

CS 398: Application Development

Software Releases

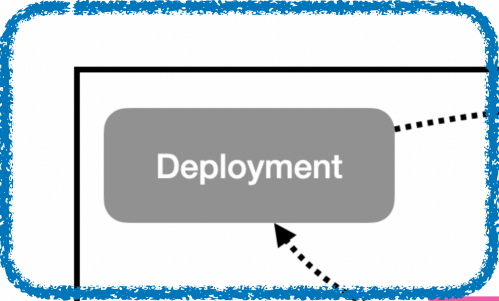
Copyright; Licensing; Packaging; Software release process

Planning

Requirements

Analysis & Design

Preliminary activities
(fixed-duration blocks)



Deployment

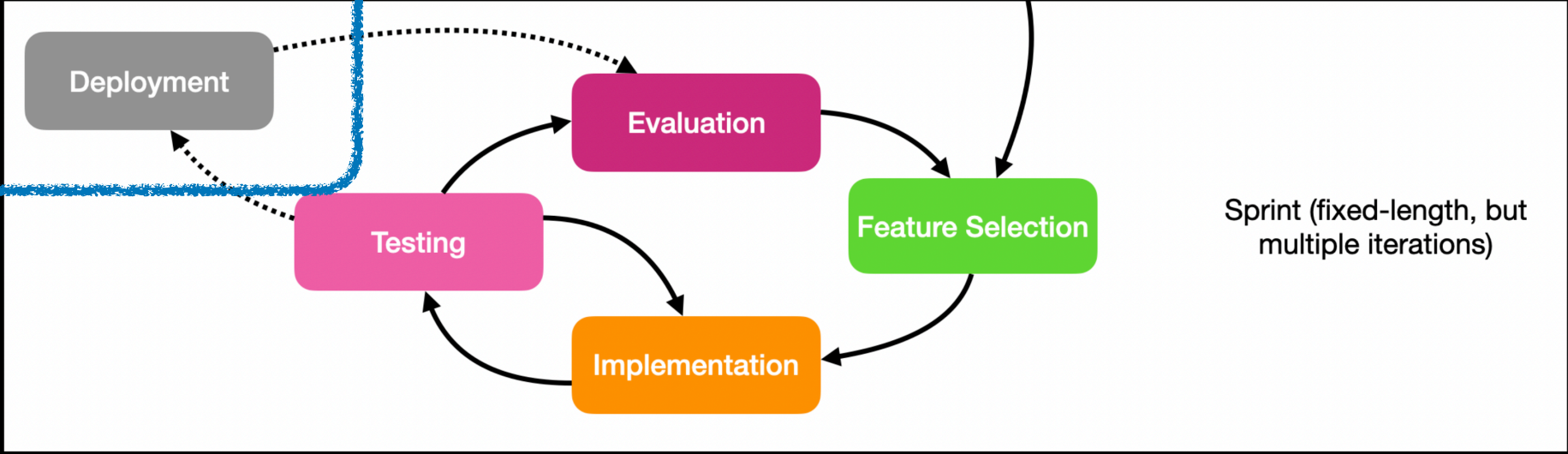
Evaluation

Feature Selection


Sprint (fixed-length, but
multiple iterations)

Testing

Implementation



The final step to getting our software into the hands of our users is **software distribution**. This can take many forms, depending on the environment in which the software needs to operate.

- 
1. **We sometimes need to install the software into a client's environment.** With complex or specialized software, we may need to install and configure it. e.g. we might write medical software that needs to integrate with existing patient data or billing systems.
 2. **We might sell packaged software,** for distribution through a retail chain. e.g. shrink-wrapped games, or firmware updates for the latest carOS.
 3. **Sometimes software is distributed from a website, or digital distribution channel.** e.g. **Visual Studio Code**, a third-party distributor e.g. **SetApp**, or a package manager e.g. homebrew, winget.
 4. **Distribution through a user-facing application store is very common.** e.g. **Steam** for distributing games; the **Apple App Store** or **Google Play Store** for mobile and desktop applications. This is really a specialized case of digital distribution.

Each of these deployment mechanisms will require software to be prepared in a different way.

e.g. selling packaged software will require a DVD image that includes digital assets. Selling from an online application store required code signing, going through a code review process with the company that runs the store, and so on.

However, there are a number of issues to sort out prior to releasing our software.

- **Legal:** how do we protect ourselves from unauthorized use, or further redistribution of our software?
- **Usability:** how do we ensure that users know how to use our software? do we provide documentation or training?
- **Logistical:** how do we package our software so that we can distribute it, and users can successfully obtain and install it?

Copyright

Before distributing your software, in any form, it's critical to establish ownership and rights pertaining to that software.

A [copyright](#) is a type of intellectual property that gives its owner the exclusive right to copy and distribute a [creative work](#), usually for a limited time. Copyright is intended to protect the original expression of an idea in the form of a creative work, but not the idea itself. A copyright is subject to [limitations](#) based on public interest considerations, such as the [fair use](#) doctrine.

[Software copyright](#) is the application of [copyright](#) in [law](#) to [machine-readable software](#). Under Canadian and US law, all software is [copyright](#) protected, in both [source code](#) and [object code](#) forms.

Practically, this means that different companies can independently produce software that solves the same problem, and there is no law preventing that from occurring, provided that they do not reuse actual source or object code from their competitor.

How do I assert copyright?

In Canada, software is protected under the [Copyright Act of Canada](#).

Copyright is acquired automatically when an original work is generated; **the creator is not required to register or mark the work with the copyright symbol in order to be protected.** The rights holder is granted: the exclusive right of reproduction, the right to rent the software, the right to restrain others from renting the software and the right to assign or license the copyright to others.

It's **common practice** to assert your copyright claim in the header of your software source files. Although is not required to assert copyright, it's a flag for potential violators, and it might make it easier to defend in court.

Copyright (c) 2022. Jeff Avery.

See U Waterloo's IP policy.
<https://uwaterloo.ca/secretariat/policies-procedures-guidelines/policies/policy-73-intellectual-property-rights>

Software Licensing

As the rights holder, you can grant others rights with respect to your software.

A **software license** is a legal instrument that grants the licensee (i.e. an end-user) permission to use the software in a manner dictated by the license. These rights could include (but aren't limited to)

- the right to install and use it,
- the right to modify the source code, or
- the right to redistribute the software with or without changes.

Authors of copyrighted software can also choose to donate their software to the public domain, in which case it is also not covered by copyright and, as a result, cannot be licensed.

	Permissive	Copyleft	Noncommercial	Proprietary
Description	Grants use rights, including right to relicense.	Grants use rights, forbids proprietization .	Grants rights for noncommercial use only.	Traditional use of copyright ; no rights need be granted.
Examples	MIT , Apache , MPL	GPL , AGPL	JRL , AFPL	Proprietary software , no public license

A [permissive software license](#), sometimes also called BSD-style is a free-software license which instead of copyleft protections, carries only minimal restrictions on how the software can be used, modified, and redistributed, usually including a warranty disclaimer.

[Copyleft](#) is the practice of granting the right to freely distribute and modify intellectual property with the requirement that the same rights be preserved in derivative works created from that property. Copyleft software licenses require that information necessary for reproducing and modifying the work must be made available to recipients.

[Non-commercial licenses](#) are intended to be used only by entities with no profit motive, including charities and public institutions.

How do I apply a license?

1. Distribute the license with your program.

- Include a `license.txt` file in your distribution.
- If you provide source code to anyone, include a statement about how it is licensed in the header of each file. Check the license to see what is required.

2. Include a licensing statement on your website.

- See terms of each license to check what's suitable.

Unless explicitly stated otherwise all files in this repository are licensed under the Apache Software License 2.0 [insert boilerplate notice here]

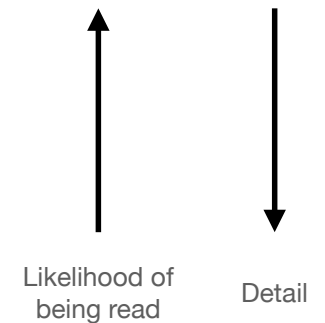
For projects, a simple license is fine. For commercial purposes? Consult a lawyer!

Documentation

User Documentation

If your product is complex, or if you want a way to showcase your features, you might consider producing different types of user documentation:

- **Getting started guide** (web page, PDF): the basics to use your software
- **Tutorials** (web pages): walk through simple tasks with examples
- **User Guide**: comprehensive documentation



This all takes effort. Tailor your documentation to the complexity of your product (and the likelihood of users actually taking the time to read it!)

Documentation can be printed and bundled (uncommon), distributed with your software (PDF, other formats, more common) or hosted on your website (very common).

Release Notes

Every public release should include release notes: a list of changes that you have made to your product for that particular release. Depending on the nature of your product, and your relationship with your users, the details can be fairly general ("added support for Windows 11") or incredibly detailed ("fixed bug XXX").

The release notes are also a good place to put things like

- OS or hardware compatibility details
- information on how users can contact you or get help

Release notes are typically released in one of the following ways:

- a `readme.txt` file included in your distribution.
- a popup window in your application that includes these details.
- a page on your website (if you do this, put a way to open that page from within your application).

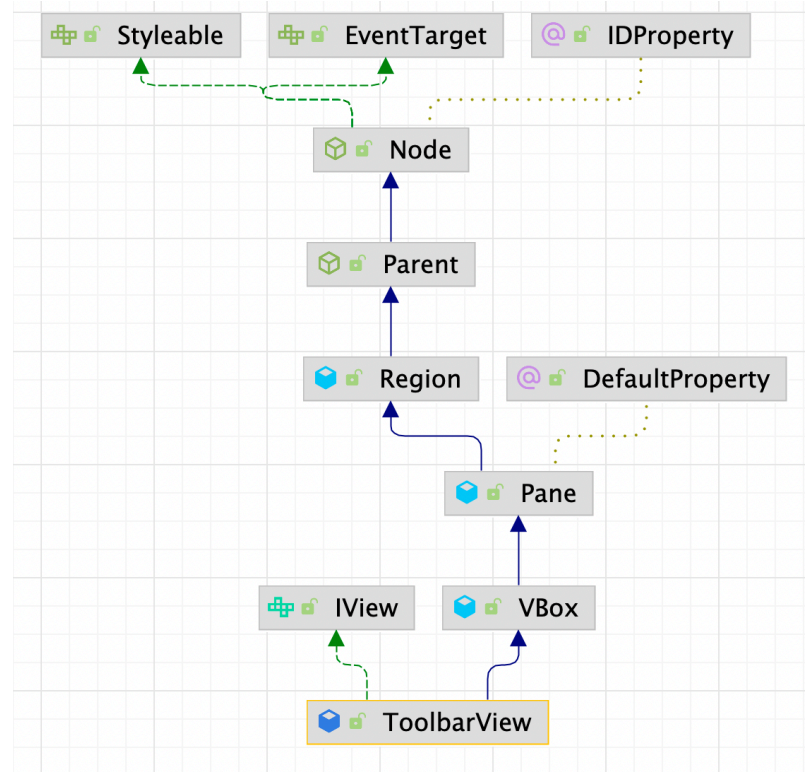
Suggestions

Starting points for writing maintainable documentation:

- Consider Markdown as an authoring format. You can then convert markdown to PDF, EPUB, or HTML for publishing using [pandoc](#). The course website is authored in Markdown and then converted to a website using [Hugo](#).

If you need to produce documentation for other developers:

- [Dokka](#) can generate HTML documentation from Kotlin code (much as [Javadoc](#) does for Java code).
- IntelliJ IDEA can generate class diagrams from source code. Right-click on the project file, then select Diagrams from the popup menu.



Packaging

Packaging

Packaging just refers to putting together your software components in a way that you can distribute to your end-user.

As we've seen, this could be physical (a retail box including a DVD), or a digital image (ISO image, package file, installer executable).

Packaging is the "bundling" part of building software. Let's focus on digital distribution.

Applications

What does packaging entail?

Packaging is also one of those tasks that we tend to hand wave: just compile, link and hand-out the executable right? Unfortunately it's not that simple.

Preparing a package includes

- Compiling and linking your application.
- Preparing images, sound clips, documentation, other resources to include with your application.
- Writing or configuring software to install the application, which includes
 - Copying binaries and resources to the correct locations
 - Registering services and resources with the OS
 - Installing applications in the correct system location
 - Creating application icons

Packaging services may be different. We'll discuss that later.

Packaging Tools

This is far too tedious to do by-hand, but luckily we have some tools to perform some of these actions for us. Here's what we need to do to make our software *installable*.

Step	Explanation
Step 1: Compiling classes	Use the <code>kotlinc</code> compiler to create classes from our source code.
Step 2: Creating archives	Use the <code>jar</code> command to create jar files of classes and related libraries.
Step 3: Creating scripts	Optionally, create scripts to allow a user to execute directly from the JAR files.
Step 4: Creating an installer	Use <code>jpackage</code> to create platform specific installers.


1. Compiling Code

To compile from the command-line, we can use the Kotlin compiler, `kotlinc`. By default, it takes Kotlin source files (`.kt`) and compiles them into corresponding class files (`.class`) that can be executed on the JVM.

```
$ kotlinc Hello.kt
```

```
$ ls  
Hello.kt HelloKt.class
```

```
$ kotlin HelloKt  
Hello Kotlin!
```



This is the easy part, just including for completeness.

We could also just do this with IntelliJ IDEA or Android Studio, where Gradle – build – build will generate class files.

As we will see at the end of this section, we can often just generate the final installer in a single step without doing each step manually.

2. Creating an archive

A JAR file is just a compressed file (just like a ZIP file) which has a specific structure and contents. Most distribution mechanisms expect a JAR file.

```
$ kotlinc Hello.kt -include-runtime -d Hello.jar
```

```
$ ls  
Hello.jar Hello.kt
```

- The `-d` option tells the compiler to package all of the required classes into our jar file.
- The `-include-runtime` flag tells it to also include the Kotlin runtime classes. These classes are needed for all Kotlin applications, so you should always include them in your distribution.

To run from a jar file, use the `java` command:

```
$ java -jar Hello.jar  
Hello Kotlin!
```

JAR File Contents

```
$ unzip Hello.jar -d contents
Archive:  Hello.jar
  inflating: contents/META-INF/MANIFEST.MF
  inflating: contents/HelloKt.class
  inflating: contents/META-INF/main.kotlin_module
  inflating: contents/kotlin/collections/ArraysUtilJVM.class
  ...
```

```
$ tree -L 2 contents/
.
├── META-INF
│   ├── MANIFEST.MF
│   ├── main.kotlin_module
│   └── versions
└── kotlin
    ├── ArrayIntrinsicsKt.class
    ├── BuilderInference.class
    ├── DeepRecursiveFunction.class
    ├── DeepRecursiveKt.class
    ├── DeepRecursiveScope.class
    ...
```

The JAR file contains these main features:

- `HelloKt.class` – a class wrapper generated by the compiler
- `META-INF/MANIFEST.MF` – a file containing metadata.
- `kotlin/` – Kotlin runtime classes not included in the JDK.

```
$ cat contents/META-INF/MANIFEST.MF
Manifest-Version: 1.0
Created-By: JetBrains Kotlin
Main-Class: HelloKt
```

The manifest file contains metadata, like which class to run first. It's similar in function to the Android Manifest file.

3. Creating scripts

We can distribute JAR files like this to our users, but they're awkward: users would need to have the Java JDK installed, and type `java -jar filename.jar` to actually run our programs.

The simplest thing we can do is create a script to launch our application from a JAR file, with the same effect as executing from the command-line:

```
$ cat hello
#!/bin/bash
java -jar hello.jar
```

```
$ chmod +x hello
```

```
$ ./hello
Hello Kotlin!
```

For simple applications, especially ones that we use ourselves, this may be sufficient. It has the downside of requiring the user to have the Java JDK installed on their system

Generating scripts

For a more robust script, we can let Gradle generate one for us. In IntelliJ IDEA, Gradle – distribution – distZip will create a zip file that includes a custom runtime script.

```
$ tree build/distributions -L 3
build/distributions
├── app
│   ├── bin
│   │   ├── app
│   │   └── app.bat
│   └── lib
│       ├── annotations-13.0.jar
│       ├── app.jar
│       ├── checker-qual-3.8.0.jar
│       ├── error_prone_annotations-2.5.1.jar
│       ├── failureaccess-1.0.1.jar
│       ├── guava-30.1.1-jre.jar
│       ├── j2objc-annotations-1.3.jar
│       ├── jsr305-3.0.2.jar
│       ├── kotlin-stdlib-1.5.31.jar
│       ├── kotlin-stdlib-common-1.5.31.jar
│       ├── kotlin-stdlib-jdk7-1.5.31.jar
│       ├── kotlin-stdlib-jdk8-1.5.31.jar
│       └── listenablefuture-9999.0-empty-to-avoid-conflict-with-guava.jar
└── app.zip
```

Public repo
Console/Archey

Running from a script

The bin/app script is quite robust, and will handle many different types of system configurations.

```
# OS specific support (must be 'true' or 'false').
```

```
cygwin=false
```

```
msys=false
```

```
darwin=false
```

```
nonstop=false
```

```
case "$( uname )" in          #(
```

```
  CYGWIN* )      cygwin=true  ;; #(
```

```
  Darwin* )     darwin=true  ;; #(
```

```
  MSYS* | MINGW* ) msys=true   ;; #(
```

```
  NONSTOP* )    nonstop=true ;;
```

```
esac
```

```
CLASSPATH=$APP_HOME/lib/app.jar:$APP_HOME/lib/kotlin-stdlib-jdk8-1.5.31.jar:$APP_HOME/lib/guava-30.1.1-jre.jar:$APP_HOME/lib/kotlin-stdlib-jdk7-1.5.31.jar:$APP_HOME/lib/kotlin-stdlib-1.5.31.jar:$APP_HOME/lib/kotlin-stdlib-common-1.5.31.jar:$APP_HOME/lib/failureaccess-1.0.1.jar:$APP_HOME/lib/listenablefuture-9999.0-empty-to-avoid-conflict-with-guava.jar:$APP_HOME/lib/jsr305-3.0.2.jar:$APP_HOME/lib/checker-qual-3.8.0.jar:$APP_HOME/lib/error_prone_annotations-2.5.1.jar:$APP_HOME/lib/j2objc-annotations-1.3.jar:$APP_HOME/lib/annotations-13.0.jar
```

```
# Determine the Java command to use to start the JVM.
```

```
if [ -n "$JAVA_HOME" ] ; then
```

```
  if [ -x "$JAVA_HOME/jre/sh/java" ] ; then
```

```
    . . .
```


JavaFX from a script

Scripts like this are really only suitable for console applications i.e. applications without a GUI.

If you are building a JavaFX or Compose desktop application, you need to either use `jlink` or `jpackage` to build an installer.

JLink will let you build a custom runtime that will handle the module dependencies for JavaFX. The simplest way to do this is to add the JLink plugin to your `build.gradle` file and let Gradle handle it.

```
plugins {  
    id 'org.beryx.jlink' version '2.25.0'  
}  
  
jlink{  
    launcher {  
        name = "clock"  
    }  
    imageZip.set(project.file("${project.buildDir}/image-zip/clock-image.zip"))  
}
```

For details see the plugin page: <https://badass-jlink-plugin.beryx.org/releases/latest/>

Clock running from a script

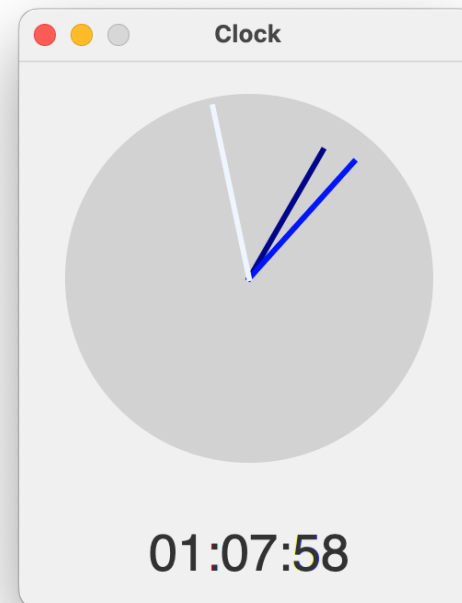
We can rebuild the clock sample using `Gradle - build - jLink` to produce a runtime script in `build/image`

Here's the resulting directory structure. Notice that it includes a number of libraries that our application needs to run.

```
$ tree build/image -L 2
AnalogClock/build/image
├── bin
│   ├── clock_advanced
│   ├── clock_advanced.bat
│   ├── java
│   ├── jrunscript
│   └── keytool
├── conf
│   ├── net.properties
│   ├── security
│   └── sound.properties
├── include
│   ├── classfile_constants.h
│   ├── darwin
│   ├── jawt.h
│   ├── jni.h
│   ├── jvmti.h
│   └── jvmticmlr.h
... (continues)
```

Running the top-level `bin/clock_advanced` image will execute our application.

```
$ ./clock_advanced
```



Android from a script?

Android users should not use either of these methods, since the Android OS doesn't support launching applications from a console, and it doesn't need an installer.

Instead, Android users will want to create an APK: a bundle that contains the application and all of its related files. (APK stands for Android Package Kit).

In Android Studio or IntelliJ IDEA, in your Android project, you can select Build - Build Bundle/APK. This will produce a file that can be side-loaded.

To install the APK file:

- Plugin your phone with a USB cable.
- Double-click the file to install it to the connected device.
- If that fails: install the Command-Line tools from the SDK manager. Use the adb tool from the console to install the APK file to your connected device.

4. Create installers

Finally, we can use `jpackage` to create native installers for a number of supported operating systems.

An installer is an application that can install something else, like our application. Installers can handle the complexities of installation that would be difficult to do with scripts alone.

Tasks that the installer performs include:

- Copying application files to the correct location.
- Installing and registering system libraries.
- Making changes to the system registry (or similar system databases).
- Creating icons on the desktop, or applications folder.
- Prompting the user if any of these tasks require elevated privileges.
- Installing extra components, like `readme.txt` or `license.txt`

JPackage is not the only one, there are commercial installers as well. e.g. JDeploy.

JLink & JPackage

Instead of running `jpackage` manually, we will install a plugin into IntelliJ and use that environment to generate our installers. We can do this by installing the [Badass-JLink plugin page](#). To use the plugin, include the following in your `gradle.build` script:

```
plugins {
    id 'org.beryx.jlink' version '2.25.0'
}
```

JPackage itself has a number of other options that you can specify in the `build.gradle` file.

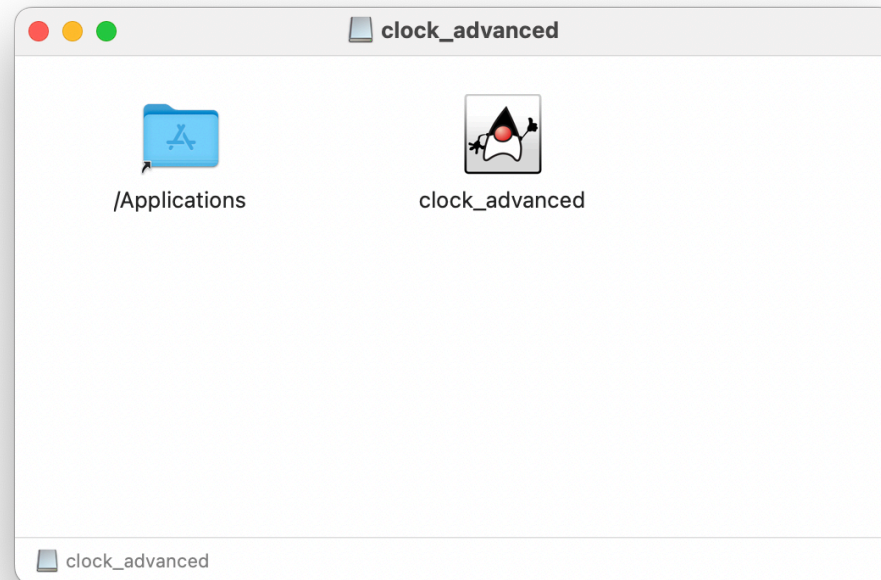
```
// build.gradle file options for jpackage
jlink {
    options = ['--strip-debug', '--compress', '2', '--no-header-files', '--no-man-pages']
    launcher{
        name = 'hello'
        jvmArgs = ['-Dlog4j.configurationFile=./log4j2.xml']
    }
}
```

If you install the plugin correctly, then you should see the `jpackage` command in `Gradle - build - jpackage`. Run this and it will create platform installers in the `build/distribution` directory.

If you install the plugin correctly, then you should see the `jpackage` command in Gradle – `build – jpackage`. Run this and it will create platform installers in the `build/distribution` directory.

This is a standard macOS installer.

Drag the `clock_advanced` icon to the Applications folder. You can then run it from that folder.



Services

Unlike applications, which are hosted by users on their own systems, services are typically deployed on servers. These can be physical systems, VMs or containers running in the cloud, or any combination of these targets.

Web services, the type that we've been considering, need to be deployed to a web server.

When we were building projects with Spring Boot, it quietly launched a web server in the background to support us testing our application. To deploy in a production environment though, we would need to install our application in an environment where a web server is already installed.

From Spring, we can produce one of two packages:

- **WAR file:** a Web Application Archive (WAR) file is a standard deployment package for services that run on a web server.
- **JAR file:** produce a standalone JAR file and deploy it (see instructions above).

AWS

Spring Boot's executable jars are ready-made for most popular cloud PaaS (Platform-as-a-Service) providers. However, you may need to adapt *your* application to the *cloud's* notion of a running process, depending on the platform.

Amazon Web Services offers multiple ways to install Spring Boot-based applications, either as traditional web applications (war) or as executable jar files with an embedded web server.

e.g. AWS Elastic Beanstalk, AWS Code Deploy, AWS OPS Works.

AWS Elastic Beanstalk allows deploying a WAR file directly to a Tomcat Platform (a well-known web/application server), or the "Java SE Platform". For web services, we would follow their documentation for configuring the cloud platform, and then upload the WAR file that we produce from IntelliJ.

This is basically trivial, and a great way to deploy services.

App Stores

An Application Store is a very unique approach to distributing software that fulfills many of the same requirements as our client packaging models, but offers additional functionality.

- App stores may want to install the software for you, to ease the user experience. This is likely more complex than just downloading and auto-running the installer. e.g. Apple or Google app stores.
- They likely include some form of user authentication, and they will track licensing of the software to you.
- They will want to charge you - either against store credit, or a credit card.
- They are not restricted to a particular type of software. I've personally used app stores to install everything from games to mobile apps to operating systems.
- They are completely proprietary. We can't point to any one service and apply our knowledge of how it works to deploying on a different service.
- You will need proprietary tools to work in one of those environments.

For all of these reasons, we'll avoid delving too deep into app stores. However, if you want to deploy software commercially, you should expect to become an expert in all of the major platforms.

What is our outcome?

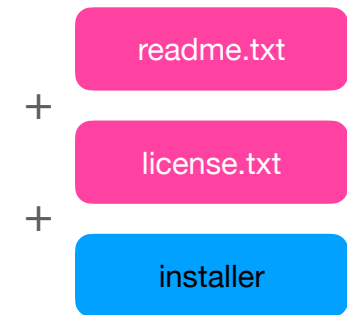
We expect the output of this process to be an installer (or scripts/custom deliverable) for each platform.

- Installer
 - Installs your software
 - Installs libraries
 - Sets the application icon
 - Creates shortcuts
 - Installs `readme.txt`
 - Installs `license.txt`

Release Process

What do we need to do for a release?

- Update our internal project tracking systems
 - Close the sprint
 - Update and close issues
 - Update our internal documentation e.g. wiki
- Prepare additional materials for the release
 - Release notes
 - User documentation
 - Update build scripts/configuration
 - Build installers



These are all prepared as part of the software release process!

Versioning

You should version your software, so that every release has a release number and date associated with it.

The standard convention is a triple, separated by decimals, of the format: `major.minor.build`. For example, 1.2.3 would be major version 1, minor version 2, build 3.

- **Major signifies a major product release.** This is somewhat arbitrary, but typically is released infrequently and includes major features changes or additions. If you charge by release, you would typically charge for every new major version. You might release a new major version as frequently as once per year, or as infrequently as once very few years.
- **Minor indicates a minor product release,** typically a combination of new minor features, and bug or compatibility fixes. You might release a minor version a few times per year and users would not ordinarily expect to pay for these.
- **Build number is internal build number** within a minor release. This is intended to reflect bug fixes only; you typically iterate over builds internally and release the final successful version publicly.

You should also use version numbers in your internal documentation: track the version number with the sprint, and use it when identifying bugs (e.g. what version number were you using for testing? What version number will include the fix?)

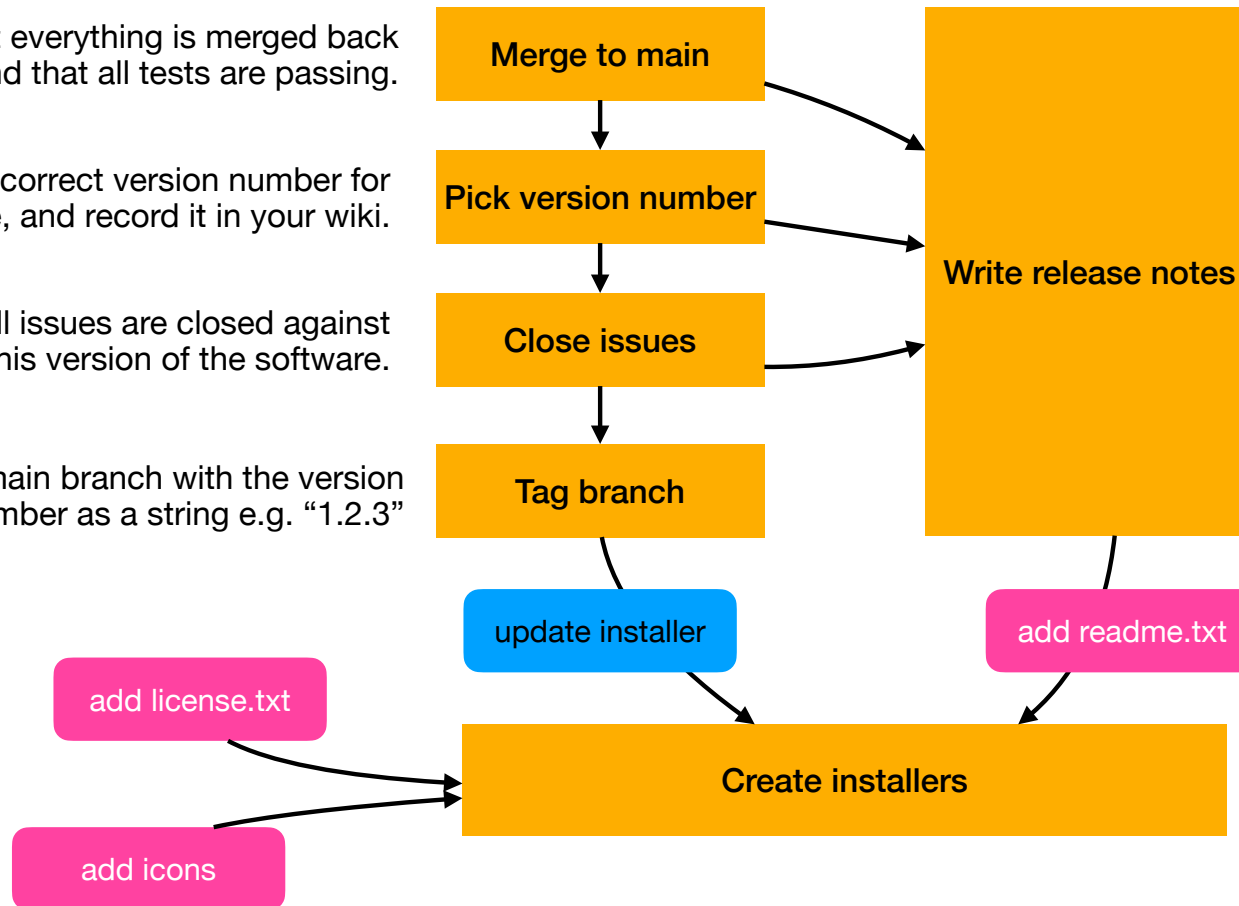
Release Process

Make sure that everything is merged back to main, and that all tests are passing.

Determine the correct version number for this release, and record it in your wiki.

Make sure all issues are closed against this version of the software.

Tag the main branch with the version number as a string e.g. "1.2.3"



The release notes should contain relevant product release detail:

- * Release date
- * Version number
- * Details of major changes that are included.
- * (Optional) List of known issues or limits.