and it’s downloaded all the files it needs, lines coming from maven itself will start with `[INFO]` or `[ERROR]` or similar, so lines without any prefix like that are printed by your program itself. You should see the hello world message on your screen.

```
vagrant@debian12:~/project$ mvn exec:java
vagrant@debian12:~/project$ mvn compile test exec:java
vagrant@debian12:~/project$ mvn test
vagrant@debian12:~/project$ mvn package
```

The development workflow is now as follows: you make your edits, then run `mvn compile test exec:java` to recompile, run your tests, then run the program. `mvn test` runs the tests in `src/test/java`. `mvn package` creates a jar file of your project in the `target/` folder.

I assume that you will be storing your Java projects in git repositories. In this case, you should create a file `.gitignore` in the same folder as the `pom.xml` and add the line `target/` to it, since you don't want the compiled classes and other temporary files and build reports in the repository. The `src/` folder, the `pom.xml` and the `.gitignore` file itself should all be checked in to the repository.

Adding a dependency

Thymeleaf is a Java templating library. It lets you write a template file or string for example

```
Hello, ${name}!
```

which you can later render with a particular name value. This is one of the standard ways of creating web applications, for example to display someone's profile page you would write a page template that takes care of the layout, styles, links etc. but uses template variables for the fields which you render when someone accesses the profile page for a particular person.

To use Thymeleaf or any other library, you first have to add it to your pom file. Go to mvnrepository.org and search for Thymeleaf, then find the latest stable ("release") version. There is a box where you can copy the `<dependency>` block to paste in your pom file. The next `mvn compile` will download thymeleaf and all its dependencies.

```
<!-- https://mvnrepository.com/artifact/org.thymeleaf/thymeleaf -->
<dependency>
    <groupId>org.thymeleaf</groupId>
    <artifactId>thymeleaf</artifactId>
    <version>3.1.2.RELEASE</version>
</dependency>
```